Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes

Albert Magyar, David Biancolin, John Koenig, Sanjit Seshia, Jonathan Bachrach, Krste Asanović
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
{albert.magyar,biancolin,jack.koenig3,sseshia,jrb,krste}@berkeley.edu

ABSTRACT

We present Golden Gate, an FPGA-based simulation tool that decouples the timing of an FPGA host platform from that of the target RTL design. 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 subcomponents, 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. To verify that these optimizations are implemented correctly, we also present lime, a model-checking tool that provides a push-button flow for checking whether optimized subcomponents adhere to an associated correctness specification, while also guaranteeing forward progress. Finally, we use Golden Gate to generate a cycle-exact simulator of a multi-core SoC, where we reduce LUT utilization by up to 26% by coercing multi-ported, combinationally read memories into simulation models backed by time-multiplexed block RAMs, enabling us to simulate 50% more cores on a single FPGA.

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 higher up in the computing stack. At the same time, myriad emerging applications, in domains like AI, virtual and augmented reality, and the Internet of Things, depend on the availability of higher-performance, more efficient computing systems. As a result, system architects have turned to specialization: in modern SoCs, application cores increasingly yield area to specialized accelerators [8]. However, this specialization begets complexity that makes these systems harder to build, verify, and program. This drives the non-recurring engineering (NRE) costs of developing custom silicon out of reach for 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.

FPGAs have long been used for prototyping and emulation of ASICs in both industry [16] and academia [13, 26]. 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 signficant effort to meet resource constraints. Commercial *emulation platforms* offer automated scaling to larger designs, but suffer from high cost of entry and pay a large performance cost when partitioning designs across many FPGAs. While manually time-multiplexed simulators such as RAMP Gold [23] present large increases in per-FPGA capacity, they require excessive engineering effort. Recent academic simulation platforms like FireSim [13] omit capacity-enhancing optimizations to focus on providing fully automated, open-source platforms for co-simulation and debugging of networked devices.

In this paper, we contribute Golden Gate, an open-source tool to enhance FPGA emulation capacity through automatic resource optimization of FPGA-hosted simulation models derived from ASIC RTL. These optimizations trade simulation time (FPGA cycles) for FPGA resources, allowing more of the ASIC to fit on a single FPGA. While this technique could be applied in a partitioned setting, it also enables commodity FPGA boards to elastically scale in capacity to avoid the economic or performance cost of partitioning. While prior work manually applied these optimizations to develop abstract processor models, Golden Gate is the first to apply them automatically. We also contribute lime, a push-button checker that verifies that an optimized model simulates its ASIC source exactly. Finally, we perform a case study, where Golden Gate and lime are used to time-multiplex highly ported memories, optimizing a traditionally FPGA-hostile element of ASIC designs [27]. This enables us to emulate a large RISC-V multiprocessor SoC on a single FPGA, where a partitioned prototype would otherwise have been required. To facilitate wider use, Golden Gate has been released as a new, optimizing compiler for the FireSim simulation platform.

2 PRIOR WORK IN FPGA EMULATION

Pre-silicon evaluation of ASICs has long been a core application for FPGAs [7]. 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 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]. 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 [9].

978-1-7281-2350-9/19/$31.00 ©2019 IEEE

(2) *Resource conversions*: ASIC power, reset, and clocking structures do not map directly to the host FPGA and must be replaced [1, 10].
(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.

2.2 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 [16]. These tools build on advances in inter-chip routing [10, 14] and time-multiplexing of pins [3] to reduce the speed and productivity overhead of using multi-FPGA host platforms. Furthermore, they may offer mechanisms for interfacing with I/O models that are co-simulated in a software environment [11], 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.

2.3 Decoupled FPGA-Accelerated Simulators

While computer architecture research has long relied on software simulators in lieu of complete RTL implementations, the RAMP [26] project aimed to use FPGAs to increase the speed and fidelity of microarchitectural simulations of manycore systems. To avoid the resource limitations of FPGA prototypes, RAMP simulators such as HASim [19] and RAMPGold [23] 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 [18] of *models*; we give an example in Figure 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. [5]. 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) [20]: 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].

[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.

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

2.4 Transformed Decoupled Simulators

To avoid the challenges of developing custom RTL timing models, recent work in decoupled simulation, such as FireSim [13], aims to strike a balance between prototyping and decoupled simulation. In these simulators, handwritten models are largely replaced with a single cycle-exact model that is *transformed* from ASIC RTL.

