A-Ports: An Efficient Abstraction for Cycle-Accurate Performance Models on FPGAs

Michael Pellauer[†] Muralidaran Vijayaraghavan[†] Michael Adler[‡] Arvind[†] Joel Emer[†‡]

[†]Massachusetts Institute of Technology
Computer Science and A.I. Laboratory
Computation Structures Group
{pellauer, vmurali, emer, arvind}@csail.mit.edu

[‡]Intel Corporation
VSSAD Group
{michael.adler, joel.emer}@intel.com

ABSTRACT

Recently there has been interest in using FPGAs as a platform for cycle-accurate performance models. We discuss how the properties of FPGAs make them a good platform to achieve a performance improvement over software models. Some metrics are developed to gain insight into the strengths and weaknesses of different simulation methodologies. This paper introduces A-Ports, a distributed, efficient simulation scheme for creating cycle-accurate performance models on FPGAs. Finally, we quantitatively demonstrate an average performance improvement of 19% using A-Ports over other FPGA-based simulation schemes.

Categories and Subject Descriptors

C.4 [Computer Systems Organization]: Performance of Systems – *modeling techniques*

General Terms

Measurement, Performance, Design

Keywords

FPGA, Performance Models, Simulation, Emulation, Prototyping

1. INTRODUCTION

Cycle-accurate *performance models* occupy a critical position in the modern digital system design flow. A performance model is a simulator which is available early in the design process and can be used to guide high-level architectural decisions. In order to be successful a performance model must be accurate, easy to develop and modify, and simulate the target system rapidly.

Currently design teams write most such models in software, using home-brewed C simulators or frameworks such as SystemC [14]. This eases model development, but the simulation speed of software models has not been able to keep pace with increasing complexity of modern circuits. Although academic models typically claim simulation speeds in the 100s of KIPS (Thousands of Instructions per Second) range, detailed industry models report simulation speeds in the low KIPS range. Table 1 shows an overview of simulation speeds of performance models around Intel.

FPGA'08, February 24–26, 2008, Monterey, California, USA.

Simulator Detail	Simulator speed (order of magnitude)
Low-Detail Model	100 KHz
Medium-Detail Model	10 KHz
High-Detail Model	1 KHz

Table 1: Simulation speeds of industrial software performance models of processors. Increasing levels of detail use more and more realistic core pipeline models. Only results generated from higher-detail models are considered accurate enough to guide architectural design decisions.

Parallelizing the software model can result in increased simulation speed by exposing the moderate degree of parallelism which can be exploited by contemporary multicore processors. While performance-model algorithms contain massive fine-grained parallelism, two factors make exploiting such a level of parallelism difficult in software. First, within one *model clock cycle*, the unit of parallel activity being simulated is equivalent to a small number of gates – always much smaller than the general purpose core the software runs on. Second, across model clock cycles there is a high amount of communication between these parallel regions. This high amount of communication does not map well to typical communication methods for multicores, such as shared memory.

Given these properties, FPGAs should be a good platform for efficient execution of performance models. The key insight is that one simulated model clock cycle does not have to correspond to one cycle on the FPGA. For example, a model running on a 100 MHz FPGA could take 10 FPGA cycles to simulate one model cycle and still achieve a simulation speed of 10 MHz.

Contemporary efforts to explore FPGAs as a platform for performance modeling include Penry et al.'s accelerators for the Liberty simulator [16], Chiou's UT-FAST which uses the FPGA as a timing model connected to a software functional simulator [7, 8], and the HAsim project [9, 15] which aims to create a variant of the Intel Asim simulation environment [10] on an FPGA. The stated goals of the RAMP platform [1, 19] also include creating accurate performance models.

In this paper we analyze the efficiency of techniques for creating a cycle-accurate performance model on an FPGA. We introduce novel metrics which allow the simulator architect to reason about space-time tradeoffs, and use these metrics to demonstrate why existing simulation techniques such as dynamic barrier-synchronization are unsuitable for FPGAs. We present A-Ports, an adaption of techniques from the Asim simulator for performing distributed cycle-accurate simulation. We demonstrate that A-Ports maintains the ease-of-use of existing abstractions, while increasing efficiency. We measure a quantitative performance improve-

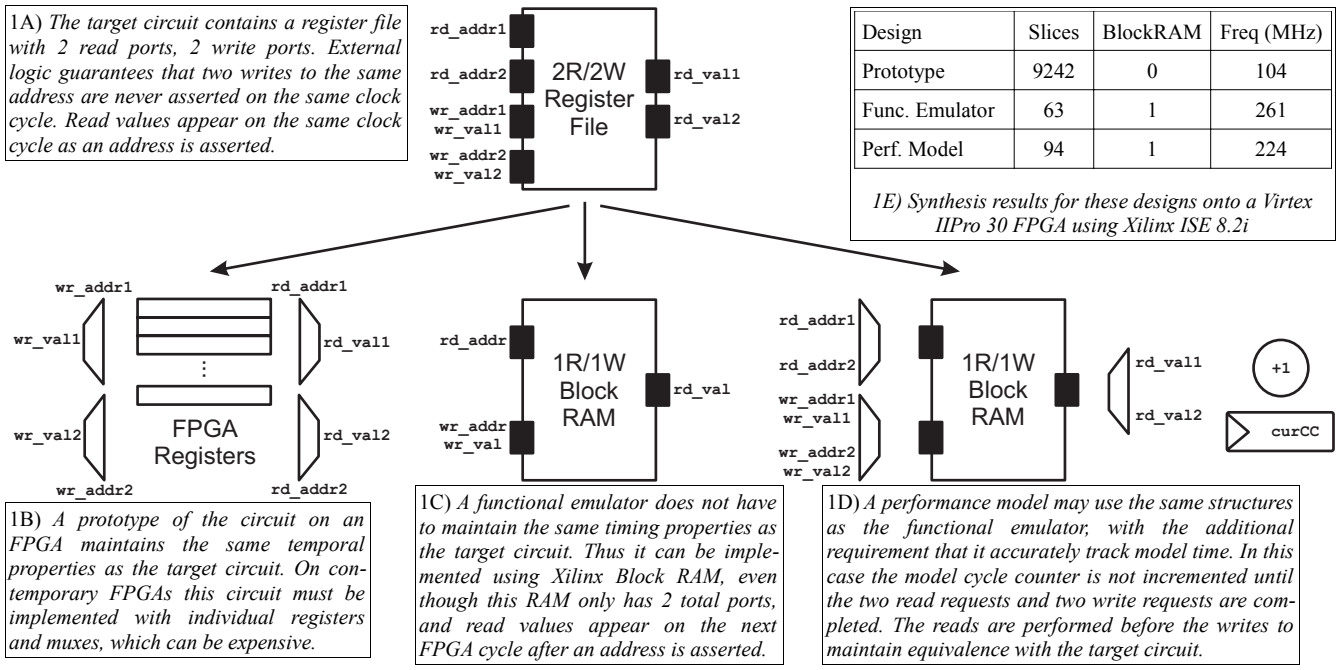


Figure 1: Taxonomy of FPGA uses in the digital system design flow, and their relative costs.

ment of 19% using A-Ports in an out-of-order processor model. We show that the performance benefit of A-Ports comes from the decoupled nature of the simulation, whereby different modules on the FPGA can be simultaneously simulating distinct points in model time. Finally, we present an algorithm whereby the decoupled A-Ports simulation can be resynchronized to the same model cycle for debugging purposes.

