FPGA Modeling of Diverse Superscalar Processors

Brandon H. Dwiel, Niket K. Choudhary, Eric Rotenberg
Department of Electrical and Computer Engineering
North Carolina State University
{bhdwiel, nkchoudh, ericro}@ncsu.edu

ABSTRACT

There is increasing interest in using Field Programmable Gate Arrays (FPGAs) as platforms for computer architecture simulation. This paper is concerned with modeling superscalar processors with FPGAs. To be transformative, the FPGA modeling framework should meet three criteria.

(1) *Configurable*: The framework should be able to model diverse superscalar processors, like a software model. In particular, it should be possible to vary superscalar parameters such as fetch, issue, and retire widths, depths of pipeline stages, queue sizes, etc.

(2) *Automatic*: The framework should be able to automatically and efficiently map any one of its superscalar processor configurations to the FPGA.

(3) *Realistic*: The framework should model a modern superscalar microarchitecture in detail, ideally with prototype quality, to enable a new era and depth of microarchitecture research.

A framework that meets these three criteria will enjoy the convenience of a software model, the speed of an FPGA model, and the experience of a prototype. This paper describes *FPGA-Sim*, a configurable, automatically FPGA-synthesizable, and register-transfer-level (RTL) model of an out-of-order superscalar processor. FPGA-Sim enables FPGA modeling of diverse superscalar processors out-of-the-box. Moreover, its direct RTL implementation yields the fidelity of a hardware prototype.

1. INTRODUCTION

There is increasing interest in using Field Programmable Gate Arrays (FPGAs) as platforms for computer architecture simulation. Interest is driven by the need to simulate multi-core architectures, the need for longer running benchmarks to evaluate these architectures at scale, and new capability to map large architectures to single-FPGA systems due to greater FPGA capacity and the perfecting of design multiplexing techniques.

This paper is concerned with modeling superscalar processors with FPGAs. To be transformative, the FPGA modeling framework should meet three criteria:

1) *Configurable*: The framework should be able to model diverse superscalar processors, like a software model. In particular, it should be possible to vary superscalar parameters such as fetch, issue, and retire widths, depths of pipeline stages, queue sizes, *etc*.
2) *Automatic*: The framework should be able to automatically and efficiently map any one of its superscalar processor configurations to the FPGA.
3) *Realistic*: The framework should model a modern superscalar microarchitecture in detail, ideally with prototype quality, to enable a new era and depth of microarchitecture research.

A framework that meets these three criteria will enjoy the convenience of a software model, the speed of an FPGA model, and the experience of a prototype.

Researchers have demonstrated mapping register-transfer-level (RTL) models of fixed commercial designs to FPGAs [5][8][11]. Key superscalar dimensions (*e.g.*, pipeline stage widths, pipeline depth, *etc*.) are not configurable, however. FAST [2] enables users to construct abstract timing models of different superscalar processor configurations and synthesize them to FPGAs, but their fidelity is limited by the abstract modeling approach.

This paper describes *FPGA-Sim*, a configurable, automatically FPGA-synthesizable, and register-transfer-level model of an out-of-order superscalar processor. FPGA-Sim enables FPGA modeling of diverse superscalar processors out-of-the-box. Moreover, its direct RTL implementation yields the fidelity of a hardware prototype.

There are two major aspects to this modeling effort. The first is creating the configurable RTL model of the superscalar processor. The second is automatically and efficiently mapping any one of its configurations to an FPGA.

1.1 Configurable RTL Model

We leveraged the FabScalar toolset [3] to develop the configurable RTL model. Our approach was to use FabScalar to generate the widest and deepest superscalar processor in its repertoire – a superset core. We then inserted Verilog preprocessor macros within each canonical pipeline stage to be able to narrow it to a desired width and collapse it to the desired degree of sub-pipelining. So the RTL model is *statically* configurable: the footprint of the core is that of the chosen configuration, not the superset core, and there is no extra synthesized logic or configuration bits for this configurability.

1.2 Automatic and Efficient FPGA Mapping

FPGA-Sim is synthesizable from the outset due to its direct RTL implementation. The main challenge lies in automatically and efficiently mapping the superscalar processor's RAMs and CAMs to the FPGA. Without intervention, all memory structures are synthesized to flip-flops and random logic. This is untenable because a superscalar processor has many memory structures, most are highly-ported, and some are quite large.

978-1-4673-1146-5/12/$31.00 ©2012 IEEE

Consequently, the superscalar processor's RAMs and CAMs need to be mapped to the FPGA's distributed/block RAMs. There are three difficulties in this regard:

- Whereas the processor's RAMs and CAMs are highly-ported, the FPGA's distributed/block RAMs are only dual-ported.
- A typical FPGA does not have native CAMs.
- FPGA-Sim must automatically support arbitrary superscalar widths, hence, an arbitrary number of ports for each modeled RAM or CAM structure.

With respect to the first issue, FPGA-Sim employs two techniques to map highly-ported RAMs to dual-ported distributed/block RAMs. The first technique is *replication* [8][11]. A RAM with R read ports and 1 write port can be implemented with R distributed/block RAMs. Furthermore, W write ports can be implemented by further replicating each of these "read-replicas": each read-replica now consists of W distributed/block RAMs instead of just 1 distributed/block RAM, and control bits associated with each row keep track of which of the W distributed/block RAMs has the latest-written value for that row. The second technique is *time multiplexing* [2][6][10]: using one distributed/block RAM and taking as many FPGA cycles as needed to complete all the reads and writes that would occur within a single processor cycle. FPGA-Sim can combine replication and time multiplexing for intermediate space/time solutions. The Verilog module of each FPGA-Sim memory structure is passed a parameter specifying the number of time multiplexing cycles to use. This parameter controls the tradeoff between resource utilization and FPGA cycles. The minimum setting uses the fewest FPGA cycles and most resources, the maximum setting uses the most FPGA cycles and fewest resources, and there are points in between these two extremes.

For the second issue, a CAM can be modeled with a RAM indexed by the search value [13]. An entry contains a bit vector indicating which entries in the modeled CAM match the value (*i.e.*, the bit vector is the modeled CAM's match lines). Writing the CAM involves setting a bit in one bit vector (the bit vector associated with the new value) and clearing the same bit in another bit vector (the bit vector associated with the replaced value). The latter requires a second RAM that mirrors the modeled CAM's contents, to know which value is being replaced.

Finally, to support arbitrary superscalar widths, we developed an extensive library of RAM and CAM Verilog modules. There is a distinct module for a given configuration of read and write ports. Other dimensions are parametrized within a module: number of rows, number of bits per row, and degree of time multiplexing (as discussed above).

Synthesizing a processor configuration to the FPGA is automated. The user specifies stage widths, depths, structure sizes, etc., in a microarchitecture parameters file. Also included in this file is the FPGA-cycles-per-model-cycle ratio (time multiplexing factor). A script replaces all FabScalar memory structure instantiations with FPGA-Sim memory structure instantiations: these instantiations have the correct number of logical read and write ports, according to the desired pipeline stage widths, and map efficiently to distributed/block RAMs as discussed above. A synthesis script directs the synthesis tool to read the parameter file and shape the superset RTL model into the desired core.

1.3 Paper Outline

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 elaborates on the first aspect of this modeling effort: creating the configurable RTL model. Section 4 elaborates on the second aspect: automatically and efficiently mapping arbitrary superscalar configurations to a single FPGA. Section 5 describes the overall FPGA-Sim setup. Section 6 presents our methodology. Results, including comparisons among FPGA-Sim, a cycle-accurate C++ simulator and RTL simulations, are presented in Section 7. Finally, we summarize the paper and discuss future work in Section 8.

