FASED: FPGA-Accelerated Simulation and Evaluation of DRAM

David Biancolin[1], Sagar Karandikar[1], Donggyu Kim[1], Jack Koenig[1], Andrew Waterman[2], Jonathan Bachrach[1], Krste Asanović[1,2]
[1]ADEPT Lab, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, USA
[2]SiFive Inc., San Mateo, California, USA
{biancolin,sagark,dgkim,jack.koenig3,waterman,jrb,krste}@eecs.berkeley.edu

ABSTRACT

Recent work in FPGA-accelerated simulation of ASICs has shown that much of a simulator can be automatically generated from ASIC RTL. Alas, these works rely on simple models of the outer cache hierarchy and DRAM, as mapping ASIC RTL for these components into an FPGA fabric is too complex and resource intensive. To improve FPGA simulation model accuracy, we present FASED, a parameterized generator of composable, high-fidelity, FPGA-hosted last-level-cache and DRAM models. FASED instances are highly performant, yet they maintain timing faithfulness independently of the behavior of the host-FPGA memory system. For a given scheduling policy, a single FASED instance can model nearly the entire space of realizable single-channel DDR3 memory organizations, without resynthesizing the simulator RTL. We demonstrate FASED by integrating it into a flow that automatically transforms RTL for multicore RISC-V processors into full-system simulators that execute at up to 150 target MHz on cloud-hosted FPGAs.

CCS CONCEPTS

• **Hardware → Simulation and emulation**; *Dynamic memory*; Reconfigurable logic applications.

KEYWORDS

emulation; FPGA prototyping; memory systems

ACM Reference Format:
David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, Krste Asanović. 2019. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), Feb. 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293894

1 INTRODUCTION

With the slowdown in process technology improvements, architects are increasingly turning to specialization to deliver advances in performance and energy efficiency. Modern SoCs contain a collage of fixed-function units and specialized accelerators, with general-purpose application processors consuming a dwindling fraction of the die. Heterogeneous specialized systems add new complexity at all levels of the computing stack, and research into new programming models, runtimes, and operating systems is expanding.

Architects and systems designers will need a comprehensive set of simulation technologies to enable this research. Both architectural and microarchitectural full-system software simulators will remain important sandboxes for prototyping new ideas. In many domains, sampling techniques permit the use of slower but more detailed microarchitectural simulators, providing greater fidelity without loss of simulation throughput. Unfortunately, there are many cases in which existing software-based microarchitectural simulators are too slow, and sampling techniques fail because samples cannot be reused for changes that have large impacts on execution behavior. A few such cases include tuning highly parallel specialized multiprocessors; runtimes that dynamically schedule and optimize code based on performance; and hardware-software co-design flows, where the hardware and software change simultaneously. In such cases, FPGAs are the only technology that can provide high-fidelity full-system simulation with low experimental latency, high throughput, and low cost per simulation cycle.

FPGA-accelerated simulation has been actively studied over the past decade, notably in the multi-university RAMP project [23], but it has seen little adoption for a number of reasons:

(1) FPGA-accelerated simulators are difficult to write or modify, and lengthy compilation times make them onerous to debug.
(2) FPGAs have historically been resource-constrained, limiting the scale of the system under simulation or incurring the great additional complexity of multi-FPGA partitioning.
(3) Purchasing and maintaining an FPGA cluster is prohibitively expensive.

Fortunately, recent technological advances address the latter two challenges: FPGAs have been scaling well, providing greater f_{max} and capacity, and are now widely available as cloud-hosted resources [1]. Alas, design complexity challenges remain.

One promising avenue is to automatically generate the FPGA-hosted components of the simulator from RTL produced by highly configurable generators such as the Rocket Chip RISC-V SoC generator [2]. The biggest limitation of this approach so far has been modeling the main memory system. The DRAM controller RTL, physical interface, and chip models cannot be simply mapped to the FPGA, so prior work used simplistic, handwritten RTL models (e.g., latency pipes) backed by FPGA-attached DRAM [11]. In this paper, we address the challenge of flexibly modeling DRAM memory-systems at greater fidelity. The techniques we propose can be applied to modeling other memory types, such as non-volatile memories, and I/O devices where transformation from ASIC RTL is difficult or impossible. This paper makes the following contributions:

First, we propose separating the concerns of host-platform mapping from target modeling, by writing the timing model of a split-timing-functional model as *target-time* RTL. This approach makes it considerably easier to describe detailed timing-model *generators* and allows new users to add new timing models without a detailed understanding of how the model will be mapped to the FPGA.

Second, we demonstrate the flexibility of this approach by presenting FASED, a last-level-cache and multi-rank DDR3 timing-model generator with fidelity comparable to cycle-accurate software-based simulators like DRAMSim2 [19]. FASED instances can be reconfigured without FPGA recompilation and are instrumented to provide the same performance and power measurements as software-based simulators.

2 ON FPGA-BASED SIMULATION

We first review the use of FPGAs for architecture studies. Throughout this paper, we make a distinction between the *target* and the *host*. The target is the design under study. Combining the target with a model of the environment in which it executes forms a determinate closed system whose behavior is defined independently of the simulation host. The host is the hardware that executes (*hosts*) the simulation. In this paper, a host consists of one or more CPUs connected to one or more FPGAs.

2.1 FPGA Prototyping

FPGAs have long been used to *prototype* ASICs by implementing the ASIC RTL directly in FPGA logic. While FPGA prototypes are both fast (10s to 100s of MHz) and detailed, they require a complete RTL description of the target design. Furthermore, larger designs must be painstakingly partitioned across multiple FPGAs. Since these multi-FPGA prototypes advance in lockstep, cycle by cycle, they are considerably slower (100s of KHz to 1s of MHz). Nonetheless, FPGAs are used widely in industry, as they allow software development and hardware validation to proceed months before silicon is available.

2.2 FPGA-Accelerated Simulation

Prior work has explored techniques to make FPGAs more usable and powerful simulation hosts. Motivated by the dawn of the multicore era, the multi-university RAMP project [23] made large strides in improving FPGA-accelerated simulators by improving resource efficiency, developing FPGA partitioning techniques, and avoiding FPGA recompilation by using reconfigurable models.

ProtoFlex [6] was an architecture-level simulator that demonstrated 16-way host-multithreading of a single FPGA-hosted functional model. ProtoFlex could switch between FPGA-hosted and CPU-hosted modes via *transplantation.* FAST [5], a cycle-accurate simulator, was split into CPU-hosted functional and FPGA-hosted timing models. RAMP Gold [20] used FPGA-hosted timing and functional models with 64-way host-multithreading to model a larger target on a single FPGA. HAsim [17] also used FPGA-hosted timing and functional models, but provided more detailed pipeline and memory hierarchy models.