In this paper we limit the discussion to performance models of synchronous digital systems. Asynchronous or analog systems are not considered. Although the work was inspired by modeling processors, none of the techniques presented are specific to microprocessors. Extending the A-Ports technique to simulate multiple clock domains or globally-asynchronous locally-synchronous (GALS) models is left to future work.

2. PERFORMANCE MODELS ON FPGAS

2.1 Comparison to Prototypes and Emulators

In this section we distinguish performance models on FPGAs from more familiar uses such as *prototypes* and *functional emulators*. Consider the example of a circuit design team that wishes to implement an ASIC which contains a 4-ported register file with two read ports and two write ports. There are many ways FPGAs can help with this process. Figure 1 shows a taxonomy which summarizes three different possibilities: prototypes, functional emulators, and performance models.

Prototypes are generally created from the final or near-final RTL of the design for verification purposes. They make use of structures which will appear in the final ASIC, whether or not these structures result in efficient FPGA configuration (Figure 1E)[1]. The advantage of prototypes is complete accuracy: the waveform output of the FPGA is exactly equivalent to the output of the final design (though the clock periods themselves may differ). The disadvantage of prototypes is that, due to the high-level of detail, they are available only very late in the design process, after most major architectural decisions have been made.

In contrast, a functional emulator is a system on an FPGA which generally does not match the waveform of the target machine, except in very broad terms. Functional emulators are primarily used to give software developers fast platforms to use in code development. As a consequence, functional emulators require less effort to create and can be made available earlier in the design process. They can also make use of FPGA-optimized structures such as Xilinx BlockRAM to ensure good performance. However, they bear little or no resemblance to the final ASIC, and thus cannot be used to explore microarchitectural design decisions.

A performance model on an FPGA occupies something of a middle ground: unlike a functional emulator, it gives us the ability to measure and reason about some aspects of the target system design. Yet it must be vastly less time-consuming to create and verify than an RTL prototype. Figure 1D represents such a compromise: the structure is behaviorally accurate and results in an efficient FPGA configuration. Since this circuit takes multiple FPGA cycles, the model-cycle counter `curCC` keeps track of the simulated model clock cycle and is incremented when simulation of that cycle is complete – in this case every 4th FPGA cycle. The system architect can observe and change the number of clock cycles of operations in the target circuit without having to change the functional definition. As with software performance models, this model gives no insight into the cycle period nor circuit area of the final ASIC, nor can the RTL which is used to configure the FPGA into a performance model be of use to the design team which is seeking to implement the target circuit. However we do not consider this to be a major disadvantage as the source code for performance models written in C or SystemC is similarly useless.

1 We acknowledge that prototyping techniques such as partitioning could be applied to an inefficient configuration. Such techniques admittedly gray the lines between pure prototypes and performance models. Nevertheless, we believe that "prototyping" is a useful term to refer to direct configuration of an FPGA into a target circuit and use it to mean such.

2.2 Modeled Clocks and Space-Time Tradeoffs

When faced with a target circuit which is inefficient to implement directly on an FPGA, designers writing a performance model for an FPGA have a range of options. They can use circuits which are fast but expensive, or can trade space for time, shrinking area but either worsening the clock period or using multiple FPGA cycles to perform the simulation. Sometimes these tradeoffs can be done in such a way that the rate-limiting step of the simulator is not affected. Other times simulator performance may suffer. To this end, we have developed a series of metrics for reasoning about FPGA performance models that can aid simulator architects in making judicious tradeoffs.

The most basic metric is the FPGA-cycles-to-Model-cycles Ratio (FMR):

$$FMR = \frac{cycles_{FPGA}}{cycles_{model}}$$

In the example shown in Figure 1D, the model takes 4 FPGA cycles to simulate one model cycle, for an FMR of 4. More generally, one can examine the FMR of a single model cycle, a region, or a run. Similar to microprocessor Cycles Per Instruction (CPI) one can consider the FMR of a specific instruction or operation type in order to gain insight into performance bottlenecks. In practice, FMR is particularly useful when considering the worthiness of potential refinements to a performance model. The overall FMR of a particular run can be derived by examining each model cycle simulated:

$$FMR_{overall} = \frac{\sum_{i=0}^{numModelCC} cycles_{FPGA}(i)}{numModelCC}$$

where $cycles_{FPGA}(i)$ represents the number of FPGA cycles needed to simulate model cycle *i*. Of course, a refinement which improves FMR would be useless if it degrades the overall clock period too far. Therefore total simulator speed must take into account the frequency that the FPGA configuration can achieve:

$$frequency_{simulator} = \frac{frequency_{FPGA}}{FMR_{overall}}$$

This gives us simulator speed in Hertz. Another metric which has proved useful for software performance models is simulated Instructions Per Second (IPS). This formula takes into account the CPI of the machine being modeled:

$$IPS_{simulator} = \frac{frequency_{simulator}}{CPI_{model}}$$

Using IPS simulator designers can compare the speed at which two models execute benchmark-specific work, even if they represent two vastly different target designs, such as an in-order pipeline and an out-of-order superscalar model (Section 5). This equation adapts naturally to our purposes. Plugging in the above formula, we can deduce the IPS of an FPGA performance model:

$$IPS_{simulator} = \frac{frequency_{FPGA}}{CPI_{model} \times FMR_{overall}}$$

2.3 Correctness Issues of Simulating Time

As a performance model is a simulation rather than an implementation, we must be concerned with both the correctness of the design, and of the simulator. In order to function correctly, a performance model must be free of *temporal violations*. A temporal violation occurs when a value from model cycle *n+k* is accidentally used to calculate a value on model cycle *n*. On an FPGA, a temporal violation typically occurs because of a race condition, whereby a producer writes a value before a consumer has properly finished computing with the predecessor value.

Another issue is the ability of a simulator to advance the model clock. If the simulator is unable to advance the clock, we will refer to this as a *temporal deadlock*. Note that this is distinct from a model-level deadlock, which results when the target machine design is faulty. If the target machine enters a deadlocked state, then the performance model should correctly model the machine remaining in that state as model time continues to advance. The absence of temporal deadlocks is important because it gives the system architect confidence that a performance model which deadlocks is due to a fault in the target machine, rather than a fault in the simulation methodology.

3. EXISTING SIMULATION SCHEMES

Now that we have established some means to reason about FPGAs as platforms for performance models, let us explore how some existing simulation methodologies would adapt to FPGAs.