2. RELATED WORK

This paper is the culmination of a Master's thesis which provides more details [16].

FAME Taxonomy

FAME [9] is a comprehensive taxonomy of FPGA modeling techniques along three key dimensions: (i) directly mapped to hardware *versus* decoupled FPGA and model clocks, (ii) full RTL model *versus* abstract model, and (iii) single-threaded *versus* multi-threaded. Multi-threaded, in this context, describes simulators that map multiple logical cores to a single physical core on the FPGA. The logical cores are scheduled similarly to an operating system scheduling threads.

These three dimensions define four FAME levels for categorizing FPGA-accelerated simulators: *Direct* (directly mapped to hardware, full RTL model, single-threaded), *Decoupled* (decoupled FPGA and model clocks, full RTL model, single-threaded), *Abstract* (decoupled FPGA and model clocks, abstract model, single-threaded), and *Multi-threaded* (decoupled FPGA and model clocks, abstract model, multi-threaded). FPGA-Sim falls under the *Decoupled* category.

FPGA Modeling of Superscalar Processors

There is significant precedent for mapping commercial superscalar processors to FPGAs, achieving fast and cycle-accurate simulation [5][8][11]. Each model is for a fixed microarchitecture. While varying some parameters may be straightforward (*e.g.*, structure sizes), others may require significant manual effort (*e.g.*, pipeline stage widths). The goal of our work is to automate FPGA modeling of diverse superscalar cores, including along fundamental dimensions such as width and depth.

FAST [1][2] is a general framework for accelerating software simulators. The simulator is partitioned into a functional model that runs on a host CPU and a timing model that runs on a FPGA. The functional model streams the dynamic instruction stream to the timing model (trace-driven

simulation). This distribution of effort is powerful because it places complexity on the platform best suited for that complexity: full-system simulation of complex instruction sets is suited to the host CPU and timing simulation is suited to FPGA acceleration. On the other hand, coordination between the functional and timing models is a potential performance bottleneck. FAST's library enables users to construct abstract timing models of different superscalar processor configurations and synthesize them to FPGAs, but their fidelity is limited by the abstract modeling approach. In contrast, FPGA-Sim's direct RTL implementation yields the fidelity of a hardware prototype. Moreover, unlike previous work, we explicitly demonstrate and extensively explore diverse working configurations out-of-the-box, made possible by a single configurable RTL model.

HAsim [7], like FAST, splits the functional and timing models and the timing model is abstract. The main difference with FAST is that HAsim tightly couples the two models on the FPGA. Thus, a key goal of the HAsim work was to construct a single functional model that works with arbitrary timing models. Three timing models were manually written: unpipelined, 5-stage in-order scalar, and a 4-way OOO superscalar.

FPGA Modeling of Multiprocessors

RAMP Gold [10] and ProtoFlex [4] are full-system simulators of large multiprocessor and many-core systems, implemented entirely on FPGAs. Both simulators focus on modeling caches, cache coherence protocols, and the interconnection network. Consequently, RAMP Gold uses a simple in-order core and ProtoFlex does not model processor timing.

Green Flash [14] maps multiple cores to a single FPGA. It uses FPGA simulation to expedite design-space exploration for the optimal processor to be used in a low-power, highly-tuned supercomputer. The design space includes a non-configurable, in-order processor with configurable caches. In contrast, FPGA-Sim provides a significantly more complex out-of-order processor that is configurable as well.

Mapping RAMs and CAMs to FPGAs

The replication technique, for mapping highly-ported memory structures onto an FPGA's dual-ported memory structures, was briefly discussed in [8][11]. These references do not describe a detailed implementation of replication, however. On the other hand, we provide details of our implementation.

The authors of FAST [2], A-ports [6] and RAMP Gold [10] describe the time multiplexing technique (multiple FPGA cycles for each model clock) for emulating highly-ported memories with dual-ported memories. We leverage this idea, as well.

What sets FPGA-Sim apart is its novel combination of replication and multiplexing, and the automated synthesis of replication and multiplexing from a single abstract parameter, MFMR (Section 4.2).

Prior works model CAMs with serial searches [2][7], requiring stalling the model clock until a match is found or until all entries are searched, depending on the modeled structure. We leverage a tip from the Xilinx application notes [13] for modeling a CAM with two RAMs, which enables constant-time searches and writes. We generalize the implementation for an arbitrary number of match and write ports, using replication and multiplexing.

3. CONFIGURABLE RTL MODEL

We used FabScalar [3] to develop the configurable RTL model of the superscalar processor. The FabScalar tool generates synthesizable RTL of arbitrary cores within a canonical superscalar template. The template is shown in Figure 1. To generate a core of a desired configuration, the core generator selects a design for each canonical pipeline stage from a Canonical Pipeline Stage Library (CPSL). The CPSL contains multiple designs for each canonical pipeline stage that differ in their *complexity* and *degree of sub-pipelining*. The complexity of a canonical stage depends on its superscalar width and the sizes of stage-specific structures (RAMs and CAMs). A canonical stage is nominally unpipelined, but it may be sub-pipelined deeper to achieve a higher clock frequency or to accommodate increased complexity in the stage for the same frequency.

Our approach was to use FabScalar to generate the widest and deepest superscalar processor in its repertoire – a superset core. We then inserted Verilog preprocessor macros within each canonical pipeline stage to be able to narrow it to a desired width and collapse it to the desired degree of sub-pipelining. Because the Verilog preprocessor is used, the RTL model is *statically* configurable: the footprint of the core is that of the chosen configuration, not the superset core, and there is no extra synthesized logic or configuration bits for this configurability. The preprocessor strips away excess superscalar ways in the front-end pipeline stages, execution lanes in the backend, etc.

4. EFFICIENT FPGA MAPPING

4.1 Challenges

The biggest challenge with synthesizing an out-of-order superscalar processor to an FPGA is efficiently mapping its many multi-ported RAM and CAM structures to FPGA resources. Table 1 shows the major RAM and CAM structures and the number of ports as a function of fetch width (F), dispatch width (D), issue width (X, the number of execution lanes), the number of load/store execution lanes (M), and retire width (R).

An FPGA is comprised of slices containing mostly LUTs and storage elements, the two most abundant resources. A Xilinx Virtex-5 slice contains four 64x2-bit LUTs and four single-bit registers or latches. Two slices form a combinational logic block (CLB) and CLBs are arranged in a grid. Communication is fastest within a slice and degrades across slices and CLBs.

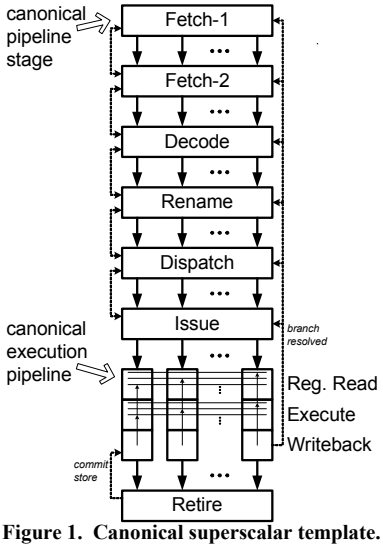


Figure 1. Canonical superscalar template.