Other work studied partitioning targets over multiple FPGAs. [8] showed that by partitioning HAsim over two FPGAs, they could model eight times as many cores, due to improved resource sharing between virtual instances. To model a datacenter-scale target, DIABLO [21] leveraged RAMP Gold's multithreading to simulate 3072 servers on 24 FPGAs.

A unifying theme of FPGA-accelerated simulators is that one clock-cycle of target time is executed over a variable number of FPGA-host cycles. This lets an FPGA-hosted simulator hide variable host latencies to DRAM and CPU-hosted components, enables optimizations that trade host time for host resources, and, crucially, facilitates deterministic simulation. This *host-target decoupling* is what differentiates an FPGA-accelerated simulator from an FPGA prototype. We expand on this property in Section 3.2.1.

2.3 Adoption Challenges

Despite their promise, FPGA-accelerated simulators have only been successfully employed by those who designed them. We attribute their limited appeal to several factors:

1. **Availability.** Much of the early FPGA-accelerated simulator research relied on boutique FPGA-emulation platforms or custom board designs, whose high cost and limited availability prevented adoption.
2. **FPGA Capacity.** Common ASIC structures, such as CAMs and multi-ported RAMs, map poorly to FPGA fabrics [24], making it difficult to host large ASIC designs on FPGAs.
3. **Ease of Use.** To avoid partitioning across multiple FPGAs, previous work focused on efficiently mapping more of the target to a single FPGA. The abstract, multithreaded models these simulators typically employ can be more difficult to implement than the machines they model, greatly undermining their usability. This complexity limits configurability, forcing users to modify a sophisticated piece of RTL to make larger changes. Furthermore, these abstract models must, like their software counterparts, be validated and calibrated, making them even more laborious to use.

2.4 Recent Technological Advances

Even as Moore's law wanes, FPGA capacity continues to scale. The largest FPGAs have over 50 MiB of BRAM and millions of logic cells. As they have scaled, FPGAs have become more heterogeneous, adding features that make them better hosts for full-system simulators. Both Intel and Xilinx sell FPGAs with embedded microprocessors, making it easier to co-simulate tightly coupled hardware and software models. Modern FPGAs include dedicated DRAM controllers that support memory bandwidths rivaling those of ASICs.

Lower cost and increased on-chip integration have also made FPGAs more accessible to researchers. Not only are commercial off-the-shelf development boards cheaper and more full-featured, FPGAs are now available as a cloud service [1]. Where in the past academics would have to purchase their own FPGAs to reproduce published experiments, instead, it is now possible to spin up identical simulations on FPGAs in the cloud. This development promises to foster more collaboration around FPGA-accelerated simulation.

2.5 Usability Through Automation

While the trends described in the previous section solve the *availability* and *FPGA capacity* challenges, usability remains a problem. Previous work [7, 12] has shown that much of an FPGA-accelerated simulator can be automatically generated from source RTL. This

RTL can be written in an HDL like Verilog, generated by a high-level synthesis tool, or emitted by languages like Chisel [3] or Bluespec.

Alas, it is not always practical to generate models from source RTL. Consider off-chip memory systems: they are too resource-intensive to host in the FPGA fabric, yet for reasonable simulation performance, they must be tightly coupled with the processor model. Components like these require an abstract model to virtualize the target memory system over DRAM attached to the FPGA—reintroducing the problem that anything but a simplistic model is difficult to design, validate, modify, and reuse.

To avoid these pitfalls, we propose writing detailed memory-system timing models as decoupled, split-timing-and-functional models with the timing models written as *target-time* RTL. Using this approach, the same RTL transformations applied automatically to the processor RTL are applied to the timing-model RTL before binding it to the functional model. Model designers can focus on modeling detailed target behavior and not worry about the mapping to the host. With our approach, since timing models are transformed from target-time RTL, it is possible to use HLS-generated RTL or even existing memory-controller RTL as a timing model.

To improve reusability, we propose writing timing models as *generators*. This allows the model designer to describe a space of *instances* with less development effort. To support reconfiguring timing models without FPGA recompilation, timing models expose timing parameters as I/Os that are bound automatically to memory-mapped registers during timing-model generation. Taken together, these techniques make it possible to describe detailed, reusable memory-system models. We demonstrate this claim with FASED.

3 THE SIMULATION FRAMEWORK

While FASED can compose with other FPGA-accelerated simulators, in this paper we extend FireSim [10] and MIDAS [13]. MIDAS is a compiler that generates FPGA-accelerated simulators automatically from Chisel RTL. MIDAS is not standalone; FireSim provides a development environment, with RTL and software models for complete target designs, as well and powerful utilities to batch out FPGA builds and simulations across Amazon EC2.

3.1 Host-Target Decoupling

Generally, host-target decoupling begins with a *target abstraction* that represents the target and its environment as a dataflow graph of actors [16, 22]. The target abstraction we use in this paper derives from the one used in RAMP [23] and resembles a synchronous-dataflow graph [15] where:

- *Tokens* are messages passed between the nodes of the graph. Tokens represent the values on wires in the target at the end of a target cycle.
- *Models* are graph nodes that model the behavior of a synchronous block of RTL. Each model executes one target cycle of simulation by dequeuing a token from each of its inputs and enqueuing a token into each of its outputs.
- *Channels* are the edges of graph. They transport tokens between models and simulate target-interconnect latency, buffering, and clock-domain crossings. At the start of simulation, a channel is initialized with a number of tokens equal to its latency.

A simulator that faithfully implements this graph decouples target time from host time. Unlike in an FPGA prototype, where every FPGA clock cycle emulates a target clock cycle, an FPGA-hosted model only executes a target-clock-cycle when it can legally fire. Thus, the behavior of the target is decoupled from the host, allowing simulators to be partitioned across the host and to tolerate variable-host-latencies to DRAM and the CPU, while remaining deterministic. Unfortunately, this results in target time advancing slower than it would in an FPGA prototype of the same host frequency. This is quantified by the FPGA-cycle-to-Model-cycle Ratio (FMR)[16], below, which increases from one (the simulator simulates a target cycle on every FPGA host cycle) as the simulator stalls on token availability and backpressure. The FMR of a simulator is variable: it is a function of both application-dependent behavior in the target and variable latencies in host services.

$$FMR = \frac{Cycles_{FPGA}}{Cycles_{Target}}$$

3.2 The MIDAS Compilation Flow

MIDAS-generated simulators compose three types of models in their target graphs:

(1) *Transformed RTL* models generated from ASIC RTL.
(2) *Abstract RTL* models intended for FPGA hosts.
(3) *Software* models that are hosted on a CPU.

The next three sections give an overview of how MIDAS, with the help of the user, maps a target graph composed of these models into an FPGA-accelerated simulator.

3.2.1 *ASIC-RTL-to-Model Transformation.*

We transform a synchronous block of RTL into a model using a FIRRTL [9] transformation called a *FAME-1* transform. The transformation gates state update of the RTL with a model-global signal, *targetFire*, which is driven with the AND-reduction of the valid signals of all input ports and the ready signals of all output ports. Thus, in these models, state update, output token enqueue, and input token dequeue occur simultaneously in a single host-clock-cycle.