These model transformations have been simple: they automatically add handshaking interfaces that effectively "pause" the forward progress of the target to allow the model to stall while it waits for tokens. This technique enables co-simulation of network interfaces to model networks of thousands of target machines [13] and FPGA-accelerated modeling of the external DRAM interfaces of the target ASIC [4]. While this resembles the clock-gating approach used to support transactional emulation [11], the flexible decoupled interface with the target simplifies instrumentation to support power modeling and debugging features [15]. However, while the explicit decoupling of the target is similar to the RAMP simulators, the ASIC RTL used within the model is largely unchanged, yielding the same resource utilization challenges as FPGA prototypes.

3 ON COMPOSITIONAL SIMULATORS

With Golden Gate, we extend the notion of a decoupled simulator with *compositional simulation*, which enables RAMP-style resource optimizations to be automatically applied to subcomponents 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. While RAMP Gold [23] employed internal decoupling to connect a time-multiplexed processor model simulating 64 independent cores with the rest of the simulator, it did so in an ad-hoc manner. Automating this approach presents two main challenges:

(1) Decomposing the simulator into decoupled "islands" that may be independently optimized.
(2) Expressing optimizations—such as time-multiplexing—as transforms that modify the "islands" 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. Therefore, we model our system as a Latency-Insensitive Bounded Dataflow Network (LI-BDN) [25] to provide both correctness and forward progress guarantees.

3.1 The LI-BDN Target Formalism

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 subcomponents.

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

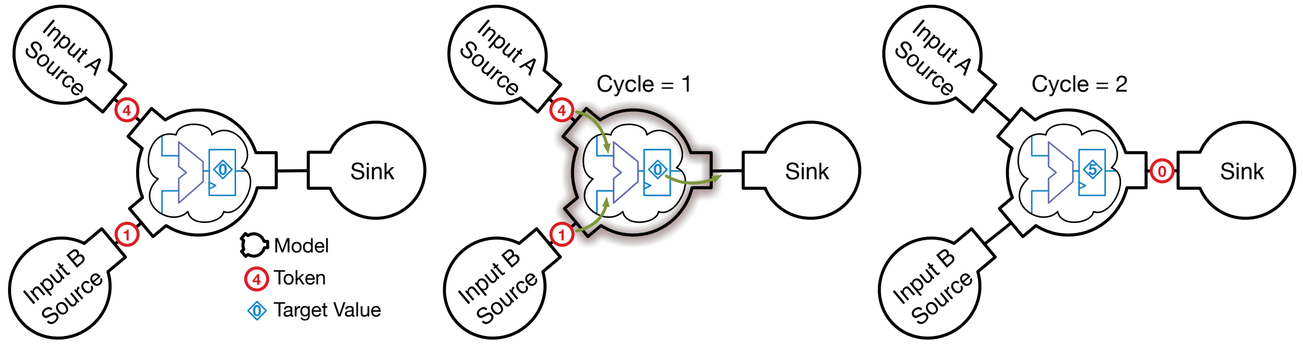

Figure 1: A 32-bit adder model and environment simulating a single cycle of target time.

(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*. To be a legal primitive LI-BDN, a module must obey the *No Extraneous Dependencies (NED)* and *Self-Cleaning (SC)* formal properties, described in detail in Section 5. In short, these properties specify when a primitive LI-BDN is obligated to produce and accept tokens over its I/O channels.

In the context of simulation, LI-BDNs allow arbitrary partitions of the input circuit to be mapped to latency-insensitive implementations, while still correctly modeling its synchronous behavior. Furthermore, it also defines rules of composition that guarantee that the composite LI-BDN will correctly simulate the full input design in aggregate. Finally, the specification for the constituent primitive LI-BDNs is presented as a concrete set of properties, which includes the NED and SC properties mentioned above, along with an obligation that the tokens exchanged on its I/O channels adhere to a functional relation with the signals crossing the corresponding partition in the original design. While other formalisms may exist for demonstrating the theoretical soundness of FPGA simulation tools, Golden Gate uses the LI-BDN abstraction to enable flexible optimization and push-button model checking.

3.2 LI-BDNs in Golden Gate

While Vijayaraghavan et al. [25] 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 structured as LI-BDN networks, and additionally, lime is the first tool to formally verify primitive LI-BDNs.