Memories on FPGAs are limited to two ports and are either composed from multiple LUTs, called distributed RAMs, or dedicated large SRAM structures in specific locations, called block RAMs. Synthesis tools will map any memory with more than two ports to flip-flops instead of the more efficient distributed or block RAMs. Table 2 shows the resource utilization of a 4R2W (4 read ports, 2 write ports) 32x32-bit register file for a dual-issue superscalar processor, when we code the register file for distributed RAMs or use naïve coding and leave it up to synthesis. The distributed RAM implementation uses *eight* 32x32-bit dual-ported RAMs to provide the necessary four read ports and two write ports. (As explained in Section 1.2: 4R1W requires four read-replicas, and making these 2W requires two replicas for each read-replica, yielding 4x2 = 8 replicas total.) The dual-ported RAMs consume roughly 24 LUTs each. The naïve implementation lets the synthesis tool implement the same register file using flip-flops. The entire register file is placed into 1,024 flip-flops and uses multiplexers for both reading from and writing to rows. Because a slice contains only four flip-flops, the register file occupies more than five times the number of slices. This becomes a heavy burden for place-and-route because communication between slices must pass through multiple hops.

Managing the resources used by each superscalar structure in an efficient manner is a key necessity if larger and wider processors are to fit on a single FPGA. As proof, we found that naïvely synthesizing a 2-way superscalar processor with 64 physical registers and a 16-entry issue queue runs out of resources on the Virtex-5 FPGA used in this paper. A potential work-around is to partition the design and synthesize it across multiple FPGAs. This approach incurs the design effort of RTL partitioning and extends simulation time (either by decreasing the FPGA frequency or adding FPGA stall cycles) due to frequent off-chip communication among partitioned modules. Moreover, it leads to inefficient utilization of individual FPGAs.

4.2 Implementation

4.2.1. Clock Decoupling

Decoupling the processor clock from the FPGA clock is vital for fitting large processors on an FPGA and for mapping complex operations to the general-purpose structures of FPGAs. With clock decoupling, multiple FPGA cycles may elapse during each model cycle. The model cycle advances only after all operations for the model cycle have completed. The simulator performance becomes dependent on the FPGA-to-model cycle ratio (FMR). Although using more FPGA cycles per model cycle increases the simulation time, FPGAs are still able to outperform software simulators.

Two prior implementations for decoupling the clocks are 1) defining a static FMR for all model cycles and 2) dynamically determining the FMR for each model cycle. A static FMR requires no global synchronization when advancing to the next model cycle because each stage knows exactly when the next model cycle begins. Large designs benefit by not adding globally-routed wires for the synchronization. However, each model cycle takes as many FPGA cycles as the longest operation; if this operation is infrequent, many FPGA cycles are wasted.

The second approach, dynamically adjusting the FMR based on events in the current model cycle, eliminates useless FPGA cycles and reduces the total number of FPGA cycles needed to complete the simulation. A centralized controller advances the model cycle once all stages have reported completion of the current cycle. Infrequent events requiring

Table 1. Structures in the superscalar processor and whether explicitly coded for distributed RAM (d RAM), block RAM (b RAM), or left up to synthesis.

Structure	Read Ports	Write Ports	d/b RAM
L1 Instruction Cache (*2 banks*)	1 shared read/write port per bank		b RAM
Branch Target Buffer (*F banks*)	1 / bank	1 / bank	b RAM
Branch Predictor (*2 banks*)	2 / bank	2 / bank	b RAM
Fetch Queue	D	F	d RAM
Rename Map Table	2D	D	d RAM
Free List	D	R	d RAM
Arch. Map Table	R	R	d RAM
Active List (ROB)	R	D	d RAM
Memory Dependence Predictor	D	1	b RAM
Issue Queue Wakeup CAM	X	D	d RAM
Issue Queue Payload	X	D	d RAM
Physical Register File	2X	X	d RAM
Load Queue CAM	M	M	left up to synth.
Store Queue CAM	M	M	
L1 Data Cache	M	1	b RAM

many FPGA cycles affect only the model cycles in which they occur instead of delaying every model cycle. The drawback is routing congestion: every flip-flop needs to be synchronized by the centralized controller. Pellauer et. al [6] showed that increasing the number of modules requiring synchronization – from 25 to 100 – increases cycle time by 35% and increases place-and-route time by 20x.

FPGA-Sim combines the static and dynamic FMR approaches for a hybrid solution: a specified *minimum FMR (MFMR)* dictates the minimum number of FPGA cycles before the model cycle is advanced, and only the few events that require stalling the model cycle further report to the central controller when additional FPGA cycles are needed. Our current implementation has three such events: an instruction cache miss, a data cache miss and complex arithmetic operations.

We performed an experiment to compare the cycle time and place-and-route time of our hybrid approach versus the fully-dynamic approach. In our context, the distinction is as follows. For the hybrid approach, all superscalar memory structures complete their reads/writes within the MFMR; consequently, the centralized controller need not receive completion signals from these modules. For the fully-dynamic approach, each superscalar memory structure separately signals completion to the centralized controller. The experiment considers both a large core (high resource usage) and a small core (low resource usage); and the MFMR is set such that memory structures take minimal resources, *i.e.*, for RAMs and CAMs, time multiplexing was used over replication as much as possible. We found no difference in cycle times between our hybrid approach and the fully-dynamic approach, for either core, but we did observe increases of 3% (large core) and 34% (small core) in place-and-route time for the fully-dynamic approach due to the increased global routing.

4.2.2. Modeling a Multi-ported RAM

In general, FPGA synthesis tools will synthesize FabScalar RAMs to flip-flops, as they have many ports. We replaced the FabScalar RAMs with a library of functionally-equivalent, FPGA-aware RAMs. Each FPGA-Sim RAM, whether using distributed or block RAMs, provides the correct number of logical ports through a combination of RAM replication and time multiplexing. The degree of time multiplexing is controlled by the MFMR. The remaining ports, if any, are fulfilled by RAM replication. In this section, we describe our implementations of replication and time multiplexing.

Replication

With replication, multiple dual-ported RAM replicas emulate a multi-ported RAM.

Table 2. Resource utilization of a 4R2W register file.

Synth. Mapping	LUTs	Flip-flops	Occupied slices
Dist. RAM	373	32	94
Flip-flops	1,882	1,024	472
Difference	5x	32x	5x

Figure 2a shows a 2-read (2R) and 1-write (1W) RAM using two dual-ported distributed RAMs. The two logical read ports map to separate distributed RAMs and the logical write port goes to both so that they receive the same data. Each read port has a dedicated distributed RAM to access that copy of the data. The cost of each read port is one additional distributed RAM, thus the cost of an M-read, 1-write RAM is M distributed RAMs.

Increasing write ports uses a similar technique except that the distributed RAMs contain different data and the RAM with the most recent write to a row must supply the data for a read operation to that row. Figure 2b shows a 1R 2W RAM and additional logic for reads. The wiring to the distributed RAM ports follows the same concept as before: the two logical write ports map to separate RAMs and the single logical read port maps to both RAMs. When performing a read, two different values are read. The most recently written value is chosen by the `ram_select_vector`. Each row of the `ram_select_vector` contains the ID of the distributed RAM that was written to last at that row. A read operation uses the ID to determine which value to select from the multiplexer. Implementing a 1R 2W RAM requires a `ram_select_vector` that is itself a 1R 2W RAM. Fortunately, it can be efficiently synthesized to flip-flops since its width is only $\log_2$[# of write ports]. To sum up, a 1-read, N-write RAM uses N distributed RAMs plus the additional logic for the `ram_select_vector`, the complexity of which depends on N and the depth of the RAM.