3.2.2 *FPGA-Host Mapping.*

Once ASIC-RTL has been transformed into models, MIDAS creates a host-agnostic mapping of the target graph. This is also the point where MIDAS links in other models, including FASED instances.

Using Chisel, MIDAS generates simulation FIFOs that implement the channels of the target graph. When a channel spans the boundary of the FPGA, MIDAS generates an *endpoint*, a FIFO with a matching head or tail on the opposite part of the host platform. Together, these endpoints implement the simulation channel. All remaining I/O on transformed-RTL models are bound to a default I/O model, which acts as an infinite source and sink of tokens. During this process, MIDAS also generates memory-mapped modules for simulation control and instrumentation. These include a DRAM-initialization module and a master that governs the advance of target time on the FPGA.

Once all of the simulation components have been generated, the simulation interconnect is elaborated and bound to a single AXI4 slave port. All memory-mapped simulation components are accessed through this interface. Additionally, a crossbar is generated to arbitrate between components that require FPGA-host

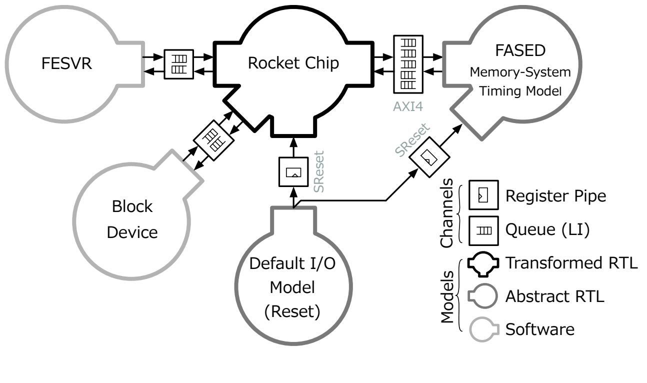


Figure 1: The graph of target designs studied in this paper.

DRAM. Ultimately, MIDAS emits a Verilog file and a C++ header describing the simulator's memory map. To generate a bitstream, the user instantiates the MIDAS-generated Verilog in a skeleton FPGA project that exposes AXI4 interfaces to the FPGA's off-chip memory systems and interconnect to the host CPU.

3.2.3 Software Simulation Driver & Software Models. To control the simulator, the user writes a C++ program that links against the MIDAS C++ libraries and the generated header. The MIDAS libraries implement basic commands used to control simulation. These commands are decomposed into memory-mapped I/O issued over the simulation interconnect. To complete the simulator, the user links in software models into this program.

3.3 Targets & Hosts of This Paper

All target designs used in this work are tethered RISC-V processors with a single-channel DRAM subsystem. They share the target graph shown in Figure 1. This graph comprises a Rocket-Chip-generated transformed-RTL model that includes one to four Rocket pipelines with L1 caches and a cache-coherence controller; software models for a UART (not shown), a block device, and the RISC-V front-end server (FESVR, which provides BIOS-like functionality); and a FASED instance, which connects to an AXI4 port presented by the processor. All remaining I/O of the transformed-RTL model, including reset, is bound to the default I/O model.

While MIDAS supports other FPGA hosts, currently FireSim only has support for Amazon EC2 F1 instances. F1's `f1.2xlarge` instances have a Xilinx UltraScale+ XCVU9P[1] attached to four 16 GiB channels of ECC-enabled DDR4 SDRAM. FPGAs are attached to a CPU with 8 hardware threads and 112 GiB of DRAM. Figure 2 shows the target mapped to an F1 host.

4 MEMORY MODEL ARCHITECTURE

FASED is a *generator*, written in Chisel [3], that can elaborate *instances* from a space of possible parameterizations. Instances themselves model a space of different memory systems: the user picks the final target-design point by programming the instance's configuration registers at runtime.

As input, FASED accepts a parameterization that constrains features of the instance, such as its interface widths, the maximum number of outstanding requests it must support, and the type of

[1] 2.6 million logic cells, 346 Mb of on-chip memory.

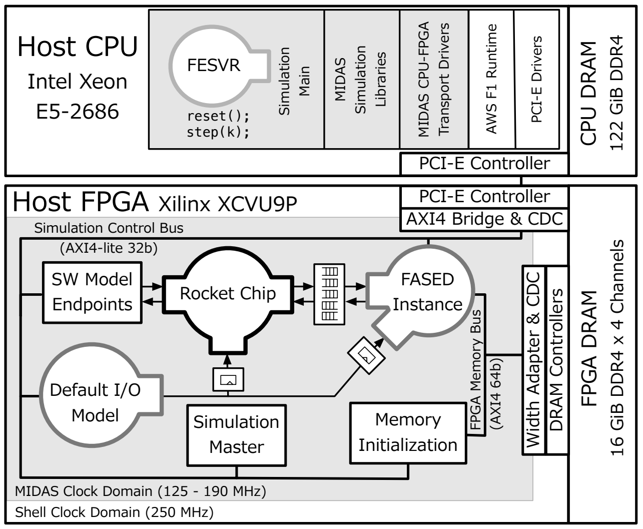


Figure 2: The target mapped to an F1 host. The contribution of this work is the FASED instance, which replaces a crude latency-pipe model provided by the prior work.

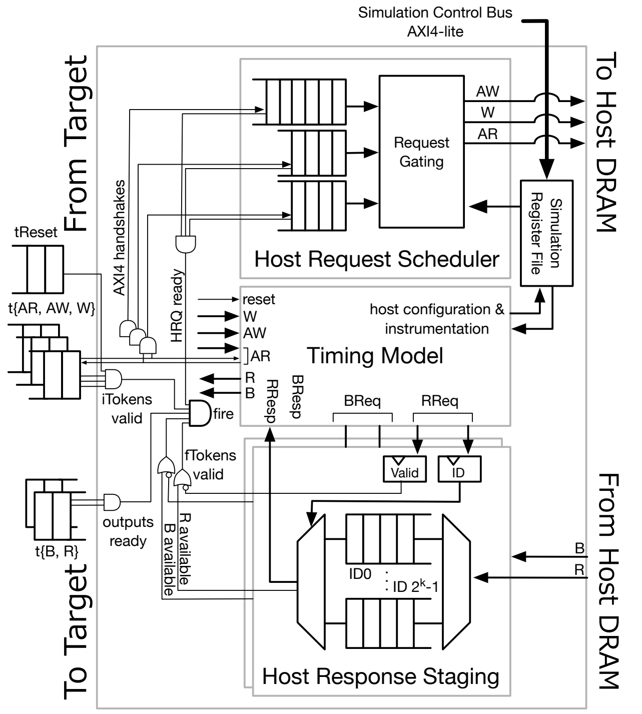


Figure 3: A block diagram of a FASED instance, with all signals that may stall timing model execution illustrated.

memory system the instance will model (a *timing-model class*). As output, FASED generates an instance RTL module and memory map of its host configuration registers. These registers control timing parameters; their values can be modified at runtime to reconfigure the instance without needing to recompile the FPGA bitstream.