Many of the research efforts in distributed simulation have focused on parallel discrete event-based simulation using algorithms like those by Chandy and Misra [6] or Bryant's Time Warp [4]. Event-based simulation is successful when large amounts of the system are idle or can be excluded from results. In performance models of microprocessors such as the Intel Asim simulator [10] activity occurs consistently in almost every module on every simulated cycle, and thus event-driven approaches lose their benefit. As such we will exclude such event-based approaches from our study, though we note that there have been projects which explore event-driven simulation on FPGAs [11]. Instead we will focus on techniques suitable for continuous simulation where there is known to be one or more global clocks.

3.1 Unit-Delay Simulation

One simulation technique that can adapt well to FPGAs is unit-delay simulation, historically used in projects such as the IBM Yorktown Simulation Engine [17]. In this simulation all modules are rate-limited to the slowest parallel element.

Consider the example of the 4-ported register file from Figure 1. The performance model in Figure 1D uses 4 FPGA cycles to simulate one model cycle. If this is the slowest module in our model, then we could simply assign all modules in our system 4 FPGA cycles to simulate one model cycle, as shown in Figure 2.

Unit-Delay simulation on FPGAs offers several benefits:

- Modules can be decomposed and designed separately.
- There are no combinational paths between modules.
- No temporal deadlocks are possible.
- All modules follow a consistent “read, calculate, write” sequence which eases implementation.
- Temporal violations are easily avoided.

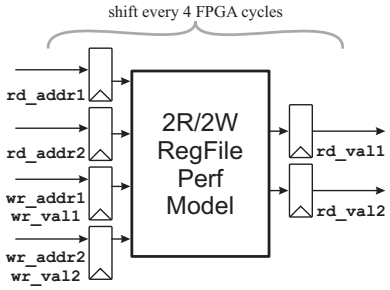


Figure 2: The performance model from Figure 1D in the unit-delay simulation scheme. The inputs and outputs are written every 4th FPGA cycle. The cycle-accurate state of the target circuit from Figure 1A may be recreated by observing the state only on every 4th FPGA cycle.

By these last two points we mean that all modules demonstrate consistent behavior. First they read all inputs, then compute for some number of FPGA cycles, and finally write all outputs. Temporal violations can occur only if a producer module writes a value prematurely before the end of the model cycle, where it could be observed by a consumer. All such violations may be avoided by restricting all producers to to write outputs only on the final FPGA cycle of a model cycle.

But what do our metrics have to say about the performance of this technique? As there are no combinational paths between modules, $frequency_{FPGA}$ will be determined by the modules themselves, which we expect would be good. But what about FMR? If the unit-delay simulator shifts to a new model cycle every n FPGA cycles then:

$$FMR_{unit\,delay} = n$$

Thus unit-delay simulation is appropriate for low values of n. In practice, however, the static worst case n is often quite large, though it applies to events that happen rarely. Moreover, unit-delay simulation cannot be used when n cannot be bounded – for example if the FPGA occasionally communicates with a host processor via a PCI connection. We conclude that although unit-delay simulation offers many benefits, it is unsuitable in a large number of practical situations.

3.2 Dynamic Barrier Synchronization

An alternative is to have all modules coordinate dynamically on when to move to the next model cycle. An FPGA performance model using dynamic barrier synchronization is shown in Figure 3. A centralized controller tracks model time, and alerts all modules when it is time to advance to the next model cycle. The modules then simulate, and report back when finished. When all modules have finished, the time counter is incremented, and the modules are alerted to proceed again. Simulation results may be observed on model cycle boundaries.

Dynamic barrier synchronization allows individual model cycles to use differing numbers of FPGA clock cycles, and removes the restriction that the worst case be bounded statically. The FMR of a barrier synchronization simulator with modules M is the mean of the FPGA cycles of the longest-running module for each simulated model cycle:

$$FMR_{barrier} = \frac{\sum_{i=0}^{numModelCC} max\left\{cycles_{FPGA}(i,m), m \in M\right\}}{numModelCC}$$

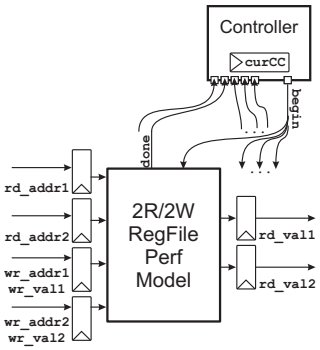


Figure 3: The performance model from Figure 1D placed in a dynamic barrier synchronization scheme. When each module asserts `done` the Controller broadcasts `begin`, at which point all modules read their inputs and proceed to the next model cycle. The cycle-accurate state of the target circuit may only be observed when the `begin` signal is asserted.

Note that this FMR is, in the worst case, equal to the FMR of the unit-delay case (when every model cycle takes exactly the same amount of time to simulate). In practice the FMR of dynamic barrier synchronization is often significantly better, as the dynamic worst case is often better than the static worst case.

To summarize, barrier synchronization on FPGAs retains the following benefits from the unit-delay scheme:

- Modules still can be decomposed and designed separately.
- Modules still follow the "read, calculate, write" abstraction.

One disadvantage of barrier synchronization compared to unit-delay is that modules may need some "shadow" state to hold stable copies of data, as they now cannot predict when the producer module will overwrite the data in the communication register. This means that temporal violations may be possible, by observing the data that the producer has written for cycle n+1 on model cycle n. (Temporal violations can be easily avoided if the module only observes the shadow state, which is initialized in the first FPGA cycle of every simulated cycle.) Additionally, the barrier synchronization simulation technique can experience a temporal deadlock if an individual module never terminates a clock cycle.

In practice, we have found all of these problems are minor, or have easy solutions. The main problem with barrier synchronization comes when one considers $frequency_{FPGA}$. The combinational signals to and from the controller can impose a large burden on the FPGA place and route tools. To assess this problem we devised an experiment, the results of which are shown in Figure 4. As it demonstrates, moving from 25 to 100 modules resulted in a decrease in clock frequency of 35%. We conclude that the dynamic barrier synchronization technique offers benefits over the unit-delay case, but also faces scaling issues which limit it to small numbers of modules.

3.3 FPGA Simulation Technique Summary

Using the metrics we introduced in Section 2, the strengths and weaknesses of these schemes can be summarized as:

	FMR	$Frequency_{FPGA}$
Unit-Delay	Fixed at worst case	Good
Barrier Sync	Good	Scales poorly

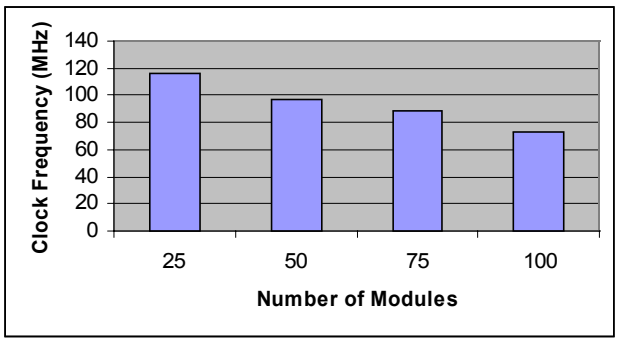


Figure 4: To test the scaling of dynamic barrier synchronization we created a simple module with a small amount of combinational logic, so that it would not affect the critical path. This module was then replicated n times in a strict linear hierarchy, so as not to impose any additional restrictions on the place-and-route tools. The modules were synthesized for the Xilinx VirtexIIPro 30 FPGA using Xilinx ISE 8.2i, and demonstrated a 39% loss of clock speed as a result of the centralized controller. In addition, we observed that the execution time of the FPGA place and route tools increased 20-fold over these same data points.