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 [13], an open-source tool that enables system designers to simulate their target RTL designs on commodity FPGAs hosted in Amazon's AWS public cloud. FireSim already provides a baseline compiler, MIDAS [15], that relies on decoupling to support co-simulation of models of network and DRAM interfaces. However, it does not actually apply any optimizations to the transformed target design, so the resource utlization 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 Inputs and Outputs

Like FireSim, our compiler consumes an RTL description of the target specified in FIRRTL [12], an RTL intermediate representation convenient for expressing compiler transformations. As output, Golden Gate produces a FIRRTL implementation of LI-BDN corresponding to the target. To complete the simulator, this network is composed with existing FireSim I/O models that produce input tokens and consume output tokens; this yields a closed system.

4.2 Compiler Organization

The Golden Gate compiler is divided into two main phases: *Target Transformation* and *Simulator Synthesis*, as shown in Figure 2.

4.2.1 Target Transformation. Here, optimization candidates are identified and the target's module hierarchy is mutated into a structure that corresponds directly with the final LI-BDN. Target transformations are performed as a series of small operations that preserve the target's RTL timing at every step. This makes it possible to do logic-equivalence checking on the input and output of each pass. Each optimization has its own analysis pass, which inspects the circuit and consumes designer-provided hints captured with FIRRTL annotations. When the pass finds a candidate subcircuit, 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.

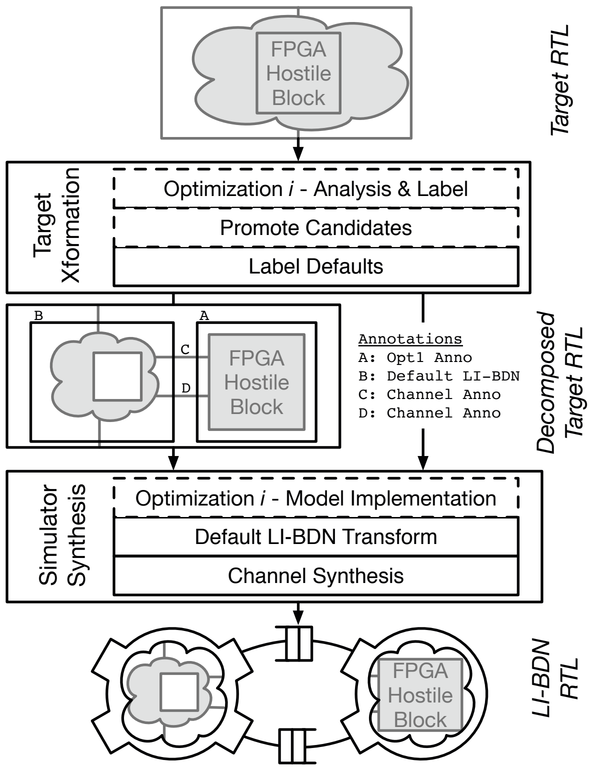

Figure 2: Golden Gate Compiler

4.2.2 Decomposed Target Form. At the end of target transformation, the RTL is in *decomposed target form*, in which every top-level module corresponds with a model in the eventual LI-BDN. All modules are labeled with annotations that indicate how they should be transformed or optimized, and how their inputs and outputs should be coalesced into token channels. Again, here the RTL is functionally identical to the source RTL.

4.2.3 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 (Section 5). Model-implementation passes come in two varieties:

(1) Transformation-based: these modify the corresponding target RTL module to produce a primitive LI-BDN.
(2) Generator-based: these inspect the structure of the target RTL to parameterize an LI-BDN model generator.

4.3 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 $\texttt{fired}_o \vee \texttt{firing}_o$.

4.4 Adding New Optimizations

Adding a new optimization consists of adding an analysis pass to the target transformation, to identify, wrap and label a candidate subcircuit, and adding a model-implementation pass to replace this block with an LI-BDN model in simulator synthesis. To ease integration of such passes, including the multi-ported RAM optimizer in Section 6, we next introduce LIME, a push-button model checker for simulation LI-BDNs.

5 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 optimized models. However, these optimizations are only practical if it is possible to establish that the optimized simulator maintains cycle- and bit-exact correspondence with the original target design. Fortunately, the LI-BDN formalism provides a framework to do this, if it can be shown that each model satisfies formal equivalence and deadlock-avoidance guarantees. However, this is nontrivial, as the the LI-BDN 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 tool for checking the correctness of Golden Gate simulation models.