4.1 Instance Organization

The block diagram of an instance is shown in Figure 3. Instances operate by using the FPGA host's DRAM as a backing store. Target

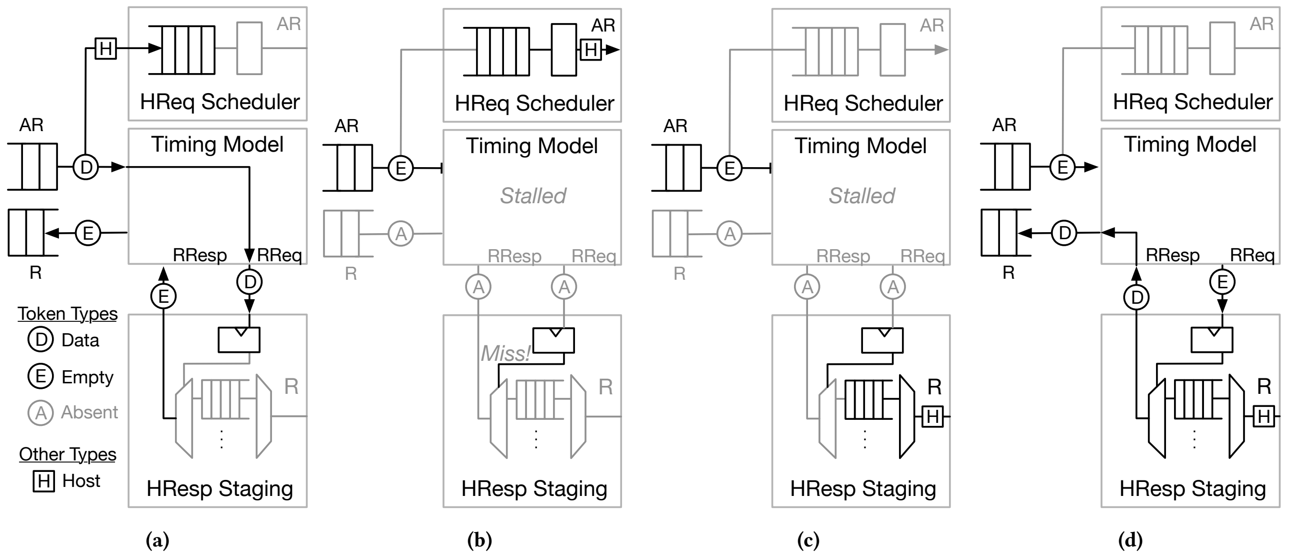


Figure 4: A FASED instance simulating a single-target-cycle read. Data tokens carry target transactions (their target-valid bit is set) whereas empty tokens do not carry a target transaction.

requests carried in simulation tokens are snooped by the *host-request scheduler*, which issues them to the host memory system. Responses from the host memory system are subsequently buffered in the *host-response staging* modules. In parallel, the *timing model*, a simulation model transformed from target-time RTL using a FAME-1 transform, consumes input tokens and generates output tokens. When the timing model wishes to release a valid memory-response token, it queries the host-response staging module for the corresponding host response. If the host memory system has not yet responded, `targetFire` is de-asserted, preventing token flow and ultimately stalling the simulator. We describe this mechanism in greater detail in the next section (4.2).

The host-request scheduler and host-response staging modules, together with the FPGA-host DRAM, constitute the functional model of an instance. Timing models are written in target-time RTL and have three interfaces:

(1) An AXI4 port through which the model receives memory requests from the rest of the target.
(2) Two functional-model request ports (BREQ & RREQ) through which the timing model fetches data for target responses from the host-response staging module. Responses are carried by the next tokens (fTokens) generated by the host-response staging unit.
(3) A host-configuration port that carries the timing parameters of the model and records instrumentation data.

During instance generation, FASED binds the host-configuration port to memory-mapped registers on the simulation bus. It then FAME-1-transforms the timing model, connects it to token queues, and binds the `targetFire` signal, which is asserted when all of the following conditions hold:

(1) All input tokens are present (*iTokens valid* in Figure 3).
(2) All output queues are ready (*outputs ready*).
(3) The host-request scheduler can accept a request (*HRQ ready*).
(4) All host-response tokens are present (*fTokens valid*).

4.2 Operation

To demonstrate how FASED instances operate, let us consider an instance with a single-cycle-memory timing-model. This is depicted in Figures 4a-4d.

Suppose we have reached the first read request issued to the memory system (Figure 4a). Let this be host and target cycle 0. When the timing model accepts this request, it is snooped by host-request scheduler. Simultaneously, the timing model makes a request to host-response staging module as it needs to reply to the read in the next target cycle.

In host cycle 2 (Figure 4b), the host-response staging module cannot produce the associated host response, since it has yet to be issued, and so generates no fToken, stalling the timing model. In parallel, the host-request scheduler issues the required read to the host memory system. While the host-response staging module waits, the timing model stalls. If the host memory system responds in K cycles, at host cycle $K + 1$ (Figure 4c), that response is received.

In cycle $K + 2$ (Figure 4d), the host-response staging module produces an output token, targetFire is asserted, and target cycle 1 executes. Here the timing model forwards the data directly into its output token. From the target's perspective, the read occurred in a single cycle, however, the cycle was executed with an FMR of $K + 2$. As the latency of the target increases, FMR decreases and approaches unity. If the target memory system is strictly slower the host memory system, the instance executes at unity FMR.

4.3 Functional Model Configuration

Our design allows both the host memory system and the timing model to reorder responses; the host-staging unit implements a set of virtual queues for each AXI4 channel. Each queue represents the FIFO ordering within a single channel ID. The size of the functional model is sensitive to the maximum number of reads and writes it must accept, the maximum number of transactions that can be in flight on the same ID, and the maximum request lengths. For small degrees of ID reuse, or small numbers of outstanding requests, the memory system model implements each virtual queue as a physical queue, and aggregates them together in one dual-ported BRAM [8].

For greater numbers of AXI IDs or greater degrees of ID reuse, it dynamically assigns entries within block RAMs, and maintains a hardware linked list to track to read-response order in a given ID.

4.4 General-Purpose Timing-Model Classes

FASED provides two simple, general-purpose timing-model-classes that can be used to model large off-chip memory systems. The first is a *latency-bandwidth pipe* (LBP) that applies independently programmable latencies to read and write requests and will not accept any new requests beyond a programmable limit. The second is a *bank-conflict* model, which adds a penalty of $max(0, t_{CP} - t_{\Delta})$ cycles to a base latency if the bank was used t_{Δ} cycles prior, where t_{CP} is the maximum conflict penalty. These models were validated in trace-driven RTL simulation against software golden models which match their cycle-by-cycle behavior exactly. We give the FPGA resource utilization for a handful of instances in Table 1[2].