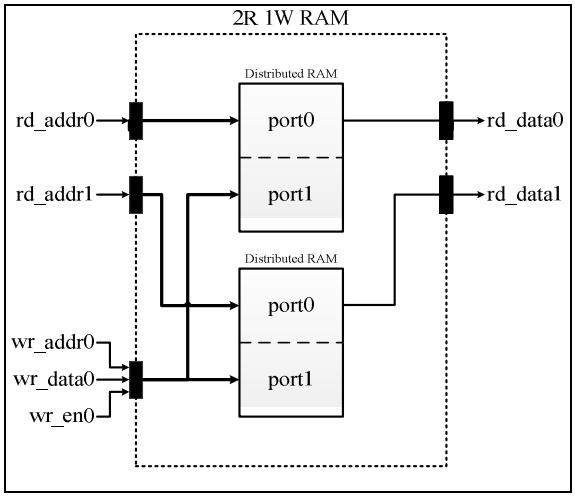
Combining the two techniques facilitates multiple read ports and multiple write ports. As a general rule, M*N dual-ported RAMs are needed for an M-read, N-write RAM. Figure 2c illustrates a 2R 2W RAM.

Time multiplexing

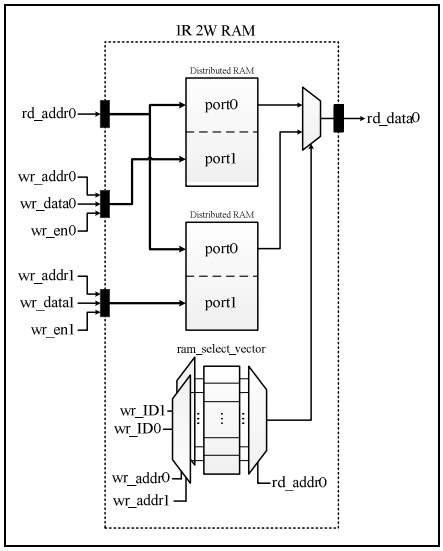
Time multiplexing maps multiple logical ports to fewer physical ports over multiple cycles. Figure 3 shows a 2R 2W RAM implemented with one dual-ported RAM. Two FPGA cycles are required to complete the accesses, *i.e.*, the MFMR must be at least two. The signal `vstall` is the active low clock enable used to stall the pipeline, *i.e.*, the model clock advances to the next cycle at the rising edge of clock if `vstall` is low. Each physical port supports a read in the first cycle followed by a write in the second cycle. During the first cycle, the state machine routes `rd_addr0` to `port0_addr` and `rd_addr1` to `port1_addr`. At the rising edge of the clock, the data read from each port is latched into a dedicated register and is available to the pipeline.

4.2.3. Modeling a Multi-ported CAM

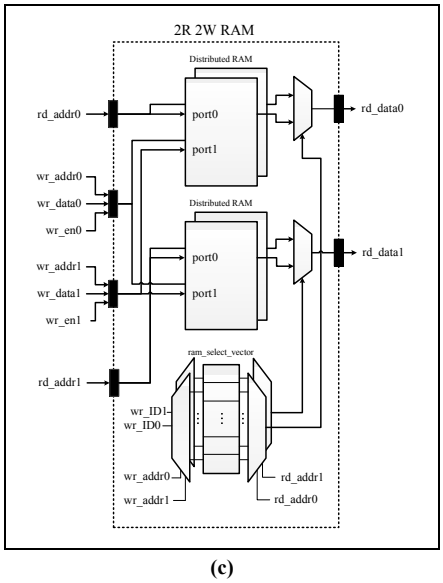
Modeling multi-ported CAMs has challenges beyond just increasing the number of ports because there is no native CAM structure to replicate. Performing a serial search over multiple cycles is a simple approach, but negatively impacts simulation speed when the entire contents must be searched. For example, the entire Issue Queue Wakeup CAM must be



(a)



(b)



(c)

Figure 2. (a) 2R 1W RAM, (b) 1R 2W RAM, and (c) 2R 2W RAM implemented with RAM replication.

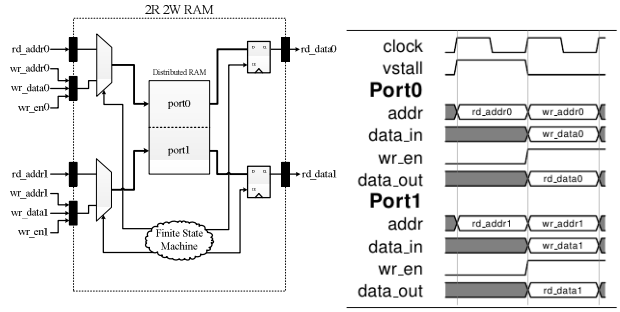


Figure 3. Time multiplexed 2R2W RAM.

searched to wake up consumer instructions. With an example issue queue size of 32, each model cycle would need 32 FPGA cycles. Instead, we model CAMs by storing each match-vector in a RAM [13]. This shifts implementation complexity from the match operation to updating the match-vectors during a write.

The example in Figure 4 demonstrates the operations of a conventional CAM (left) and an FPGA CAM (right). The conventional CAM contains eight entries, each two-bits wide. The FPGA CAM requires two RAMs: one with the same dimensions and contents of the CAM (Value-RAM) and the other with transposed dimensions of the CAM (Vector-RAM) (*i.e.*, the RAM width is equal to the CAM depth and the RAM depth is $2^{CAM\ width}$). The Vector-RAM is indexed by a search value and contains match-vectors for each possible search value. The initial state is shown in Figure 4a. In Figure 4b, the CAMs are searched for the value 3. The conventional CAM compares each entry with 3 to *generate* a match-vector. The FPGA CAM simply *reads* the match-vector from entry 3 of the Vector-RAM. Both implementations take one cycle to complete the match operation.

In Figure 4c, the value 0 is written to entry 2 of the conventional CAM. The same effect is achieved in the FPGA CAM by clearing a bit in the previous value's (3's) match-vector and setting a bit in the new value's (0's) match-vector. The purpose of the Value-RAM is to provide the previous value. The address (2) determines which bit is updated in both match-vectors. Updating a single bit of the match-vectors is simplified by the fact that these RAMs are actually composed of 1- and 2-bit LUTs, each having a write enable signal.

The Issue Queue Wakeup CAM requires the same number of match ports as the issue width and write ports as the dispatch width. Creating a multi-ported CAM requires replication and/or time multiplexing, as was done for RAMs. The diagram of a 1-match (1M) 2-write (2W) CAM is shown in Figure 5. In this example, two write ports are achieved via replication. Only the logic for the first write port (`Value_RAM_0`, `Vector_RAM_0`) is shown in detail; the logic for the second write port (`Value_RAM_1`, `Vector_RAM_1`) is identical.

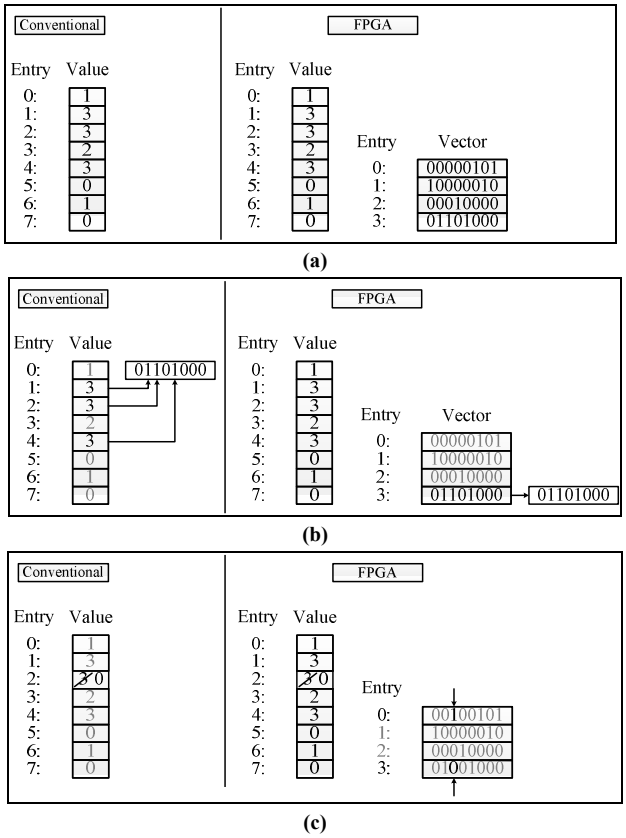


Figure 4. A conventional CAM (left) and an FPGA-modeled CAM (right).