We conclude that unit-delay simulation is a good choice for models with small, bounded cycle simulation times, whereas dynamic-barrier synchronization is a good choice only for models which decompose into a small number of individual modules.

One approach would be to attempt to improve the clock frequency of the barrier simulation method, perhaps by pipelining the combinational AND-gate, or arranging the modules into a tree in order to ease the place-and-route requirements. But even if the *frequency*$_{FPGA}$ problem could be solved completely, the barrier synchronization approach still limits performance by forcing all modules to move in lockstep. In the next section we present A-Ports, an alternative which removes the centralized controller entirely, thus achieving good *frequency*$_{FPGA}$. In Section 5 we demonstrate that this scheme can achieve better FMR than dynamic barrier synchronization, with a demonstrated average improvement of 19% on our out-of-order processor model.

4. DISTRIBUTED PERFORMANCE MODELING VIA A-PORTS

4.1 Asim Ports in Software

The Intel software performance model Asim represents time via a mechanism known as ports. A model in Asim is decomposed into modules which have no inherent notion of time – conceptually they may be considered to be infinitely fast. All communication between modules must go through ports, which are essentially FIFOs of a message type t with a user-specified latency *l* and bandwidth *b*. The idea of using FIFOs to track simulation time is examined in Figure 5.

A message placed into a port on model cycle *n* will emerge on model cycle *n+l*. If a module does not write the port on model cycle *n*, then a special value NoMessage will emerge on cycle *n+l*, indicating no activity on this port. Ports with a latency of 0 are allowed, but may not be arranged into "combinational loops." An observer may determine how many cycles a module has simulated by counting the number of read/writes to its Ports.

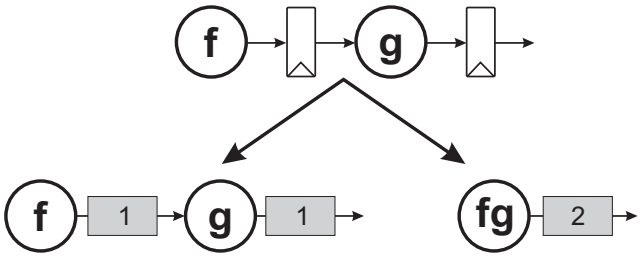


Figure 5: Modeling a 2-stage pipeline using Asim Ports. One choice is to decompose the system into two separate modules and connect it with ports of latency=1. It is often convenient to combine them and use a latency=2 port. Thus highly pipelined operations such as multipliers are more easily modeled because the functionality is retained in a single stage. Retiming the pipeline of the target may be evaluated by changing the latency of the port without changing the functionality of the modules.

In software the main benefit the Asim port abstraction is that it eases model development. The implementors of each module do not need to worry about explicitly representing model time at all, but simply to read and write in a FIFO manner. No temporal violations are possible because modules are not allowed to "peek" at the next values in the ports. Also, the simulator will never deadlock as long as each module takes a finite amount of time to simulate each model cycle (the same as barrier synchronization in Section 3.2).

More recently, there has been interest in using Asim ports for performance reasons: namely to control a highly-parallel software simulator. Currently, in software Asim parallel simulation is controlled by a centralized clock server which uses barrier synchronization to coordinate between a small number of threads (equal to the number of host cores the simulator is running on). Barr demonstrated that this centralized controller could be removed and simulation controlled by using the ports themselves [2]. Ultimately, though, assigning a thread per module would result in hundreds of threads which would overwhelm the available parallelism of today's 8-to-16 core servers. In software this was overcome by limiting the number of parallel threads, which also undoes much of the benefit compared to barrier synchronization. In contrast, an FPGA is fully able to take advantage of this level of parallelism.

4.2 Implementing A-Ports on FPGAs

We name our technique A-Ports, to distinguish it from prior work on Asim ports, and to emphasize the generality of the approach. The goal of the A-Ports implementation for FPGAs is to expose the highest degree of parallelism possible. To that end modules make a local decision about whether or not to proceed to the next model cycle by examining their local input and output ports. When a module does read its inputs to begin a cycle, it may take any number of FPGA cycles to simulate the model cycle. A module concludes by writing all of its output ports (Figure 6). All ports must be read/written every model cycle.

In software, Asim ports can be implemented with infinite buffering. On an FPGA buffer sizes must be limited. As shown in Figure 7, we implement an A-Port of message type *t* as a FIFO of *t*+1 bit-wide elements, the extra bit indicating NoMessage (in addition to the standard FIFO valid bits). Because buffer sizes are statically

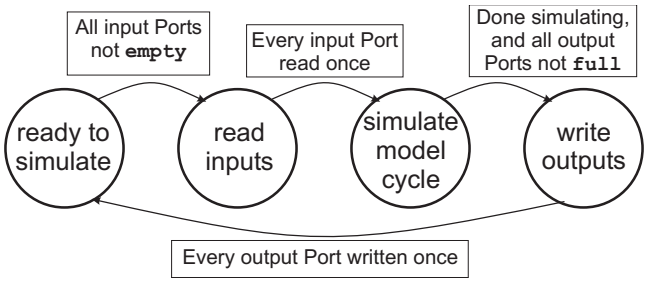


Figure 6: The protocol for modules to communicate via A-Ports. These states are only conceptual – a module may read, simulate, and produce output in the same FPGA cycle.

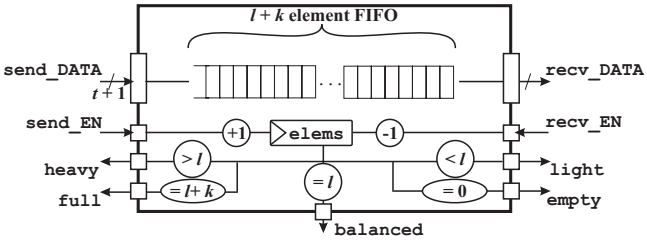


Figure 7: The A-Port hardware interface. The A-Port transmits messages of type *t* with latency *l*, and *k* extra buffering slots. An extra bit specifies Message or NoMessage. The Port pictured has bandwidth *b*=1. Increasing bandwidth increases the minimum buffer requirement, and the designer may elext to add additional enqueue/dequeue ports in an attempt to improve simulator throughput. A-Ports with large buffers can be implemented using FPGA RAM resources.

fixed, a simulator deadlock can be introduced if insufficient buffer space is provided. This deadlock can be avoided based on the following sufficient conditions:

- Each A-Port of latency *l* must contain at least *l*+1 buffering.
- Each A-Port of latency *l* and bandwidth *b* is initialized to contain *lb* copies of NoMessage at simulator startup.
- Modules should be arranged in a connected graph.