Example Instance	Logic LUTs	FFs	BRAM	f_{MAX}
8 read, 8 write	1337	972	3	281
32 read, 32 write	2119	1500	1	264
Above w/ no ID reuse	1289	873	1	317

Table 1: Resource counts and best-case f_{MAX} (MHz) for three different LBP models (maximum AXI4 burst length of 8 beats). Supporting more concurrent transactions (row 2) requires a larger functional model; this can be mitigated by giving the generator hints (row 3).

4.5 Composable Last-Level-Cache Model

All timing model classes can be generated with a single-banked, write-back, last level cache (LLC) model with a random replacement policy. Since we can reuse the same functional model, the model only instantiates tag and metadata arrays, letting us model an LLC that would be too large to fit on the FPGA. FASED LLC models have a runtime-configurable number of sets and MSHRs, associativity, and line size. Refills from the backing memory model are prioritized over reads over writes. Reads or writes made to a set with a pending writeback or refill are interlocked. We make the cache composable with all other timing-models by implementing an additional internal AXI4 bus (stripped of its data fields). We give the FPGA resource utilization for a handful of LBP-backed LLC instances in Table 2.

Example Instance	Logic LUTs	FFs	BRAM	f_{MAX}
4 MiB, 16 ways	2166	1240	27	222
4 MiB, 8 ways	2265	1272	39	220
4 MiB, direct mapped	1848	1241	34	242
64 MiB, 8 ways	2545	1426	251	152

Table 2: Resource counts and best-case f_{MAX} (MHz) for four different LBP-backed LLC models, labeled with the largest capacity and associativity they can model (128B cache lines).

We validated the LLC model in RTL simulation backed with a latency bandwidth pipe. We generated trace-based microbenchmarks and measured cache behavior for a set of generated instances programmed with runtime settings.

[2]We used Vivado 2017.1, targeting the XCVU9P-FLGB2104-2-i device present on F1 instances. We registered all I/O, and overconstrained the design to 400 MHz to obtain a best-case f_{MAX}. We exclude memory LUTs (lightly used) and DSP48s (unused).

5 ON DRAM MEMORY SYSTEMS

Before describing FASED's DRAM timing-models (Section 6), we review some relevant background on DRAM memory systems.

5.1 DRAM Device Architecture

In a DRAM IC, arrays of bit cells are hierarchically arranged into multiple parallel *banks*. Banks provide the primitive level of concurrency in a DRAM memory system: they can service independent requests assuming they do not simultaneously require shared resources like the data, address and command buses. Multiple DRAM ICs can be arranged in parallel to widen the data bus; address and command buses fan out to each IC.

A basic DRAM operation requires a series of three commands: *activate (ACT)*, *column access (CAS)*, and *precharge (PRE)*. The ACT command enables the word-lines of the array corresponding to a single *row* of the bank. The cells of the row are sensed and saved in a *row buffer*. A CAS command then selects a subset of the row buffer to read or write; data is bursted over successive clock edges. While the row buffer remains *open*, the row can be accessed by issuing new CAS commands. To access a different row, a PRE command must be issued to *close* the row and recharge the bit-lines.

DRAM gradually loses its stored state over time as bit cell capacitors leak. To maintain their state, DRAM cells must be periodically refreshed. In the DDR standards, JEDEC mandates that cells must be refreshed once every 64 ms. Since activations to every row cannot generally be guaranteed during normal use, DRAM devices are refreshed explicitly with a refresh command (REF). To reduce complexity, this command refreshes a constant number of contiguous rows in all banks concurrently. DRAM manufacturers generally have kept the number of refresh commands required to iterate through the entire array constant: 8192 commands per 64 ms interval, or one every 7.8 μs.

5.2 DRAM Controller Architecture

A DRAM controller is responsible for responding to memory requests from one or more requesters by scheduling those requests over its memories as a judicious stream of DRAM commands.

Memory access scheduling (MAS) is the process by which, for a given cycle, a controller selects a single DRAM command to be issued from a legal set. Legal commands are constrained by the current state of each bank, the availability of shared resources like the command and data buses, and timing constraints imposed by the DRAM devices. Good MAS policies strike a balance between minimizing latency, maximizing bandwidth, minimizing power, and maintaining quality-of-service guarantees. In this paper we consider two common MAS policies: First-Come First-Served (FCFS) and First-Ready FCFS (FR-FCFS) [18].

In a FCFS MAS, commands for the oldest pending memory reference are issued first. This is the simplest MAS policy, but tends to under-utilize available DRAM bandwidth as younger requests that may hit an open row buffer must wait behind commands that miss. In a FR-FCFS MAS, first, ready (legally issuable) column commands are prioritized over ready row commands. Second, commands for older references are prioritized over younger ones. This permits younger but ready column commands to be issued before older row commands, improving DRAM bandwidth utilization considerably.

5.3 DRAM Memory System Simulators

The current state of the art in DRAM simulation in academia is cycle-accurate software simulators like [4, 14, 19]. These simulators generate DRAM command streams that have been validated against industrial models. In trace-driven mode, operating at full throughput and only as a timing-model, these models simulate at rates of hundreds of KHz to ones of MHz (reported in [14]).

6 DRAM TIMING MODELS

We provide two DDR3 timing-model classes based around FCFS and FR-FCFS MAS. DDR3 timing models have a runtime-configurable address assignment, speed grade, page policy, and rank, bank, and row count. The timing models generate legal DDR3 command traces that have been validated against Verilog golden models.

6.1 Timing Model Design

Both timing-model classes consist of four components: transaction scheduler, DRAM state trackers, MAS, and backend.

6.1.1 Transaction Scheduler. The transaction scheduler consists of a single, unified, configurable-depth queue that can accept an AXI4 read and write transaction simultaneously. Reads are given priority when only one slot is available. Transactions are passed to the MAS as they can be accepted at a rate of one transaction per cycle.

6.1.2 DRAM State Trackers. We decoupled the MAS model design from structures that track the state of DRAM devices. The DRAM state tracker is arranged hierarchically into rank and bank state trackers. State trackers present to the MAS a bit vector indicating which commands they may legally accept. Trackers have a counter for each command type, which indicate the next earliest cycle that the tracker can legally accept a command of that type. When the MAS issues a command, it informs the associated state tracker, which updates its counters accordingly.

