A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs

MICHAEL PELLAUER and MURALIDARAN VIJAYARAGHAVAN
Massachusetts Institute of Technology
MICHAEL ADLER
Intel Corporation
ARVIND
Massachusetts Institute of Technology
and
JOEL EMER
Massachusetts Institute of Technology and Intel Corporation

Computer architects need to run cycle-accurate performance models of processors orders of magnitude faster. We discuss why the speedup on traditional multicores is limited, and why FPGAs represent a good vehicle to achieve a dramatic performance improvement over software models. This article introduces A-Port Networks, a simulation scheme designed to expose the fine-grained parallelism inherent in performance models and efficiently exploit them using FPGAs.

Categories and Subject Descriptors: C.4 [**Computer-Communication Networks**]: Performance of Systems—*Modeling techniques*

General Terms: Measurement, Performance, Design

Additional Key Words and Phrases: FPGA, performance models, simulation, emulation

ACM Reference Format:
Pellauer, M., Vijayaraghavan, M., Adler, M., Arvind, and Emer, J. 2009. A-port networks: Preserving the timed behavior of synchronous systems for modeling on FPGAs. ACM Trans. Reconfig. Techn. Syst. 2, 3, Article 16 (September 2009), 26 pages. DOI = 10.1145/1575774.1575775.
http://doi.acm.org/10.1145/1575774.1575775.

This work was funded by a grant from Intel and by NSF grant CCF-0541164.
Author's address: M. Pellauer; email: Pellauer@csail.mit.edu.
Collaboration with the RAMP project was made possible by NSF grant CNS0551739. The authors would like to acknowledge the collaboration and valuably feedback of Angshuman Parashar, Zhihong Yu, Tao Wang, and Guan-Yi Sun.

1. INTRODUCTION

The processor design flow begins when the architect is given a set of requirements—for example, a high-performance out-of-order x86 processor, or a low-power in-order ARM processor. The architect then uses intuition and knowledge of existing systems in order to identify an initial target architecture. This intuition must be backed up by detailed quantitative studies on representative inputs before the architecture is finalized. This process is iterative, as each study leads to tweaking critical architecture parameters.

Consider the MIPS R10K-like target processor shown in Figure 1. We use this processor as an ongoing example throughout this paper. This is a 4-way superscalar processor, meaning that it can fetch and decode up to 4 instructions every clock cycle. It uses out-of-order issue logic, meaning that if the head of the instruction stream is stalled the processor can examine younger instructions to find independent operations to issue. It has 4 execution units of varying capabilities, and thus can issue up to 4 instructions per cycle under ideal circumstances. To support this the register file has 7 read ports and 4 write ports (the Jump Unit only requires one read port). Once this initial architecture is identified the architect would like to study the effect of various parameters such as branch predictor schemes, ALU pipeline depths, and ROB sizes.

Early in the design process these studies are usually not concerned with the amount of circuit area these various choices would require, nor the final clock frequency they could achieve, beyond basic ballpark estimates. Instead the architect is primarily concerned with studying the dynamic performance of the system as measured in clock cycles—thus these simulators are called *performance models*. Typical duties of performance models include tracking statistics via counters and generating cycle-by-cycle traces of the system operating on critical input segments.[1]

The most successful performance models:

—Are accurate enough to give architects confidence in their decisions.

—Are easy to design and modify, allowing for exploration of a range of options.

—Simulate fast enough to allow a wide range of inputs and dynamic situations to be studied in a reasonable amount of time.

Currently design teams write most such models in software, using home-brewed C/C++ simulators or frameworks such as SystemC. This eases model development, but the simulation speed of software models has not been able to keep pace with increasing complexity of modern processors. 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 I shows an overview of simulation speeds of performance models around Intel:

[1] We note that it is increasingly common to combine performance models with detailed estimates of a system's power consumption and exposure to dynamic soft errors, as these are closely tied to cycle-by-cycle behavior.

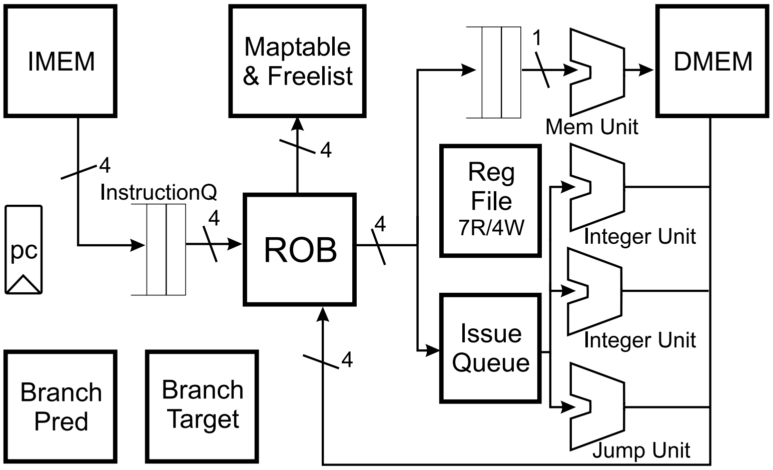


Fig. 1. Example out-of-order superscalar processor target.