To see why, consider that when the simulator starts up every module will be able to simulate a cycle, unless they have a zero-latency input port. The "no combinational loops" requirement guarantees that any such modules are transitively connected to modules which have non-zero-latency inputs, and thus are able to simulate. Furthermore, note that by simulating a model cycle, a module can never disable other modules from simulating model cycles, but only enable them (though it may disable itself). Therefore there will always be one or more modules in the simulator which are able to proceed to the next model cycle.

4.3 Model Time in the A-Ports Scheme

Note that in the A-Ports scheme the representation of time is implicit. The modules and A-Ports do not need to keep a local counter which tracks the current model cycle – in fact a module does not need to know what cycle it is currently simulating. The removal of these counters represents a savings of FPGA register resources in systems with a large number of modules, and contributes to A-Ports' efficiency.

Not only do A-Ports not know what model cycle they are simulating, but frequently the producer and consumer modules attached

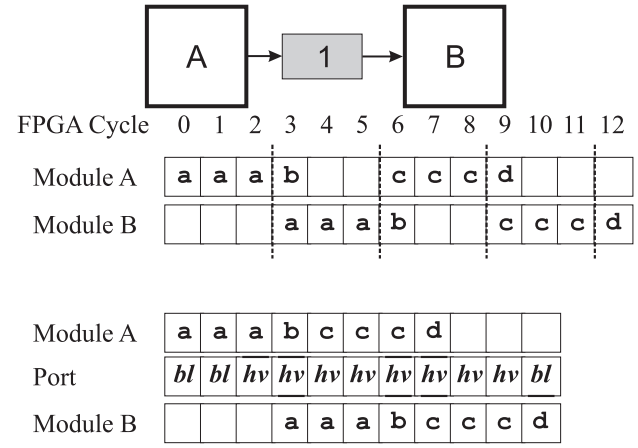


Figure 8: Modules A and B are shown simulating 4 model clock cycles a, b, c, and d. a and c require 3 FPGA cycles to simulate, whereas b and d require only 1. With barrier synchronization it takes 13 FPGA cycles to do the simulation. The distributed technique improves this to 11, as the A-Port moves between balanced (bl) and heavy (hv) states. Bold lines indicate A-Port sends and receives.

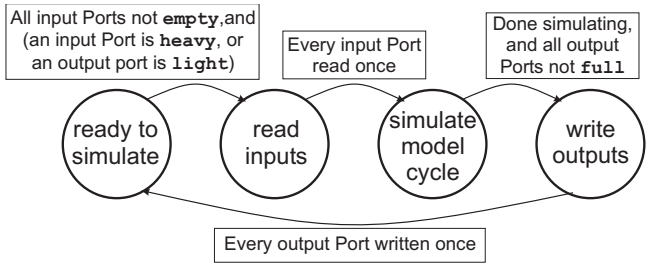


Figure 9: The protocol for resynchronizing decoupled modules to the same clock cycle. If all modules follow this protocol than eventually the system will quiesce.

via an A-Port may be simulating different model cycles (Figure 8). The producer may run into the future, pre-computing values until the buffer is full. Similarly, a fast consumer may "drain" the A-Port, allowing it to run into the future with respect to the producer. We say that simulation via A-Ports is *decoupled* because a module can "slip" ahead as long as its input data is available, and there is sufficient buffering to place its result. Similarly a module can fall behind if it is producing at a slower rate, or its input data is not available.

On any given FPGA cycle we say an A-Port of latency *l* is *balanced* when the FIFO contains exactly *l* elements. When an A-Port contains more than *l* elements it is *heavy*, and similarly it is *light* when it contains fewer than *l* elements. Observe:

- When an A-Port is balanced, the modules it connects are simulating the same model cycle.
- When an A-Port is heavy, the producer module is simulating into the future compared to the receiving module.
- When an A-Port is light, the situation is reversed.

The amount of time that adjacent modules can "slip" in time is limited by the buffering available. The consumer module of an *l*-latency A-Port can run ahead at most *l* model clock cycles before draining the buffer. A producer writing into an A-Port with *k* extra buffering can only proceed *k* cycles ahead before filling the buffer.

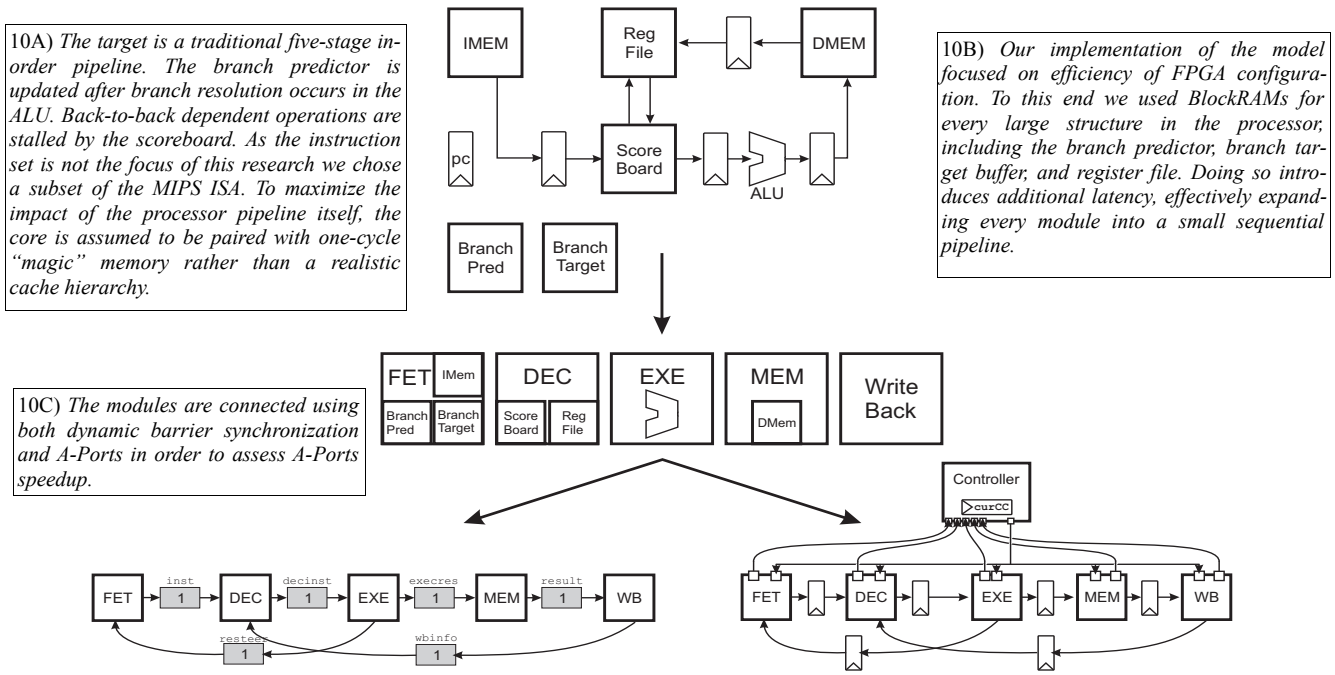


Figure 10: Modeling a 5-stage in-order pipeline

Selecting the appropriate buffer sizes can have a significant impact on simulator performance, as we will show in Section 5. However, as long as the minimum requirements to prevent deadlock presented above are followed, these optimizations can be performed without changing the results of simulation, or inadvertently introducing bugs into the simulator.