6.1.3 Memory Access Scheduler. Each MAS maintains a data structure of memory references it is scheduling across. When a new transaction is received, a record in this structure is populated with the decoded rank, bank, and row addresses, along with MAS-specific metadata to ease scheduling decisions. On every target cycle, the MAS selects a legal command based on the available references and the bit vectors presented by the state trackers. The MAS releases reads to the backend on the cycle the first beat would return from DRAM. Writes are released the cycle after a CASW command is issued. Both MAS models have a simple, runtime-configurable page policy. The open-page policy keeps pages open until the next refresh or another row is to be opened. The closed-page policy always issues auto-precharged CAS commands. To maintain a global age-order of memory references, the FR-FCFS MAS uses a single collapsing buffer with a runtime-programmable depth.

6.1.4 Refresh Policy. Both MAS use an interrupting, all-ranks refresh scheme that makes no attempt to pull-in or delay refresh commands. When a refresh is requested, the MAS precharges all banks in all ranks, and issues REF commands as soon as possible.

6.1.5 Backend. The backend receives read and write references from the MAS and drives the AXI4 response channels back to the target. If no LLC is being used, read response data is fetched from the host-response staging unit. The backend can apply an additional runtime-configurable latency to simulate additional latency on read responses and write acknowledgments.

We give the FPGA resource utilization for a handful of representative DDR3 instances in Table 3.

Example Instance	LUTs	FFs	BRAM	f_{MAX}
FCFS Single Rank (SR)	2272	1647	1	272
FCFS Quad Rank (QR)	4433	3263	1	219
FR-FCFS, SR, 8 Deep	3172	2188	5	239
FR-FCFS, QR, 16 Deep	8206	4805	1	158

Table 3: Resource counts and best-case f_{MAX} for four different DDR3 models. The last instance can schedule over 16 references and thus has a larger functional model.

6.2 Validation

Since our DRAM-timing models generate DRAM-command traces, we validated traces collected in RTL simulation against a Micron DDR3 golden model. The golden model simulates a single-device slice of the memory organization and detects DRAM timing violations in the command trace. We made small modifications to the provided test bench to support validation of multi-rank organizations and to accept our trace format. This is the same validation approach used by all popular software cycle-accurate simulators. To perform performance validation, we generated memory-reference traces for which we could estimate bandwidth and row buffer hit rates a priori. The traces were tailored to expose different memory-access scheduling behavior in the MAS models.

6.3 Selecting Legal Runtime Configurations

Our most sophisticated DDR3 timing-model instances expose over thirty runtime-programmable settings that specify the low-level DRAM timings, address-assignment scheme, LLC-model configuration, and MAS-structure sizes. Each of these settings has a legal range of values that depends on hardware in the generated instance. To make it easier to program instances, we provide a runtime-configuration utility that assigns these settings from higher-level free parameters. Users specify the DRAM data-bus width, device DQ width, number of ranks, and total DRAM capacity. Our utility then searches for a DDR3 device in a database and chooses one with an appropriate density and speed grade for that memory system.

6.4 Comparison to DRAMSim2

FASED models nearly all of the same aspects of DRAM as DRAMSim2, but it is lacking support for burst chopping and for power-down and self-refresh modes. We plan to add these features in the future. FASED does not natively model multi-channel memory systems, as this can be easily accomplished by using multiple instances. Similarly, FASED does not natively simulate a clock-domain crossing between the controller and the rest of the processor, instead, this is modeled in the MIDAS-generated channels. FASED makes up for these limitations with simulation speed: as part of a full-system simulator FASED executes 100x - 1000x faster than popular cycle-accurate simulators running standalone (measured in [14])! We expand on FASED's performance in Section 8.

	Logic LUTs				Registers				36K BRAM			
Target Design	Simulator		Memory Model		Simulator		Memory Model		Simulator		Memory Model	
SC-FCFS	48 664	5.3%	4680	0.5%	24 608	1.3%	3071	0.2%	30	1.8%	4	0.2%
SC-FRCFS	50 638	5.5%	6095	0.7%	25 019	1.4%	3565	0.2%	30	1.8%	4	0.2%
SC-LLC-FCFS	49 969	5.5%	6158	0.7%	24 989	1.4%	3536	0.2%	70	4.2%	44	2.6%
QC-FRCFS	128 714	14.1%	6014	0.7%	62 480	3.4%	3565	0.2%	100	6.0%	4	0.2%
QC-LLC-FRFCFS	130 329	14.3%	7747	0.8%	62 870	3.4%	3957	0.2%	140	8.3%	44	2.6%

Table 4: XCVU9P resource utilization for a space of different targets. Percentages indicate the share of total FPGA resources consumed by that design partition. Simulator totals are inclusive of the memory model.

Benchmarks	**Insns (T)**		**D$ MPKI**		**I$ MPKI**	
perlbench	2.98	2.99	9.0	8.9	10.0	10.1
gcc	2.43	1.35	36.6	29.5	9.7	11.1
mcf	1.60	0.91	97.9	80.9	0.1	0.1
omnetpp	1.11	1.11	56.9	56.6	9.3	10.4
xalancbmk	1.21	1.21	62.9	62.9	7.9	7.6
x264	4.55	4.55	3.0	3.0	2.9	3.0
deepsjeng	2.51	2.14	8.7	8.2	15.4	15.3
leela	2.59	2.59	5.8	5.8	1.5	1.5
exchange2	3.24	3.24	0.0	0.0	0.1	0.1
xz	9.41	2.25	19.8	15.7	0.2	0.1

Table 5: Dynamic instruction counts and L1 MPKIs of SPEC2017int rate and speed (single threaded), respectively.

7 EXPERIMENTAL SETUP

In Section 3, we described how MIDAS takes a target and maps it to a host. Here, we describe the microarchitecture of the target machines we simulate, give an overview of the SPEC2017 Integer benchmarks that we run in our evaluation, and explain how we instrument our target designs.

7.1 Target Designs

Our target designs are derived from the Rocket Chip generator [2], which contains Rocket, a single-issue in-order scalar core implementing the RISC-V ISA (RV64IMAFDC). Rocket Chip has been taped out over a dozen times for both research and commercial purposes. In our experiments, we use the default configuration of Rocket Chip, which includes a 16 KiB L1 I$, a blocking 16 KiB L1 D$, and 32-entry fully-associative L1 I and D TLBs. We only change the default configuration to increase the number of performance counters and deepen the L2 TLB to 1024 entries.

At the system level, our targets consist of single or quad-core instances of Rocket Chip (labeled SC or QC), composed with either a latency-bandwidth pipe or a quadruple-rank DDR3-2133 (14-14-14) FCFS or FR-FCFS DRAM models over a 64-bit AXI4 bus. In all targets, the simulated system has 16 GiB of DRAM capacity. Additionally, two targets include a 4 MiB LLC model (labeled LLC). In the target's periphery, we have a UART and block device that interact with simulation models co-hosted in software.

We report the utilization of several host-mapped targets in Table 4. We separate the utilization contributions into "Simulator" (the component of the design generated by MIDAS, including the FASED instance), and Memory Model (only the FASED instance). Not shown is the contribution of the F1 shell, which consumes a constant 14.5%, 10%, and 16.7% of the XCVU9P's Logic LUT, Register, and 36K BRAM resources respectively.