Table I. Simulation Speeds

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

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—yet these gates typically require multiple host instructions to simulate. 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, intuition tells us that FPGAs should represent a better platform for efficient execution of performance models. Contemporary efforts to explore FPGAs as a platform for performance modeling include Penry et al.'s [2006] accelerators for the Liberty simulator, UT-FAST [Chiou et al. 2007a; 2007b] which uses the FPGA as a timing model connected to a software functional simulator, and our HAsim project [Pellauer et al. 2008a; 2008b] which aims to create a variant of the Intel Asim simulation environment [Emer et al. 2002] on an FPGA. The goals of the RAMP project also include serving as a platform for the execution of accurate performance models [Arvind et al. 2006; Wawrzynek et al. 2007].

The key insight all of these projects share 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. The main challenge then becomes tracking the simulated *model clock cycle* in a distributed way that exposes sufficient fine-grain parallelism for the FPGA to exploit.

In this article we present A-Port Networks, an adaption of techniques from the Asim simulator designed to perform efficient cycle-accurate simulation on highly parallel substrates such as FPGAs. We give a taxonomy of existing distributed simulation techniques and explore their strengths and weaknesses on FPGAs. We give an implementation of A-Ports Networks for FPGAs and discuss why it addresses these weaknesses. We demonstrate a performance improvement of 19% using A-Ports to simulate our processor over dynamic barrier synchronization.

We limit the discussion to models of synchronous digital systems—asynchronous or analog systems are not considered. Although we use general-purpose processors as an ongoing example, none of the techniques presented are microprocessor-specific. Extending the A-Ports technique to simulate multiple clock domains or globally asynchronous locally synchronous (GALS) systems is left to future work.

2. BACKGROUND: PERFORMANCE MODELS IN ASIM

The problem of creating a performance model for a synchronous system can be generalized to the *dynamic snapshot* problem:

—Given a model in state *s* and input *i*, what is the relevant state of the model at time *t*?

By *relevant state* we mean the state elements which the architect observes in order to determine the performance of the system. For example, in the processor in Figure 1 the architect may decide that the internal pipeline registers of the execution units are irrelevant, while the result output by the ALU is relevant. This is similar to the difference between architectural state and microarchitectural state, though in many cases the distinction is not so cut-and-dry.

Intel's Asim [Emer et al. 2002] is a framework for creating performance models. Asim's main goal is to allow architects to develop performance models quickly by reusing existing pieces. To encourage this, the target system is decomposed into individual modules (branch predictors, caches, etc.) that can be swapped for variations in a plug-and-play manner. In order for this swapping to be successful, practice has shown that the modules must have a clear and well-documented interface as well as an explicit and easy-to-change indication of the time the computation takes. To this end, Asim has developed a formalism known as *ports*, which formalizes the interface and helps separate concerns of timing from functionality.

2.1 Asim Ports

In Asim, individual modules are arranged into a directed graph connected by *ports*, communication channels annotated with a user-specified latency *l*. The modules themselves have no inherent notion of time—we can consider their

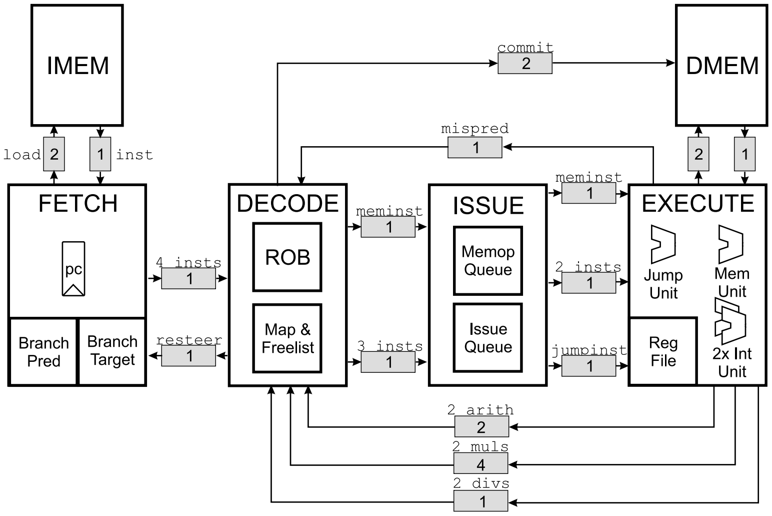


Fig. 2. Target processor as a port-based model.

computation to be infinitely fast. Time is represented only in the delay of communication between modules. Ports of latency zero are allowed, but may not be arranged into “combinational loops”—a familiar restriction to hardware designers. Each port has a single writer and reader, and all communication between modules goes between ports. Latencies are statically specified and may not change dynamically.