4.4. Resynchronizing Decoupled Models

Off-FPGA storage mechanisms such as flash memory, PCI buses, or Ethernet typically have throughputs much lower than on-chip communication. Thus the process of recording the results of simulation can itself become the simulator performance bottleneck, as useful work must stall until the buffers drain.

One solution to this problem is to perform sampling. We disable result recording and allow the simulator to run in "fast mode" without having to worry about filling up network buffers. Result recording begins upon a user-specified trigger, such as simulating a specified number of model cycles or entering a critical section.

But what if the decoupled modules have slipped in time as a result of the A-Ports? In that case the modules could be simulating different model cycles and the result recording may become confused. As we are using an implicit notion of time with no local counters, the modules cannot know which exact model cycle they are simulating.

The solution to this problem is to rebalance the decoupled modules before enabling the result capture. To resynchronize the system, modules enter a mode where they use the following protocol:

- If any output A-Ports are `light`, or any input A-Ports are `heavy`, simulate the next model cycle (assuming all input A-Ports are not `empty`).

If all modules follow this protocol (shown in Figure 9), the system will eventually quiesce. At the point of quiescence every A-Port will be balanced, and thus every module will be on the same model clock cycle. To see why, consider that at any given FPGA cycle there will be a non-empty set of modules which are furthest ahead in model cycles. These modules will, by definition, have no light outputs or heavy inputs, and therefore will not move forward. Any incoming ports to this group must be light and any outgoing ports must be heavy. Therefore the modules which are connected to these ports will attempt to simulate the next model cycle. The only reason they would not be able to proceed would be if they did not have all of their inputs ready. Yet somewhere in the system there must be a non-empty set of modules which is farthest behind in time, and thus able to simulate the next cycle. If the graph is connected, any module which can simulate will only make progress towards increasing the set of modules farthest ahead in time. Eventually this set will include every module, every port will be balanced, and the system will not proceed.

When the simulator quiesces, it is straightforward to add a mode where the simulator can step forward one model cycle at a time. This stepping mode can be useful for debugging or for real-time interaction between the user and the simulator.

5. QUANTITATIVE ASSESSMENT

In order to assess the A-Ports scheme we identified two target circuits. First, a traditional five-stage in-order microprocessor pipeline, explained in detail in Figure 10. Second, a more realistic out-of-order superscalar processor, as shown in Figure 11. The designs were implemented using Bluespec SystemVerilog [3], and were synthesized for a Xilinx Virtex IIPro platform and assessed for simulation speed and efficiency. The results, shown in Figure 12, show that the out-of-order processor is clearly a better architecture. As a processor, it would execute benchmarks substantially faster (assuming the circuit design team was able to achieve an equivalent clock speed).

These results represent the insights into the target design that most users of performance models care about. However, as simulator architects, we are also interested in comparative simulator performance, as shown in Figure 13. These results demonstrate that when we consider simulator performance the situation is reversed

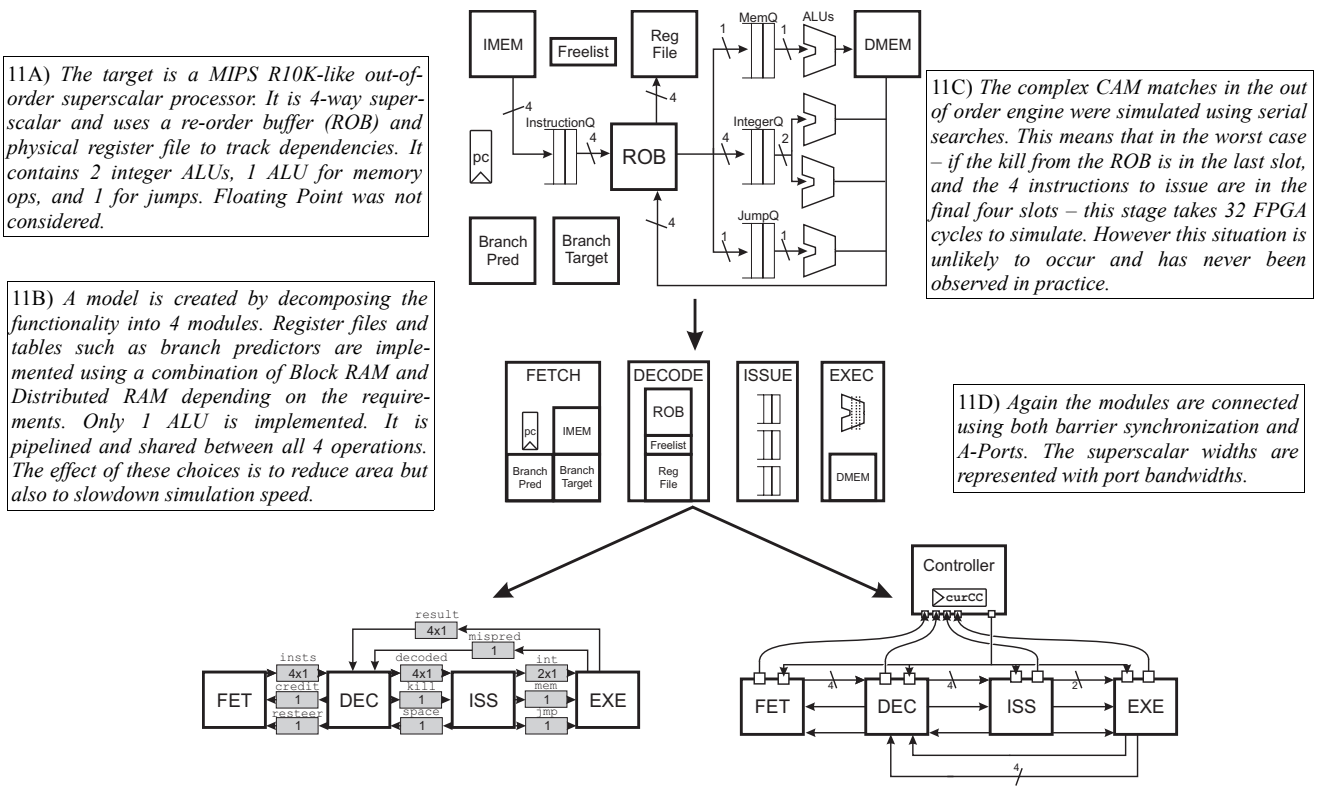


Figure 11: Modeling an out-of-order processor.