7.2 System Software

All benchmarks were run on Linux kernel version 4.15.0-rc6. We built base Linux distributions with Buildroot and BusyBox. As part of our FPGA batch-job submission scripts, these base images are modified to add the desired workload and to run the workload immediately after Linux boot by altering the `init` script. During simulation, target processes pipe standard out to the target filesystem. We retrieve these files by remounting the filesystem on the host-CPU after the simulation has completed.

7.3 Instrumentation

To measure core-side performance counters, we run a target program that, on a one-target-second timer interrupt (1 billion cycles), reads a core's performance counters and dumps them to the target filesystem. We pin an instance of this program to each core. To measure DRAM-side statistics, we pause the simulator every one-billion target cycles and read out the memory-mapped instrumentation registers. Unlike using a target program to obtain these values, this approach does not alter the target's behavior.

7.4 An Overview SPEC2017int On Rocket

In our experiments, we run SPEC2017 intrate and intspeed suites with reference inputs, cross-compiled for RISC-V systems with the `-O2` flag. Intspeed benchmarks require as much as 16 GB of memory, while intrate benchmarks require 2 GB per copy. In Table 5, we give each suite's dynamic instruction count and L1 MPKIs when running on the Rocket configuration of Section 7.1

8 DEMONSTRATING FASED

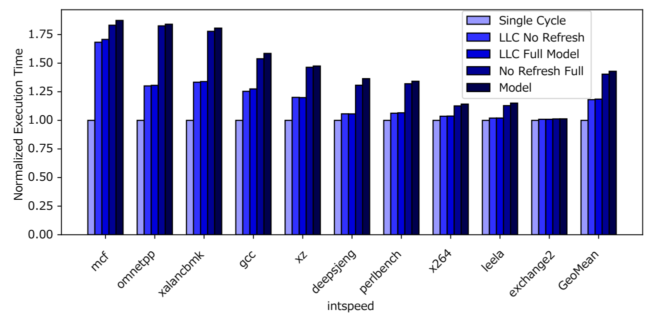
In this section we demonstrate FASED in a series of small experiments and study its performance characteristics.

8.1 Working Set Study

Caches help insulate processors against long and variable latencies to DRAM. Unfortunately, large LLCs are difficult to implement directly on FPGAs due to area limitations. FASED enables the architect to explore cache sizes that would otherwise be impossible to host on a single FPGA. We explore this by sweeping LLC-cache sizes from 64 KiB to 1 MiB in size. In Figure 6 we show the speed up provided by different cache sizes and contrast them against a cache-less and single-cycle memory-system.

8.2 Effects of DRAM Limitations

In this experiment, we quantify the extent to which specific timing details affect target execution time. We start with the validated model, *Full*, and gradually strip out features: *No Refresh* disables



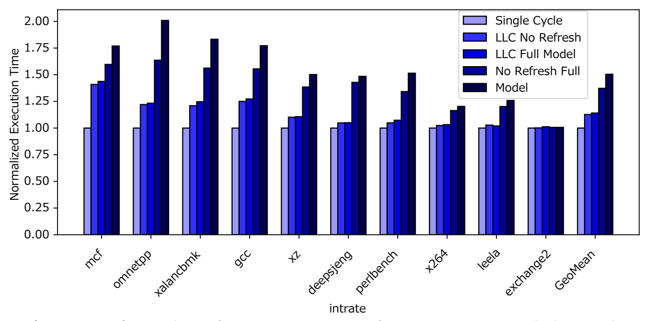


Figure 5: Target-execution time of SPEC2017 intspeed and intrate (4 copies) with reference inputs for DRAM models with and without refresh enabled. Runtime is normalized to that of a single-cycle memory system. LLCs, if present, are 256 KiB and 1 MiB large for intspeed and intrate respectively and are 8-way set associative.

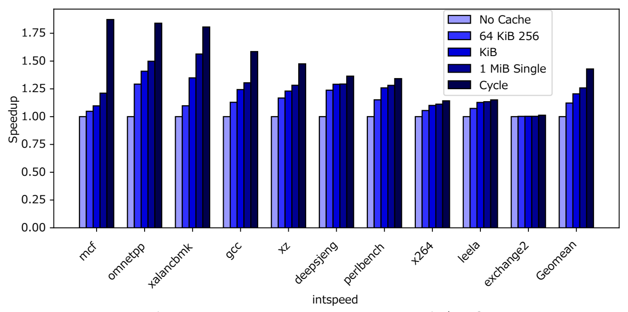


Figure 6: Speedup in SPEC2017 intspeed (reference inputs) vs LLC model size. All caches are 8-way set associative.

refresh, *No ACT Limits* sets t_{FAW} and t_{RRD} to zero, and finally, *Ideal* removes all other timing considerations[3].

The slowdowns of these models relative to a single-cycle memory system are shown in Figure 5. Without an LLC, the largest source of slowdown is refresh, which contributes a 1.01× and 1.11× slowdown for intspeed and intrate, respectively. Eliminating t_{RRD} and t_{FAW} had almost no effect; we suspect reducing rank count and increasing device density would induce a perceptible slowdown. Remaining DRAM non-idealities contribute a 1.02× slowdown. Once a cache is included, these slowdowns are effectively mitigated: refresh contributes a only 1.01× slowdown in intrate.

8.3 Simulation Performance

Table 6 gives the host-execution times and execution speeds for a handful of SPEC2017 runs. The targets with DDR3 models consistently run at rates above 100 MHz. Theoretically, FASED instances can operate at the host frequency (160 MHz) if the functional model can always serve timing model requests in time. On our host, reads and writes take on average 47 and 37 cycles, respectively, when unloaded. These latencies have a significant effect on simulator performance when modeling a single-cycle memory system as that host latency is fully exposed to the simulator, but they have relatively little impact when modeling a realistic DRAM system whose latency (cycles) is similar to that of the host. Instances with LLCs fall between these extremes: LLC hits are fast in target time and thus expose the host-DRAM latency to simulator, while misses present sufficient target-latency to hide the host-DRAM access. For our DDR3-2133 target memory system, reads take 20, 34, and 48 cycles for row hits, closed row, and row misses, respectively. These latencies are large enough that many accesses can be served by the functional model before the timing model requires them, eliding simulator stalls. Ironically, modeling a slower DDR-speed-grade slows down the simulator: since t_{CS} is smaller, reads complete in fewer target *cycles* despite taking more target *time*.

[3] t_{RTP}, t_{WTR}, t_{RTRS}, t_{RTP}=0; $t_{RAS} = t_{RCD} + t_{CS}$ $t_{RC} = t_{CS} + t_{RCD} + t_{RP}$.