Our target processor is recast as a port-based model in Figure 2. The system has been partitioned into modules using the pipeline stages as a general guideline. Pipeline registers were replaced ports of latency 1, such as those connecting Fetch and Decode. The instruction- and data-memories are represented as simple static latencies, which is unrealistic but illustrative for the purposes of this paper. The latencies associated with the ALU operations are more complex, and require a greater explanation of port semantics.

The interface for sending a message into a port is as follows:

```
Send(<msg_type> data, int current_time);
```

Because a producer may not have sent a message, the interface for receiving is:

```
bool Receive(int current_time, <msg_type>& data_out);
```

The `Receive` method returns true when the port has a message at that cycle, which is written into the `data_out` parameter. Each module then defines a `clock` method which represents simulating a single model cycle:

```
clock(int current_time);
```

In general, this method queries the module’s input ports to determine if they contain any messages. The module then performs all necessary computations and local state updates. It may also place messages into its output

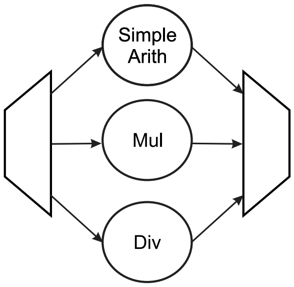


Fig. 3. Requirements for the processor's ALU.

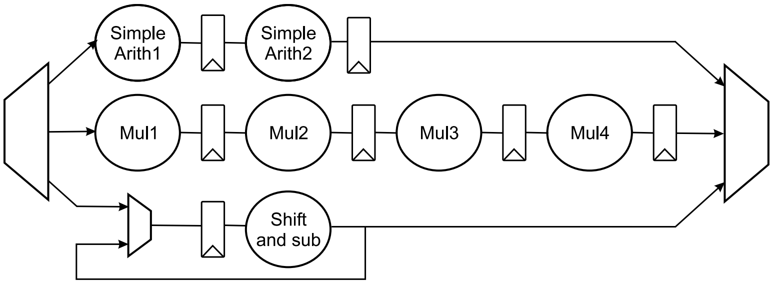


Fig. 4. A potential target ALU.

ports. The port uses its latency l to record that the message will appear on cycle `current_time`+l.

Now let us return to our example processor's Integer Unit. The ALU datapath has the general requirements shown in Figure 3—it must be able to perform simple arithmetic operations, multiplies, and divides. The architect wishes to explore the effect of various pipeline depths on overall system throughput. One potential target is shown in Figure 4, which uses a 2-stage pipeline for the simple operations and a 4-stage pipeline for the multiplier. Because the architect expects that divide operations are rare, she is considering implementing them with a circular shift-and-subtract. (The issue stage must know not to place more than one divide instruction in flight simultaneously.)

A port-based model of this ALU is shown in Figure 5. As it demonstrates, performing the calculation of the operations themselves is separated from the timing they require. As the arithmetic and multiply operations are systolic pipelines they are represented by performing the calculation, then placing the result into ports of latency 2 and 4, respectively.[2] The circular divider pipeline is represented differently—the output port is latency 1 and is paired with a counter. The integer unit determines that a divide should take the target n

[2] This bears some similarity to circuit designers altering the placement of pipeline registers late in the design flow. This technique is generally referred to as *retiming*, because moving combinational logic past registers can be used to change the delay of the critical path. Interestingly, from a modeling perspective "retiming" is not a good name, as the intent of this transformation is to preserve the behavior of the the target system with respect to the model clock.

```
clock(int cur)
    if (in_port.Receive(cur, &inst))
       VAL res = Execute(inst);
       switch(inst)
           ARITH: arith_port.send(res, cur);
           MUL:   mul_port.send(res, cur);
           DIV:   assert(!dividing);
                  dividing = true;
                  div_time = DivTime(inst);
                  div_res = res;
    if (dividing)
         if (--div_time == 0)
             div_port.send(div_res);
```

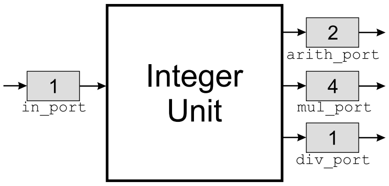


Fig. 5. Modeling the ALU with ports.

model cycles to calculate the result, and then places the result into the port $n-1$ cycles later. If the issue stage accidentally issues a new division while the circular pipeline is busy, an assertion fails.