5.1 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 properies for a given model, with each BMC case structured as an input to the UCLID5 [22] verification system. The LIME flow is depicted in Figure 3, 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.

5.1.1 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 [12], 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.

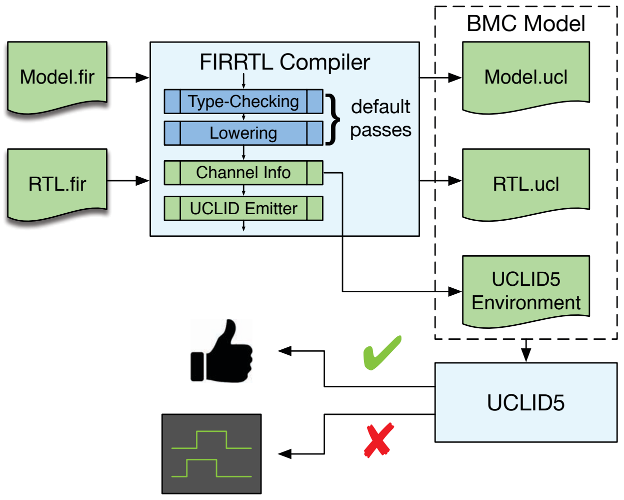

Figure 3: LIME Flow

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.

5.1.2 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.

5.2 Model Checking LI-BDNs

In all LIME property checking flows, the general structure of the model resembles the diagram shown in Figure 4. 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.

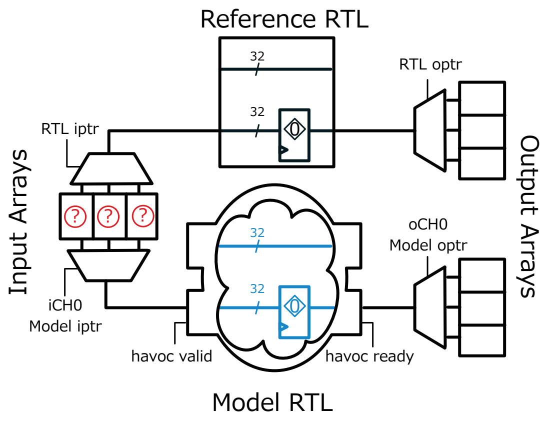

Figure 4: Partial Implementation Model

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 nondeterministically 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.

5.2.1 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. [25] 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 model check must assert the conjunction of the PI invariant PI_j for every $o_j \in O$.

Invariant PI_j

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

5.2.2 *No Extraneous Dependencies.*

Vijayaraghavan et al. [25] 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.

To express this property, we represent it as the conjunction of multiple LTL [21] 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

$$\mathrm{CC}_j(i) := \text{ output } o_j \text{ depends combinationally on input } i$$

$$\mathrm{obligated}_j := \min_{\{i \in I : \mathrm{CC}_j(i)\}} \mathrm{enq\_cnt}_i > \min_{\{o \in O\}} \mathrm{enq\_cnt}_o$$
$$\bigwedge \min_{\{i \in I\}} \mathrm{enq\_cnt}_i \geq \min_{\{o \in O\}} \mathrm{enq\_cnt}_o$$
$$\bigwedge \mathrm{enq\_cnt}_{o_j} = \min_{\{o \in O\}} \mathrm{enq\_cnt}_o$$

$$\mathrm{NED}_j := \mathbf{G}\left(\mathrm{obligated}_j \Rightarrow \mathbf{F}\left(\mathrm{out\_ready}_j\ \mathbf{R}\ \mathrm{out\_valid}_j\right)\right)$$

5.2.3 *Self-Cleaning.*

Vijayaraghavan et al. [25] define SC:

> A primitive BDN has the SC property, if when all the outputs are enqueued n times, all the input FIFOs must [eventually][2] 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 $\mathrm{obligated}_i$ indicating that the simulation LI-BDN has dequeued fewer than n tokens from that channel.

[2]Clarified in [24].

SC LTL property for input i

$$\mathrm{obligated}_i \Leftrightarrow \text{ input channel } i \text{ has a dequeue obligation}$$
$$\mathrm{obligated}_i := \mathrm{deq\_cnt}_i < \min_{\{o \in O\}} \mathrm{enq\_cnt}_o$$
$$\mathrm{SC}_i\ : \mathbf{G}\left(\mathrm{obligated}_i \Rightarrow \mathbf{F}\left(\mathrm{in\_valid}_i\ \mathbf{R}\ \mathrm{in\_ready}_i\right)\right)$$

6 CASE STUDY: MULTI-PORTED MEMORIES

ASIC multi-ported RAMs are a classic culprit of poor resource utilization in FPGA prototypes, as they cannot be trivially implemented in BRAM and are instead decomposed into LUTs and registers [27]. While using double-pumping, BRAM duplication, or FPGA-optimized microarchitectures [17] 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.