A write requires two cycles, one cycle to set a bit in the new value's match-vector and one to clear a bit in the old value's match-vector:

1. *First cycle*: The new value, `write_data0`, is selected as the write address (addr1) of `Vector_RAM_0`. The bit corresponding to `write_addr0` is *set* in the indexed vector (bit selection is not explicitly shown), to reflect that this CAM entry will match when later searching for `write_data0`. Concurrently, the old value in the CAM entry, `prev_data0`, is read from `Value_RAM_0`, setting up the next cycle.

2. *Second cycle*: The old value, `prev_data0`, is selected as the write address (addr1) of `Vector_RAM_0`. The bit corresponding to `write_addr0` is *cleared* in the indexed vector, to reflect that this CAM entry will no longer match when later searching for `prev_data0`. If `write_data0` (new value) and `prev_data0` (old value) are the same, however, then the bit is not cleared. Concurrently, `Value_RAM_0` is updated with the new value, `write_data0`, at entry `write_addr0`.

A bit vector, called a mask-vector, is maintained for each write port. A bit is 1, only if the corresponding CAM entry was last written to by the port. To perform a match operation, all Vector-RAMs that constitute the match port – `Vector_RAM_0` and `Vector_RAM_1` in this example – are read in parallel. The match-vector from each Vector-RAM is bit-wise *AND*ed with its mask-vector, thereby passing only those bits of the match-vector contributed by the Vector-RAM. (For example, in Figure 5, the match-vector from `Vector_RAM_0` is bit-wise *AND*ed with its mask-vector, `mask_vector_0`.) The masked match-vectors from all Vector-RAMs are then bit-wise *OR*ed to produce the overall match-vector of the match port, `match_vector0`.

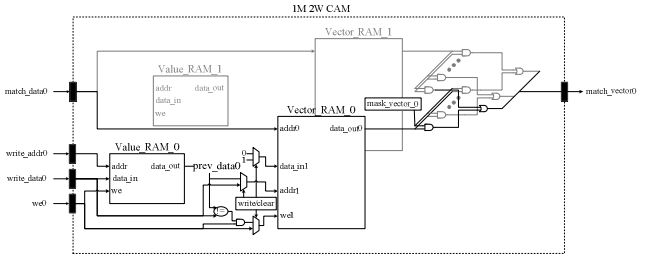


Figure 5. 1M 2W CAM.

5. FPGA-SIM FRAMEWORK

The FPGA-Sim framework is shown in Figure 6. At a high level, the framework can be divided into three major parts: (1) the modeled processor, (2) the modules that facilitate hardware simulation (Section 5.1), and (3) an optional checker core for verifying correctness (Section 5.2).

5.1 Facilitating Hardware Simulation

FPGA-Sim is derived from the first version of the FabScalar toolset, which does not provide architectural support for full-system simulation, most notably, system call, MMU, and FPU support. Consequently, simulation is of a single-threaded application and only regions of the application free of system calls and floating-point instructions. Moreover, virtual addresses are physical addresses. The framework for facilitating hardware simulation is implemented accordingly. Section 8 discusses plans for full-system simulation.

5.1.1. Loading a Benchmark

Peripherals, their FPGA-synthesized controllers, and FPGA-synthesized glue logic are involved in kicking off a simulation. The FPGA's peripherals are an SD card slot and DRAM.

A benchmark checkpoint is loaded from an SD card. The checkpoint contains the architectural register and memory state of the benchmark at a precise point in its execution, the desired starting point of simulation. Checkpoints are created by functionally simulating the benchmark on a PC to the desired starting point and taking a snapshot of architectural state.

After the FPGA is programmed, an autonomous start-up sequence kicks off simulation. A state machine loads the checkpoint from the SD card, loading the checkpoint's memory state into DRAM and register state into the modeled processor's register file. The BEE3 system used in this paper has two channels of DDR2 SDRAM per FPGA, each 2 GB. The PISA ISA [12] specifies a 31-bit virtual address space, or 2 GB, which fits within a single channel. Therefore, address translation is not required: virtual addresses are physical addresses.

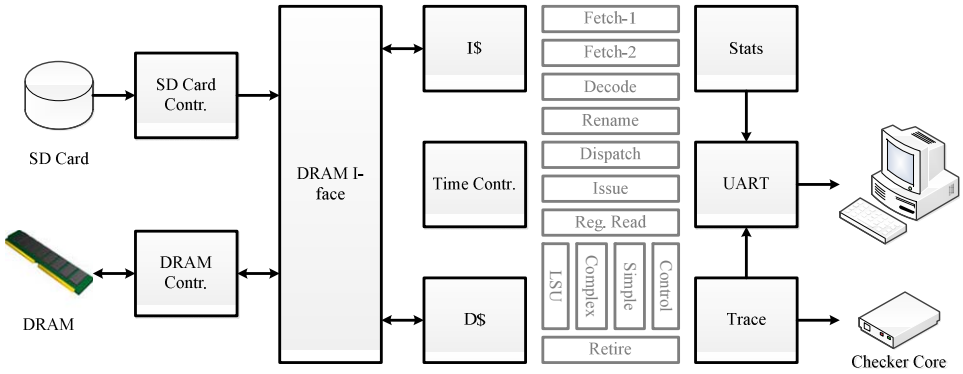


Figure 6. FPGA-Sim framework.

5.1.2. Improving Simulation Speed with Caches

The first version of FabScalar generates cores that assume single-cycle memory access, *i.e.*, perfect caches. FPGA-Sim simulates single-cycle memory by stalling the model clock until memory requests complete.

While the modeled processor always perceives cache hits, the absence of caches has the side-effect of slowing down simulation because all memory requests are serviced from DRAM. To improve simulation speed, FPGA-Sim implements 32KB instruction and data caches. More details can be found in the MS thesis [16].

With FPGA-Sim's caches in place, one might ask: why not expose them to the modeled processor? The issue is not so much the presence or absence of caches, but whether or not the pipeline is designed to stall/unstall loads due to cache misses. The next version of FabScalar will generate cores with caches and the ability to stall/unstall loads due to cache misses. It will still be desirable to simulate an arbitrary miss penalty, different from FPGA-Sim's physical latencies. This can be achieved as follows: (1) Stall the model clock when a block is not returned within the modeled miss penalty, which is a generalization of the current approach. (2) Delay signaling completion when a block is available before the modeled miss penalty.

5.1.3. Time Controller

The time controller module implements our MFMR solution. It receives inputs from the modeled processor and FPGA-Sim's caches and stalls the model clock according to MFMR and dynamic events.

5.2 Checker Core

An optional checker core turns FPGA-Sim into a verification framework. The checker core is also a debug assist.

The checker core is a 5-stage in-order pipeline. Because of its simplicity, like a purely functional simulator, it provides a golden reference model.

The BEE3 system used in this paper has multiple FPGAs connected by high-speed links. Therefore, the checker core is synthesized to a separate FPGA, though it need not be in other setups.