– the five-stage simulator can simulate model clocks more than twice as fast (14 MHz vs 6 MHz), due to the multiple instructions which the out-of-order superscalar model executes during every model cycle. However when we consider simulated Instructions per Second, as defined in Section 2.2, the situation is more balanced (5.1 vs 4.7 MIPS). This metric correctly compensates for the difference in target CPI – remaining differences are due to the overhead of simulating out-of-order execution.

Beyond comparing the modeled processors against each other, we also compared them to barrier synchronization versions. Figure 13 shows that the 5-stage simulator using A-Ports is an average of 23% faster versus barrier synchronization. For the out-of-order model, the situation is more complicated. Using the minimum buffer sizes results in a 4% improvement versus barrier synchronization. However, as we noted in Section 4.3, the A-Ports buffer size limits the amount adjacent modules can slip in model time. Figure 13 demonstrates that increasing the amount of buffering results in a significant performance improvement for the out-of-order model, allowing it to achieve a simulation rate 19% faster than barrier synchronization. In contrast, increasing the buffer sizes does not result in any further improvement for the 5-stage pipeline. This is because the modules in the 5-stage pipeline are more evenly balanced, and thus do not slip with respect to each other as frequently for our benchmarks.

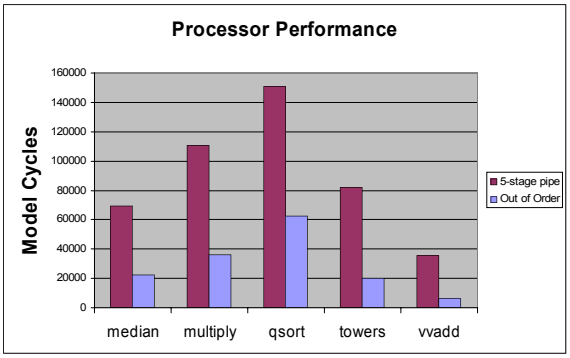
6. RELATED WORK

Early efforts at creating performance models on FPGAs shared the goal of creating a model early in the design process [18, 20], but these efforts used the FPGA clock itself as the simulation clock, at the cost of accuracy. Thus these are more closely aligned with what we have termed an emulator.

An alternative to reimplementing the entire performance model onto the FPGA is maintaining a software simulator and accelerating critical tasks in hardware. Penry et al. [16] explored using the Power PCs on Xilinx Virtex IIPro FPGAs to accelerate the software Liberty Simulation Environment. Logic was configured into the FPGA fabric that allowed Liberty to track the number of clock cycles a task took. Thus all model timing was equivalent to FPGA timings.

The approach of taking many FPGA cycles to simulate one model cycle was popularized by the RAMP project [1, 19]. RAMP aims to model systems with hundreds of chips in them by spreading them across multiple FPGAs, and across multiple boards. Ramp Description Language [12], or RDL, allows the model-builder to create "channels" between units. These channels have FIFO semantics with user-specifiable model time latency and bandwidth, similar to the A-Ports presented here. However the focus of RAMP channels is different, in that they are meant to connect large units, such as processor cores which may even be on different FPGAs. Hence RAMP channels use a credit-based protocol appropriate for connecting large blocks. In contrast, A-Ports due not force the designer to use blocks which interact with a credit-based protocol, as they are meant to connect much smaller blocks on the level of pipeline stages. We note that a RAMP channel could be implemented using two A-Ports, one flowing from producer to consumer with the data, the other flowing in the reverse with the credit.

Chiou's UT-FAST is a hybrid hardware-software performance model which uses a software functional emulator to drive an FPGA which adds timing information to the instruction stream [7, 8]. UT-FAST originally used FPGA registers to add timing information to the instruction stream, with a one-to-one correspondence between FPGA cycles and model cycles. Collaboration between the HAsim project and UT-FAST resulted in UT-FAST developing a more generalized connector which was also inspired



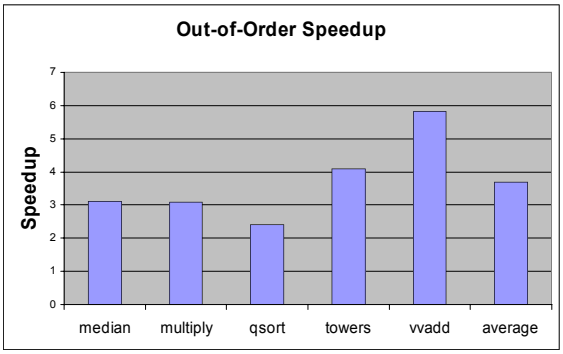
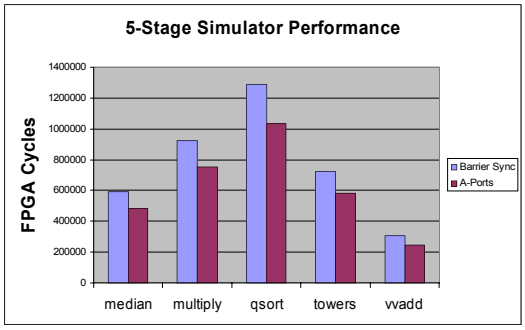
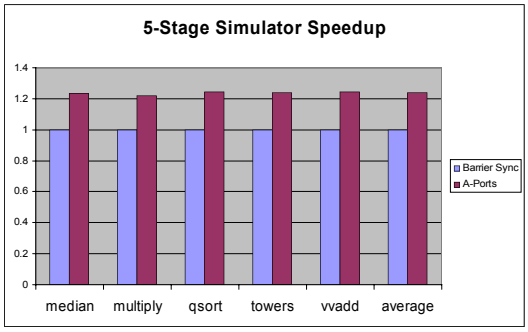
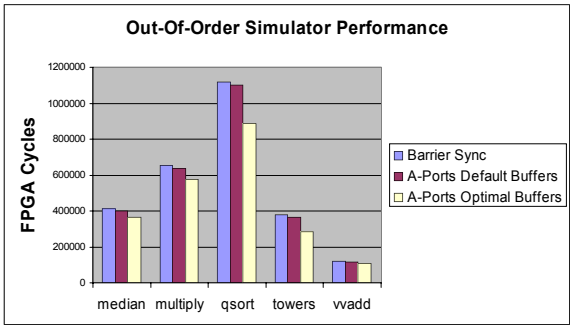
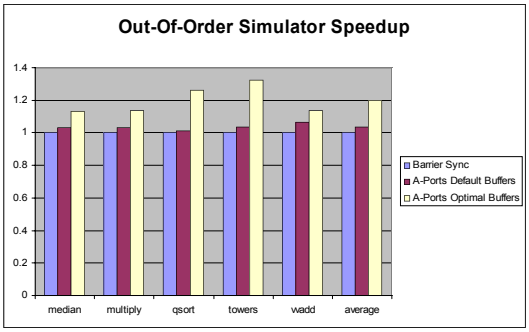


Figure 12: Assessing performance of the target processors. We measured our 5-stage pipeline and out-of-order models running small benchmarks: numeric median and multiplication, quick sort, Towers of Hanoi, and vector-vector addition. While we acknowledge the limitations of trying to draw conclusions from small benchmarks running on processors not paired with a realistic memory hierarchy, the results showed the out-of-order processor performing between 2.4 and 5.8 times faster than the 5-stage pipeline, depending on the amount of instruction-level parallelism available in the benchmark.