8.4 Instrumentability

FASED allows the user to instrument timing models without changing target behavior. Coupled with fast simulation speeds, this allows a FASED instance to provide insight into system-wide behavior that would be difficult to collect otherwise. We demonstrate this in Figure 7, where we see how row-buffer and LLC hit rates are inversely correlated through each of 641.leela's games of Go.

9 CONCLUSION

FPGAs in the cloud provide a compelling platform to build fast, detailed full-system simulators. However, if FPGA-accelerated simulation is to see wider adoption, the usability limitations of prior work must be addressed. One promising avenue lies in automatically deriving cycle-exact, bit-exact FPGA models from synthesizable target RTL modules. This reduces model-building and validation effort while enabling researchers to also observe cycle-time, area, and power impacts using commercial ECAD tools.

FASED addresses a longstanding hole in these RTL-transforming approaches by providing models of outer cache hierarchies and DRAM memory systems—components of the target design that cannot be naively transformed from ASIC RTL. However, by applying the same transformation on a target-time timing-model, FASED obviates many of the pitfalls of handwriting FPGA-hosted models, as it separates the concerns target-behavior modeling and host-platform mapping. As a result, FASED instances are both fast—capable of running at the host frequency—and detailed—comparable to cycle-accurate software simulators of DRAM—while being far less onerous to implement.

ACKNOWLEDGMENTS

Research partially funded by DARPA Award Number HR0011-12-2-0016, RISE Lab sponsor Amazon Web Services, ADEPT/ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, NVIDIA, Siemens, and SK Hynix. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

		perlbench		gcc		mcf		omnetpp		xalancbmk		x264		deepsjeng		leela		exchange2		xz	
	Model Type	hr	*f*	hr	*f*	hr	*f*	hr	*f*	hr	*f*	hr	*f*	hr	*f*	hr	*f*	hr	*f*	hr	*f*
Speed	Single Cycle	14.4	95	20.4	73	24.7	62	13.8	67	14.0	68	12.9	123	13.4	87	8.6	119	6.9	153	50.5	90
	FCFS-256KB	14.7	100	20.8	91	25.6	102	14.1	86	14.5	88	13.1	125	13.6	91	8.6	121	6.9	153	51.8	105
	FCFS	14.7	126	20.9	113	25.7	112	14.3	119	14.7	118	13.2	137	13.6	117	8.7	135	7.0	152	52.1	110
Rate	Single Cycle	31.6	50	28.5	38	33.3	36	37.1	35	36.6	37	20.8	79	25.8	44	14.9	73	7.0	151	22.6	51
	FRFCFS-1MB	31.2	54	24.4	57	23.9	72	32.5	49	31.4	54	20.4	84	24.8	48	14.3	78	7.1	150	20.7	62
	FRFCFS	21.6	111	19.6	98	20.2	104	27.3	96	22.8	110	15.9	125	15.4	110	11.4	120	7.0	152	16.4	107

Table 6: Simulation times (hours) and rates (*f*, MHz) for SPEC2017 intspeed and intrate (four copies) running on single and quad-core Rocket Chip targets. In all cases, the FPGA-host frequency is 160 MHz.

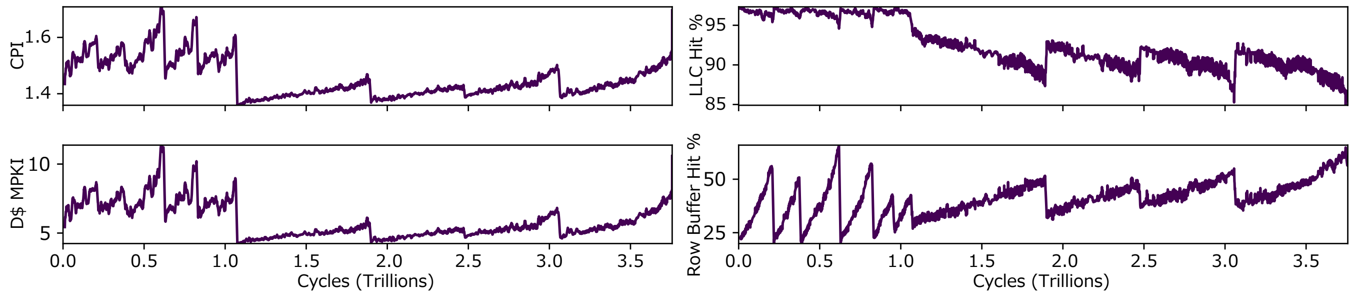


Figure 7: CPI, D$ MPKI, and row buffer and LLC hit rates running 641.leela_s with on Rocket with 256 KiB of LLC and a FCFS MAS model. These plots use a rolling average of 10 samples spaced a billion cycles apart.

REFERENCES

[1] Amazon. 2016. Amazon EC2 F1 Instances (Preview). https://aws.amazon.com/ec2/instance-types/f1/.

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

[3] Jonathan Bachrach et al. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[4] Niladrish Chatterjee et al. 2012. USIMM: the Utah SImulated Memory Module A Simulation Infrastructure for the JWAC Memory Scheduling Championship.

[5] Derek Chiou et al. 2007. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC, USA, 249–261. https://doi.org/10.1109/MICRO.2007.36

[6] Eric S. Chung et al. 2008. 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 (FPGA '08)*. ACM, New York, NY, USA, 77–86. https://doi.org/10.1145/1344671.1344684

[7] Brandon H. Dwiel et al. 2012. FPGA Modeling of Diverse Superscalar Processors. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS '12)*. IEEE Computer Society, Washington, DC, USA, 188–199. https://doi.org/10.1109/ISPASS.2012.6189225

[8] Kermin Elliott Fleming et al. 2012. Leveraging Latency-insensitivity to Ease Multiple FPGA Design. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '12)*. ACM, New York, NY, USA, 175–184. https://doi.org/10.1145/2145694.2145725

[9] 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.

[10] 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

[11] Asif I. Khan. 2008. *Emulation of Microprocessor Memory Systems Using the RAMP Design Framework*. Master's thesis. Massachusetts Institute of Technology.

[12] 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

[13] Donggyu Kim et al. 2017. Evaluation of RISC-V RTL with FPGA-Acclerated Simulation. In *CARRV '17*.

[14] Yoongyu Kim et al. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (Jan 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[15] E. A. Lee et al. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[16] Michael Pellauer et al. 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

[17] 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

[18] Scott Rixner et al. 2000. Memory Access Scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 128–138. https://doi.org/10.1145/339647.339668

[19] Paul Rosenfeld et al. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[20] 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

[21] Zhangxi Tan et al. 2015. DIABLO: A Warehouse-Scale Computer Network Simulator Using FPGAs. In *ASPLOS '15*. ACM, New York, NY, USA, 207–221. https://doi.org/10.1145/2694344.2694362

[22] Muralidaran Vijayaraghavan et al. 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

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

[24] Henry Wong et al. 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