6.1 Our Target ASIC

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 [2]. This generator can emit a broad space of systems based on the RISC-V ISA, each consisting of multiple cores, coherent cache hierarchies, peripherals, and outer memory system interfaces. Here, we study two different "core complexes"—consisting of the cores and inner caches—each based on a different RISC-V core implementation: Rocket [2], an in-order scalar core, and BOOM [6], a unified physical register file, superscalar out-of-order core. In each case, we evaluate the impact of substituting each core's floating point (FP) and integer register files for an optimized memory model. Since these cores can be generated with a space of different register files configurations, we describe register file parameters for the instances we study in Table 1.

Type	Size	Read Ports	Write Ports	BMC Runtime
Rocket integer	31×64b	2 comb.	1	445 s
Rocket FP	32×64b	3 comb.	2	334 s
BOOM integer	100×64b	6 comb.	3	637 s
BOOM FP	64×64b	3 comb.	2	372 s

Table 1: Register file specifications for the two target cores.

6.2 Model Microarchitecture

The optimized memory model is structured around a dual-port, synchronous read memory that stores the contents of the simulated memory, which, unlike the FPGA-hostile memory it models, can be implemented in BRAM. 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 6, 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.

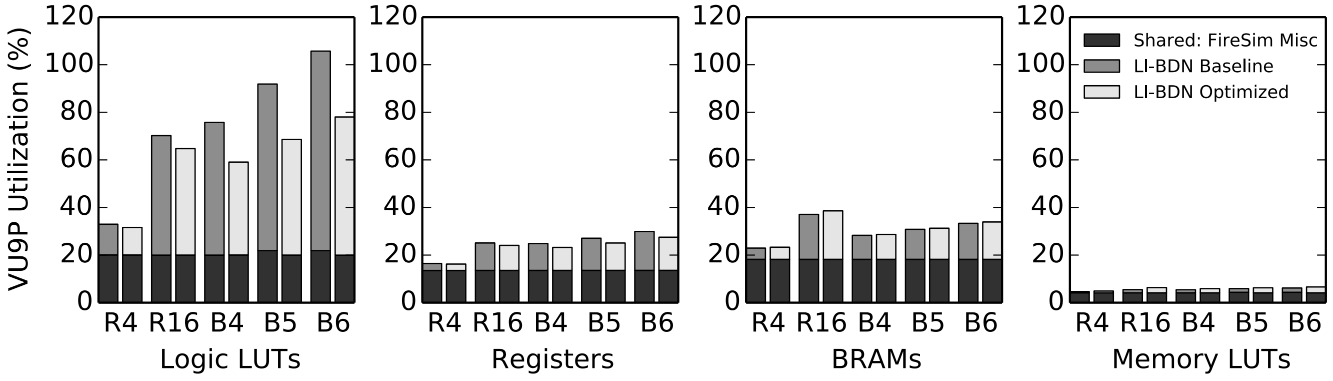

Figure 5: Total VU9P resource utilization for baseline and optimized simulators. Designs are labeled R[ocket]N or B[oom]N, with N corresponding to core count. *FireSim Misc* accounts for all resources in the Amazon-provided shell (v1.4.0) and FireSim hardware for co-simulation; this is fixed across all designs. We omit DSP48s and URAMs as they are constant across all designs and lightly used. *Baseline* penta- and hexa-core BOOM designs failed in placement due to over-utilization– we report post-synthesis utilization. *Optimized* versions of the same designs use 26% fewer logic LUTs, and successfully place and route.

6.3 Verifying the Model With LIME

The LIME flow can be applied to any GOLDEN GATE simulation model, but it is especially useful for the widely applicable memory optimization. While each instance of the model is the output of a parameterized generator, checking a large subspace of the optimized models using LIME provides a high degree of confidence in the correctness of the transformation. In multi-core SoC, checks are also 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. Implementation bugs may appear only in specific corner cases, such as a certain interleaving of I/O token arrivals interacting pathologically with the write collision semantics of the memory. In contrast with labor-intensive, model-specific directed random testing, 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 1 shows that this bound results in quick runtimes; longer BMC checks can be amortized over many uses of common configurations.