As instructions commit from the modeled processor, FPGA-Sim pushes their PCs and results (modifications to architectural state) into a FIFO queue in program order. FPGA-Sim builds a trace and sends it to the checker core over the inter-FPGA link. The checker core compares its per-instruction results with those from the modeled processor. If and when a mismatch is detected, the dynamic instruction count, expected value and actual value are sent to the host PC.

The checker core is not a substitute for debugging tools. Rather, it is a debug assist, quickly pinpointing the first incorrect committed instruction and setting the stage for further diagnosis.

6. METHODOLOGY

6.1 Core Configurations

Five different processor configurations that test the three superscalar dimensions (widths, depths, and structure sizes) were synthesized to a single FPGA. The configurations, their MFMR values and their cycle times are shown in Table 3. Core-1 was selected as an "average" core. Core-2 increases issue width and register file size. Core-3 is narrower (fetch and dispatch widths of 2) with smaller structure sizes. Core-4 stresses pipeline depth with a three-deep issue stage and four-deep register read stage (whereas the other cores use two-deep issue and one-deep register read). Finally, Core-5 increases both the frontend and backend widths. The MFMR range begins with the lowest value that allowed the design to both fit on a single FPGA and finish synthesis in a reasonable amount of time. We stopped increasing the MFMR once all memories were using the minimum resources.

6.2 Experimental Setup

We performed all FPGA-Sim experiments on a BEE3 system with four Xilinx Virtex-5 LX155T FPGAs. Each FPGA uses a single channel of 2 GB DDR2 SDRAM, an SD Card reader for loading checkpoints, a UART port for communicating with the host PC, and a 72-bit bus for communicating between the target core and checker core on different FPGAs. The communication bus runs at 200 MHz, the DRAM runs at 133 MHz and the core clock runs at 50 MHz. Xilinx ISE 10.1

was used from synthesis to place-and-route. FPGA-Sim can run on any FPGA system that has similar peripherals.

We compare FPGA-Sim to C++ and Verilog simulators. The C++ simulator is part of the FabScalar toolset and is intended to accurately model FabScalar-generated cores. The only notable difference is that the C++ simulator does not implement a load violation predictor whereas two of the cores in Table 3 implement one. Verilog simulations are of the equivalent FabScalar-generated cores; the Verilog simulator is Cadence NC-Verilog.

The benchmarks are six 100 million instruction SimPoints [15] from SPEC2000. The SimPoints provide regions that are free of floating-point instructions and system calls; these constraints were explained in Section 5.1.

7. RESULTS

Sections 7.1 and 7.2 explore the effect of MFMR on resource utilization and simulation speed, respectively. Section 7.3 compares the simulation speeds of FPGA-Sim, the C++ simulator, and the Verilog simulator. Section 7.4 evaluates the cycle-accuracy of FPGA-Sim and the C++ simulator, with respect to the Verilog simulator.

7.1 Effect of MFMR on Resource Utilization

The Virtex-5 LX155T has 97,280 LUTs and the same number of flip-flops. Figure 7 and Figure 8 show LUT and flip-flop utilization, respectively, for the five cores. Utilization is shown on the left vertical axis. Absolute number of LUTs or flip-flops is shown on the right vertical axis. MFMR is varied along the horizontal axis. Utilization includes the modeled core and other FPGA-synthesized modules in the FPGA-Sim framework, except for the checker core, which is synthesized to a separate FPGA.

Table 3. Core configurations used for experiments.

Parameter	Core-1	Core-2	Core-3	Core-4	Core-5
Fetch Width	4	4	2	4	5
Dispatch Width	4	4	2	4	5
Issue Width	4	6	4	4	5
Active List	128	128	64	128	128
Physical Register File	96	128	64	96	128
Issue Queue	32	32	16	32	32
Load/Store Queue	32	32	16	32	32
Fetch-to-Execute Depth	9	9	9	13	9
Branch Predictor	128 Kb	128 Kb	128 Kb	128 Kb	128 Kb
Branch Target Buffer	120 Kb	120 Kb	120 Kb	120 Kb	120 Kb
Load Violation Predictor	20 Kb	0	20 Kb	0	0
MFMR	3-9	3-11	3-9	3-9	5-11
Cycle Time	20 ns	20 ns	20 ns	20 ns	20 ns

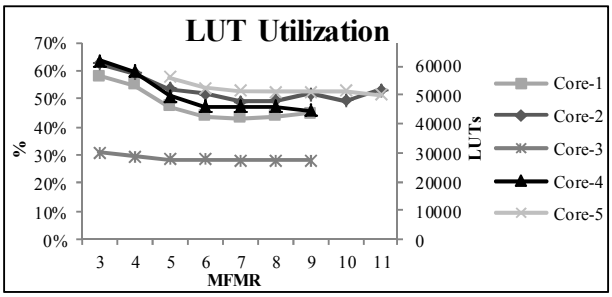


Figure 7. LUT utilization.

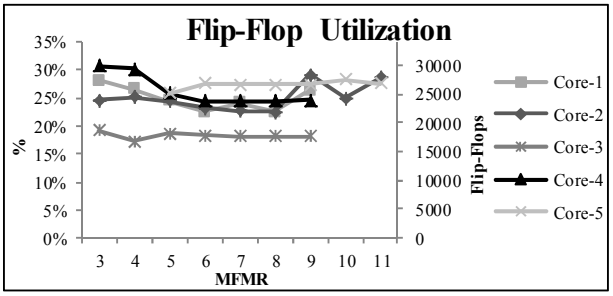


Figure 8. Flip-flop utilization.

Core-4 shows the largest drop in resources when increasing MFMR from 3 to 9, with 17,511 (28%) fewer LUTs and 6,037 (20%) fewer flip-flops. Core-1 and Core-4 are configured similarly, except that Core-1 has a load violation predictor and Core-4 has a deeper pipeline. These differences cause Core-4 to use 5,501 (10%) more LUTs and 2,581 (9%) more flip-flops when MFMR is 3. A comparison of Core-5 (fetch width 5, issue width 5) against Core-2 (fetch width 4, issue width 6) shows that fetch width has a greater impact on resource utilization than issue width: Core-5 uses more resources than Core-2 for MFMR < 9. Core-3 uses the least resources by a large margin, with 26,440 (47%) fewer LUTs and 8,728 (32%) fewer flip-flops than the next smallest core, Core-1. Core-3's front-end is half as wide as Core-1's and several of its memories are smaller. The smaller memories already use few resources and increasing MFMR provides minimal benefit.

When MFMR is sufficiently high, all logical write ports map to the same RAM, causing the `ram_select_vector` to be removed, reducing flip-flop usage. Beyond MFMR of 6, resources decrease more slowly or even increase. As the degree of time multiplexing increases, more logical ports are mapped to a physical port. This reduces RAM replication but increases the number of multiplexer inputs. Beyond MFMR of 6, the latter effect begins to cancel or outweigh the former effect.