	5-stage A-Ports	Out of Order A-Ports
FPGA Slices	9220	22,873
Block RAMs	25	25
Clock Speed	96.9 MHz	95.0 MHz
Average FMR	6.90	15.6
Simulation Rate	14 MHz	6 MHz
Average Simulator IPS	5.1 MIPS	4.7 MIPS

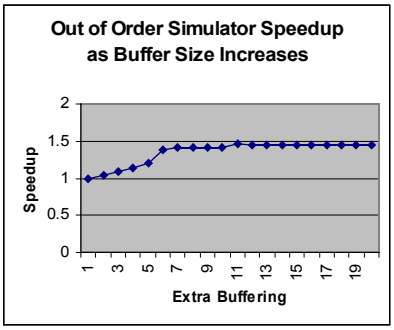


Figure 13: Assessing the performance of the models, rather than the targets they are simulating. The simulator was synthesized onto a Virtex II platform using Xilinx ISE 8.2i.Using A-Ports with minimum buffering results in an average improvement of simulation speed of 24% on the 5-stage pipeline and 4% on the Out-of-Order simulator. Increasing the buffering to their optimal sizes (Section 5) improved this to 19.6%. While the serial searches and reuse of the ALU reduces FMR of the Out-of-Order simulator, but the simulated IPS of the two simulators is about the same. This is because the OOO model executes so many more instructions per simulated clock.

by Asim ports, as presented in [7]. Storage within this connector also represents model-level storage and queues, which means that there is less of a clear division between simulator implementation and model timing. The contribution of the A-Ports methodology presented here is to cleanly decouple model-level timing and simulation, simplify the synchronization protocol, lower the overhead by removing registers to explicitly track model time, and present a means to resynchronize the system to the same model cycle.

There is a parallel between clock simulation using A-Ports and Carloni's theory of latency-insensitive design [5]. The main difference is that Carloni's technique is based on small combinational circuits connected by latency-insensitive wrappers. In contrast, in the A-Ports scheme modules are larger entities which may have local state, and thus may take many physical FPGA cycles to perform their computation. This allows the designer to tradeoff simulation speed and module resources, as described in Section 2.2. In this respect A-Ports is close in inspiration to Lee's synchronous dataflow abstraction [13].

7. DISCUSSION

In this paper we explored FPGAs as a platform for executing cycle-accurate performance models. Some metrics were developed to quantitatively evaluate their performance. We used these metrics to gain insight into the strengths and weaknesses of existing schemes for synchronous simulation. This paper introduced A-Ports and explored how the ability of adjacent modules to be simultaneously simulating different model cycles can lead to a performance improvement. Finally, we implemented two models and demonstrated an average improvement in simulation rate of 19% for our out-of-order model given appropriately sized buffers.

In the future we hope to extend the technique to efficiently handle modeling multiple clock domains. Additionally we hope to use the multiple physical clock domains on the FPGA to allow adjacent modules to run in separate FPGA clock domains. The goal of the HAsim project [9, 15] is to use A-Ports, combined with other techniques from software performance models [15], to create a high-detail model of a chip-multiprocessor (CMP) on an FPGA.

ACKNOWLEDGMENTS

The authors would like to acknowledge the help and feedback of Angshuman Parashar, Nirav Dave, Derek Chiou, and the anonymous reviewers.

BIBLIOGRAPHY

[1] Arvind, K. Asanovic, D. Chiou, J. C. Hoe, C. Kozyrakis, S.L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek. RAMP: Research Accelerator for Multiple Processors - A Community Vision for a Shared Experimental Parallel HW/SW Platform. Technical Report.

[2] K.C. Barr, R. Matas-Navarro, C. Weaver, T. Juan, and J. Emer. Simulating a Chip Multiprocessor with a Symmetric Multiprocessor. In *Boston Area Archictecture Workshop (BARC)*, January 2005.

[3] Bluespec, Inc. Bluespec Language Reference Manual.

[4] R. Bryant. Simulation on a Distributed System. In *Proceedings of the First International Conference on Distributed Systems*, July 1979.

[5] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, September 2001.

[6] K. M. Chandy and J. Misra. Asynchronous Parallel Simulation via a Sequence of Parallel Computations. In *Communications of the ACM*, pp. 198-206, April 1981.

[7] 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 International Conference on Computer-Aided Design (ICCAD)*, 2007.

[8] 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 MICRO*, 2007.

[9] N. Dave, M. Pellauer, J. Emer, and Arvind. Implementing a Functional/Timing Partitioned Microprocessor Simulator with an FPGA. In the *Proceedings of the Second Workshop on Architecture Research using FPGA Platforms (WARFP)*, February 2006.

[10] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A Performance Model Framework. *Computer*, pp. 68-76, February 2002.

[11] D. Dalton, V. Bessler, J. Griffiths, A. McCarthy, A. Vadher, R. O'Kane, R. Quigley and D. O'Connor. APPLES: A Full Gate-Timing FPGA-Based Hardware Simulator. *Field-Programmable Logic and Applications*, September 2003.

[12] G. Gibeling, A. Schultz, and K. Asanovic. The RAMP Architecture & Description Language. Technical Report.

[13] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, January 1987.

[14] OSCI. SystemC Language Reference Manual version 2.1

[15] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. Quick Performance Models Quickly: Timing-Directed Simulation on FPGAs. To appear in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2008.

[16] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August and D. Connors. Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-processors. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2006

[17] G. Pfister. The Yorktown Simulation Engine. *In Proceedings of 19th Conference on Design Automation (DAC)*, 1982.

[18] J. Ray and J.C. Hoe. High-Level Modeling and FPGA Prototyping of Microprocessors. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, pages 100-107, 2003.

[19] J. Wawrzynek, D. Patterson, M. Oskin, S. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: A Research Accelerator for Multiple Processors, *IEEE Micro* Mar/Apr 2007, pp. 46-57.

[20] R. Wunderlich and J. C. Hoe. In-System FPGA Prototyping of an Itanium Microarchitecture. In *Proceedings of International Conference on Computer Design (ICCD)*, October 2004.