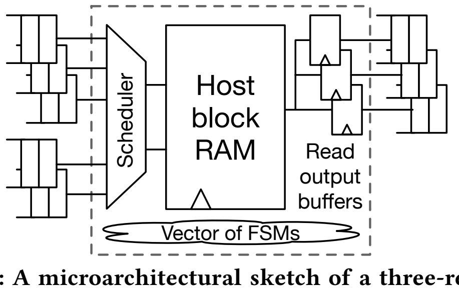

Figure 6: A microarchitectural sketch of a three-read, two-write, optimized GOLDEN GATE memory model.

	R4	R16	B4	B5	B6
Baseline	135 MHz	60 MHz	90 MHz	N/A	N/A
Optimized	135 MHz	60 MHz	80 MHz	60 MHz	50 MHz

Table 2: f_{FPGA} for successfully implemented simulators.

6.4 Adding the Optimization To GOLDEN GATE

To enable our optimization in GOLDEN GATE, we added an analysis pass that finds annotated RAMs and a model-implementation pass that inspects the parameters of the target RAM (ports, width, and depth) and invokes our model generator (Section 6.2). We also annotated the register file RAMs in the target 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 4.2). 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.

6.5 Results

The results of applying the multi-ported memory optimization are shown in Figure 5. Optimized BOOM designs use 26% fewer LUTs than baseline designs, allowing up to six BOOM cores (the B5 and B6 configurations) to be simulated with a single VU9P FPGA, a significant increase over the baseline maximum of four cores. The sixteen-core, Rocket-based design also saw an appreciable 7.8% reduction in LUT utilization over the baseline, despite having lesser-ported, shallower regfiles. We report FPGA frequencies for all designs that closed timing in Table 2.

As discussed in Section 2.3, replacing components of the target with decoupled models will impact the FMR, therefore lowering simulation throughput. FireSim, being a decoupled simulator, generally has FMR greater than unity. In particular, FireSim uses a last-level-cache and DRAM model [4] (utilization included in *FireSim Misc.* of Figure 5) to implement deterministic simulation of the target's outer memory system using host FPGA DRAM. When booting a Buildroot Linux distribution, adding multi-cycle RAM models increases FMR from 1.9 and 1.8 to approximately 6.9 and 9.3 for Rocket and BOOM designs, respectively. FMR is a function of the highest port-count model in each target, but is nearly constant across core count, since the models are not combinationally coupled and therefore execute concurrently. Finally, we note that the performance penalty of using multi-cycle models may often be less than that of partitioning—while using only one FPGA.

7 FUTURE WORK

Golden Gate is the first step toward a general optimizing compiler for FPGA-accelerated simulators. Future research aims include:

(1) *Other Resource Optimizations.* CAMs are another FPGA-hostile structure [27] that can be replaced with multi-cycle models. We suspect large improvements lie in multi-threading [19, 23], where multiple instances of a module are multiplexed over a single physical instance.

(2) *Exploring Resource-Performance Tradeoffs.* Users may wish to use more hardware resources if doing so would improve simulation performance. Future versions of Golden Gate could strike different Pareto-optimal points along the resource-performance curve and automate selection of optimizations.

(3) *Proving Model Correctness.* We plan to extend lime to use k-induction for unbounded checks of LI-BDN properties.

8 CONCLUSION

In this paper, we present Golden Gate, an open-source compiler that automatically decomposes ASIC RTL into a graph of communicating, latency-insensitive simulation models. Golden Gate can identify costly structures, pull them into separate models, and replace them with much smaller, multi-cycle implementations. Using lime, multi-cycle models can be shown to simulate the RTL timing of their ASIC source and to avoid deadlock when composed with the rest of the simulator. Finally, to demonstrate the applicability of this approach, we develop a multi-ported RAM optimization and apply it to a multi-core SoC, enabling the use of a single FPGA where a partitioned prototype would otherwise have been necessary.

9 ACKNOWLEDGEMENTS

The 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. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. We also wish to thank Pramod Subramanyan for his valuable insights on LIME and Murali Vijayaraghavan for his correspondence on LI-BDNs.