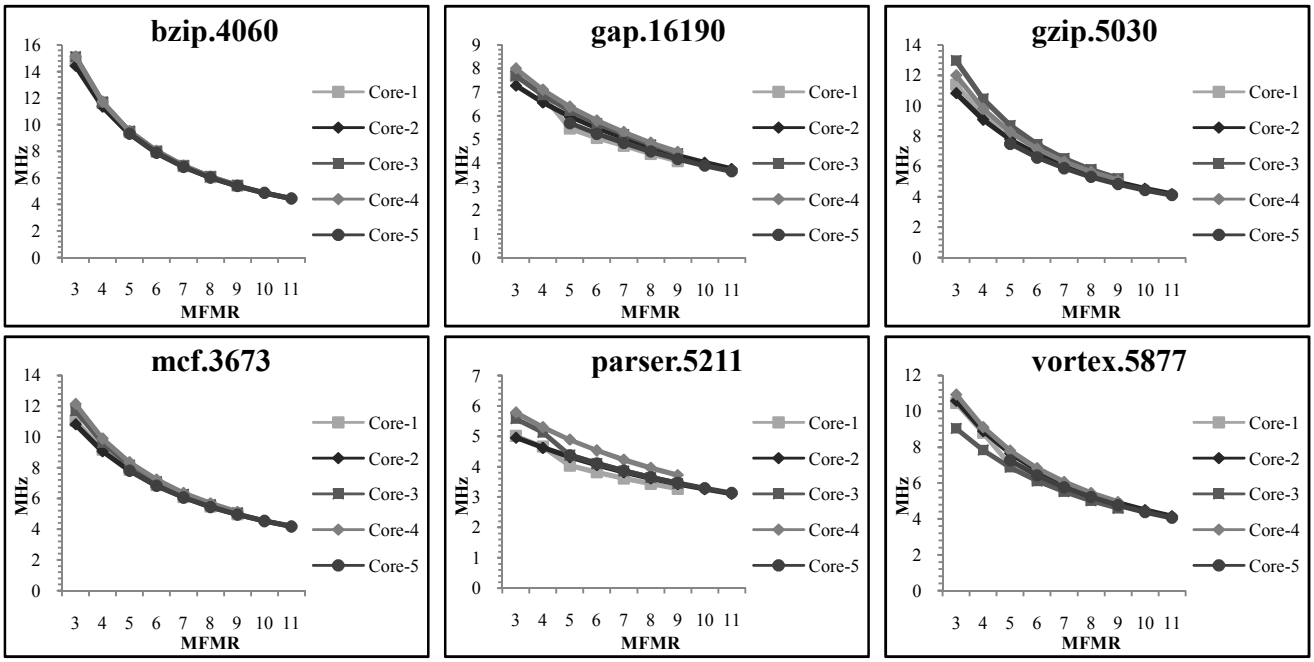


Figure 9. Effective clock frequencies.

7.2 Effect of MFMR on Simulation Speed

Simulation speed is the number of model cycles per second, which we will call the effective clock frequency. The effective clock frequency is less than the FPGA clock frequency due to extra FPGA cycles introduced by MFMR and dynamic events that stall the model clock. Effective clock frequency is calculated as follows:

$$Freq_{EFF} = Freq_{FPGA} * \frac{Cycles_{Model}}{Cycles_{FPGA}}$$

Figure 9 shows effective clock frequency as a function of MFMR. Notice that effective clock frequency depends on the benchmark: benchmarks differ in the number of dynamic events that stall the model clock (*e.g.*, memory accesses, complex arithmetic). In contrast, effective clock frequency is the same for different cores: this is the hallmark of a hardware prototype. Actually, there is a little variation across cores especially at MFMR of 3. One cause (diagnosed in gzip.5030) is a confluence of two factors: (1) Mispredicted branches may cause either advantageous or disadvantageous wrong-path prefetching into the instruction cache. (2) There can be slight variations in mispredicted branches due to different processor window sizes (different update latencies). Since cache misses stall the model clock in the current setup (Section 5.1.2), miss variation across cores causes variation in effective clock frequency.

The peak frequencies when dynamic events are ignored are 16.7 MHz at MFMR=3 (50MHz ÷ 3) and 4.5 MHz at MFMR=11 (50MHz ÷ 11), a difference of 73%. The benchmark that gets closest to the peak frequencies is bzip.4060: 15.1 MHz and 4.4 MHz, respectively, a difference of 71%. Dynamic events are rare for bzip.4060 and the simulation speed nearly reaches the maximum. On the other hand, parser.5211 has many complex arithmetic instructions that diminish the effect of the MFMR. Its effective clock frequencies are only 5.8 MHz at MFMR=3 and 3.1 MHz at MFMR=11, a difference of only 47%.

7.3 Speed against C++, Verilog

In Figure 10, we compare model cycles per second of FPGA-Sim, the C++ simulator and the Verilog simulator. All cores use MFMR=5 which is the minimum MFMR that can fit Core-5. Thus, these results are pessimistic for cores one through four (which can go down to MFMR=3). The Verilog simulator is two orders of magnitude slower than the C++ simulator and three to four orders of magnitude slower than FPGA-Sim.

The speedup of FPGA-Sim over the C++ simulator (Figure 11) ranges between 37x for Core-3 running parser.5211 and 154x for Core-5 running bzip.4060. Whereas core complexity does not affect FPGA-Sim's speed, the sequential C++ simulator tends to simulate smaller cores faster than larger cores due to fewer interactions per model cycle. Thus, FPGA-Sim's speedup is consistently highest for Core-2 (issue width of 6) and lowest for Core-3 (smallest core). On average, FPGA-Sim is 87x faster than the C++ simulator.

7.4 Accuracy

Figure 12 reports the IPCs for the three simulators. The C++ simulator cannot accurately model cores one and three because it does not have a load violation predictor, therefore, these cores are excluded from this section.

FPGA-Sim is 100% cycle-accurate with respect to RTL. The C++ simulator differs anywhere between 0% and 23%. Its IPCs match within 5% for 9 out of 18 benchmark/core combinations.

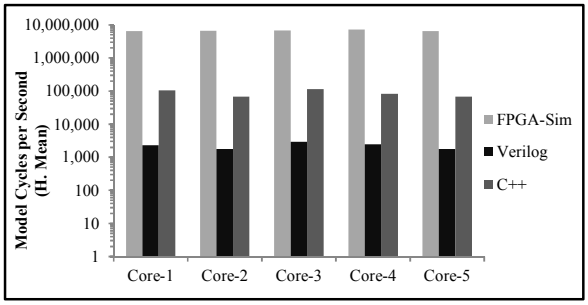


Figure 10. Performance of FPGA-Sim, C++ and Verilog.

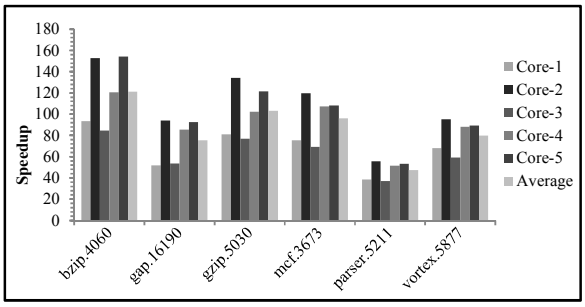


Figure 11. FPGA-Sim speedup over C++.

8. SUMMARY AND FUTURE WORK

We outlined three criteria for an FPGA modeling framework of superscalar processors, that we believe will help bring a new era and depth to microarchitecture research in the future: configurable in major superscalar dimensions, automatically and efficiently synthesizable to an FPGA, and realistic. A framework that meets these three criteria will enjoy the convenience of a software model, the speed of an FPGA model, and the experience of a prototype. Most prior FPGA modeling efforts do not provide superscalar cores at all, let alone configurable ones; among the few frameworks that do model superscalar cores, they are either fixed commercial prototypes or flexible abstract models; and the latter flexible frameworks require manual, albeit streamlined, construction of the abstract models for different configurations, as opposed to providing arbitrary configurations out-of-the-box as a software model does.

This paper makes the following contributions:

- We developed and describe *FPGA-Sim*, a configurable, automatically FPGA-synthesizable, and register-transfer-level (RTL) model of an out-of-order superscalar processor. FPGA-Sim enables FPGA modeling of diverse superscalar processors out-of-the-box. Moreover, its direct RTL implementation yields the fidelity of a hardware prototype.
- We shine a light on the primary challenge of efficiently mapping an already-synthesizable model to an FPGA: mapping a superscalar processor's pervasive, highly-ported RAMs and CAMs to the FPGA's dual-ported memory structures. The basic techniques – replication and multiplexing – are not new. Nonetheless: (1) FPGA-Sim features a novel combination of replication and multiplexing, and automatically composes the suitable replication/multiplexing mixture from a single abstract parameter, MFMR. (2) Our work is the first to describe implementations of both RAM and CAM replication in detail, for the enrichment of readers and users. (3) Our work deploys a method described in the Xilinx developer notes, whereby arbitrary CAM match and write operations are both fast; prior works implement serial searches.
- We demonstrate the ability of FPGA-Sim to model diverse superscalar processors "out-of-the-box", by actually presenting extensive experiments with five diverse cores.
- We performed in-depth experiments to evaluate the quality of FPGA-Sim: (1) Aggressive superscalar processors (wide-issue, deep pipeline, large ILP-extracting structures), approaching commercial designs, fit in a single modern FPGA. (2) FPGA-Sim is three to four orders of magnitude faster than the Verilog simulator and up to 154 times faster than FabScalar's cycle-accurate C++ simulator (87 times faster, on average). The effective clock rate of the prototype processor (which factors in MFMR and other FPGA stalls) ranges from 3.1-15.1 MHz. (3) FPGA-Sim is 100% cycle-accurate with respect to FabScalar cores whereas FabScalar's cycle-accurate C++ simulator is off by up to 23%.
- We also explored FPGA-Sim's space/time tradeoff as controlled by MFMR. (1) Increasing MFMR from 3 to 9 decreases LUTs by up to 28% and FFs by up to 20%. (2) An MFMR of 3 runs with an effective frequency that is up to 71% faster than with an MFMR of 11.

We plan to explore two major research thrusts to strengthen this work:

First, we plan to qualitatively and quantitatively assess the extensibility of FPGA-Sim. In particular, we want to revisit conventional wisdom that RTL modeling is too low level for proposing and evaluating new microarchitecture ideas. To us, RTL modeling is not strictly an issue of result fidelity, although this is clearly a must in an increasingly technology-constrained future as we try to squeeze more performance and energy-efficiency from silicon. More importantly, microarchitecture research has matured to such a great extent that RTL modeling may be the catalyst for finding new ideas and revisiting discarded ones by compelling researchers to think structurally. On the issue of extensibility, we are encouraged by several factors: (1) FPGA-Sim's single configurable Verilog representation means microarchitecture modifications extend to its arbitrary configurations. (2) Modern HDLs such as System Verilog provide structs and other conveniences that streamline describing hardware, combining the efficiency of software languages with the intuitiveness with which HDLs express concurrent hardware.

Second, we plan to extend FPGA-Sim to full-system simulation, *i.e.*, booting an operating system and running whole applications (including system calls, exceptions, interrupts, virtual memory, I/O, etc.). Only then can we derive the full benefits of FPGA acceleration.

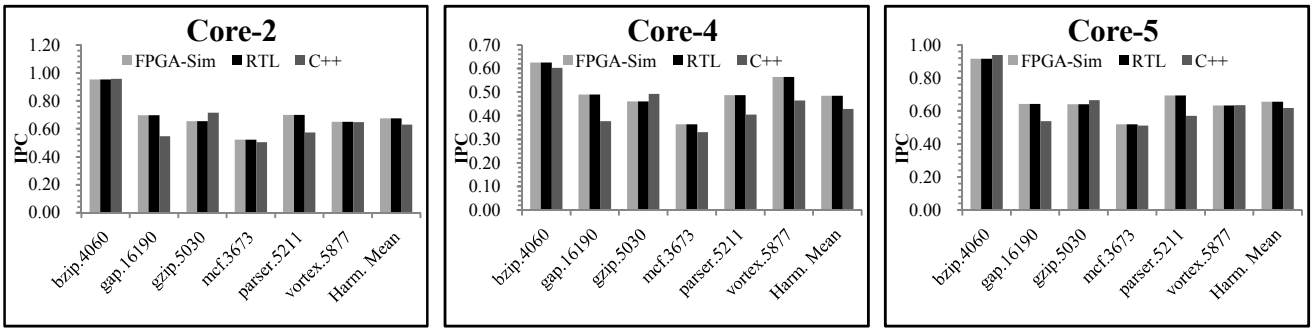


Figure 12. IPC comparisons.

9. ACKNOWLEDGEMENTS

We thank Lieven Eeckhout and the anonymous reviewers for their valuable feedback on improving this paper. This research was supported by NSF grants CCF-0811707 and CCF-1018517 and gifts from Intel and IBM. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

[1] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu. The FAST Methodology for High-Speed SoC/Computer Simulation. In Proceedings of the *International Conference on Computer-Aided Design (ICCAD)*, November 2007.

[2] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In Proceedings of the *40th International Symposium on Microarchitecture (MICRO),* December 2007.

[3] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg. FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template. In Proceedings of the *38th International Symposium on Computer Architecture (ISCA)*, June 2011.

[4] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai. ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 2(2):1-32, 2009.

[5] S. L. Lu, P. Yiannacouras, R. Kassa, M. Konow, and T. Suh. An FPGA-based Pentium in A Complete Desktop System. In Proceedings of the *15th International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2007.

[6] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. A-ports: An Efficient Abstraction for Cycle-Accurate Performance Models on FPGAs. In Proceedings of the *16th International Symposium on Field-Programmable Gate Arrays (FPGA),* February 2008.

[7] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on FPGAs. In Proceedings of the *2008 International Symposium on Performance Analysis of Systems and Software (ISPASS),* April 2008.

[8] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, S. Steibl, and H. Wang. Intel Nehalem Processor Core Made FPGA Synthesizable. In Proceedings of the *18th International Symposium on Field-Programmable Gate Arrays (FPGA),* February 2010.

[9] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson. A Case for FAME: FPGA Architecture Model Execution. In Proceedings of the *37th International Symposium on Computer Architecture (ISCA)*, June 2010.

[10] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović. RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors. In Proceedings of the *47th Design Automation Conference (DAC)*, June 2010.

[11] P. H. Wang, J. D. Collins, C. T. Weaver, B. Kuttanna, S. Salamian, G. N. Chinya, E. Schuchman, O. Schilling, T. Doil, S. Steibl, and H. Wang. Intel® Atom™ Processor Core Made FPGA-synthesizable. In Proceedings of the *17th International Symposium on Field-Programmable Gate Arrays (FPGA),* February 2009.

[12] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar ToolSet. *University of Wisconsin-Madison Technical Report* CS-TR-1308, 1996.

[13] http://www.xilinx.com/support/documentation/application_notes/xapp204.pdf

[14] M. Wehner, L. Oliker, J. Shalf, D. Donofrio, L. Drummond, R. Heikes, S. Kamil, C. Konor, N. Miller, H. Miura, M. Mohiyuddin, D. Randall, and W. Yang. Hardware/Software Co-design of Global Cloud System Resolving Models. *Journal of Advances in Modeling Earth Systems*, v. 3, p. 22, October 2011.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. Proceedings of the *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, December 2002.

[16] B. H. Dwiel. FPGA Modeling of Diverse Superscalar Processors. M.S. Thesis, Department of Electrical and Computer Engineering, North Carolina State University, November 2011.