REFERENCES

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

[2] Krste Asanović et al. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

[3] Jonathan Babb et al. 2006. Logic Emulation with Virtual Wires. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 16, 6 (Nov. 2006), 609–626. https://doi.org/10.1109/43.640619

[4] David Biancolin et al. 2019. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19).* ACM, New York, NY, USA, 10.

[5] Luca Carloni, Kenneth McMillan, and Alberto Sangiovanni-Vincentelli. 2006. Theory of Latency-insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 9 (Nov. 2006), 1059–1076. https://doi.org/10.1109/43.945302

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

[7] Katherine Compton and Scott Hauck. 2002. Reconfigurable Computing: A Survey of Systems and Software. *ACM Comput. Surv.* 34, 2 (June 2002), 171–210. https://doi.org/10.1145/508352.508353

[8] Rehan Hameed et al. 2010. Understanding Sources of Inefficiency in General-purpose Chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10).* ACM, New York, NY, USA, 37–47. https://doi.org/10.1145/1815961.1815968

[9] Scott Alan Hauck. 1995. *Multi-FPGA Systems.* Ph.D. Dissertation. Seattle, WA, USA. UMI Order No. GAX96-16615.

[10] William N.N. Hung and Richard Sun. 2018. Challenges in Large FPGA-based Logic Emulation Systems. In *Proceedings of the 2018 International Symposium on Physical Design (ISPD '18).* ACM, New York, NY, USA, 26–33. https://doi.org/10.1145/3177540.3177542

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

[12] Adam Izraelevitz et al. 2017. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design (ICCAD '17).* IEEE Press, Piscataway, NJ, USA, 209–216.

[13] Sagar Karandikar et al. 2018. Firesim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18).* IEEE Press, Piscataway, NJ, USA, 29–42. https://doi.org/10.1109/ISCA.2018.00014

[14] Mohammed A. S. Khalid and Jonathan Rose. 2000. A Novel and Efficient Routing Architecture for multi-FPGA Systems. *IEEE Trans. Very Large Scale Integr. Syst.* 8, 1 (Feb. 2000), 30–39. https://doi.org/10.1109/92.820759

[15] Donggyu Kim et al. 2016. Strober: Fast and Accurate Sample-based Energy Simulation for Arbitrary RTL. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16).* IEEE Press, Piscataway, NJ, USA, 128–139. https://doi.org/10.1109/ISCA.2016.21

[16] Helena Krupnova and Gabriele Saucier. 2000. FPGA-Based Emulation: Industrial and Custom Prototyping Solutions. In *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, Reiner W. Hartenstein and Herbert Grünbacher (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 68–77.

[17] Charles Eric Laforest, Ming G. Liu, Emma Rae Rapati, and J. Gregory Steffan. 2012. Multi-ported Memories for FPGAs via XOR. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '12).* ACM, New York, NY, USA, 209–218. https://doi.org/10.1145/2145694.2145730

[18] Edward Lee and David Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[19] Michael Pellauer et al. 2011. HAsim: FPGA-based High-detail Multicore Simulation Using Time-division Multiplexing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11).* IEEE Computer Society, Washington, DC, USA, 406–417. http://dl.acm.org/citation.cfm?id=2014698.2014876

[20] Michael Pellauer, Muralidaran Vijayaraghavan, Michael Adler, Arvind, and Joel Emer. 2009. A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 2, 3, Article 16 (Sept. 2009), 26 pages. https://doi.org/10.1145/1575774.1575775

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

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

[23] Zhangxi Tan et al. 2010. RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors. In *Proceedings of the 47th Design Automation Conference (DAC '10).* ACM, New York, NY, USA, 463–468. https://doi.org/10.1145/1837274.1837390

[24] Muralidaran Vijayaraghavan. 2009. *Theory of composable latency-insensitive refinements.* Master's thesis. Massachusetts Institute of Technology.

[25] Muralidaran Vijayaraghavan and Arvind. 2009. Bounded Dataflow Networks and Latency-insensitive Circuits. In *Proceedings of the 7th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'09).* IEEE Press, Piscataway, NJ, USA, 171–180. http://dl.acm.org/citation.cfm?id=1715759.1715781

[26] John Wawrzynek et al. 2007. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro* 27, 2 (2007), 46–57.

[27] Henry Wong, Vaughn Betz, and Jonathan Rose. 2014. Quantifying the Gap Between FPGA and Custom CMOS to Aid Microarchitectural Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 10 (Oct 2014), 2067–2080. https://doi.org/10.1109/TVLSI.2013.2284281