Based on this interface our integer unit module can be replicated twice and plugged into the example processor from Figure 2. In general we have found port-based modeling to provide the following benefits:

—Encourages reuse by formalizing the module interface and separating timing concerns from functionality.

—Enables the architect to easily conduct a certain class of design exploration—playing "what if" games by changing the latencies of ports and observing the effects on system behavior.

—Eases model development because each module follows a similar "read, calculate, write" pattern.

—Allows a controller to coordinate simulation, as we shall discuss.

2.2 Sequential Simulation in Software

Sequential simulation in software Asim is coordinated by a centralized controller, which tracks the current model clock cycle and decides which module should execute next. The general simulation algorithm is as follows:

```
modelcycle = 0;
moduleQ = sort(modules);
while (1)
    foreach m in moduleQ
         m.clock(modelcycle);
    modelcycle++;
```

Note that if the model does not contain zero-latency ports, then the sorting step can be avoided. Zero-latency ports represent a causal dependence between the producer and consumer, implying that one must be simulated before the other. The controller determines a simulation order by performing a topological sort of the modules. (Cycles in the module graph can be cut at any nonzero-latency port for the purposes of determining simulation order. Such a port is guaranteed to exist because of the "no combinational loops" restriction.) As port latencies are static, this sort only needs to be performed on simulator startup.

2.3 Parallel Simulation in Software

Modules which are not connected by such a causal dependence may be simulated in parallel during each cycle in order to improve simulation rate. In parallel Asim the centralized clock server runs in a thread, and uses barrier synchronization to coordinate between a small number of simulation threads (linearly related to the number of host cores on which the simulator is running). Because of the causal relationship imposed by zero-latency ports, best performance is achieved when the model is partitioned in such a way that closely coupled modules are executed by the same thread. Each thread is given a set of modules to simulate, and stalls on a barrier when complete:

```
modelcycle = 0;
threads = partition(sort(modules));
while (1)
    foreach t in threads
         t.clockAll(modelcycle);
    wait_for_barrier();
    modelcycle++;
```

Barr et al. [2005] demonstrated that this centralized controller could be removed and simulation controlled by using certain "SMP" ports, where the producer and consumer would be in different threads. Since each module knows the explicit model cycle, a consumer could "peer backward" through incoming ports to determine when it was safe to proceed with simulation. The controllerless simulation for each thread became:

```
modelcycle = 0;
while (1)
   if (in_port.ProducerHasSimulated(modelcycle - in_port.latency))
       foreach m in modules
            m.clock(modelcycle);
       modelcycle++;
```

As this demonstrates, each thread was still responsible for sequentially simulating a number of modules. This was because 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. Unfortunately, limiting the number of parallel threads also undid much of the benefit compared to barrier synchronization. In contrast, an FPGA is fully able to take advantage of this level of parallelism.

3. EXISTING SIMULATION TECHNIQUES ON FPGAS

In this section we discuss various existing simulation techniques with the goal of exposing as much parallelism as possible in Asim-like port-based systems

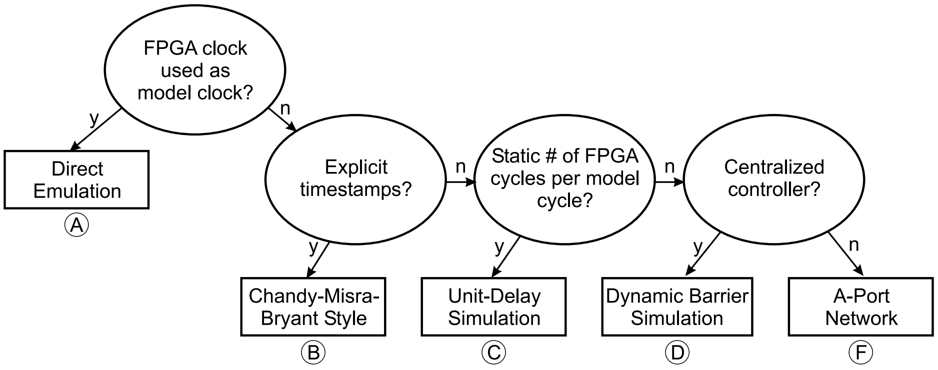

FPGA clock used as model clock?
y
n
Direct Emulation
A
Explicit timestamps?
n
Static # of FPGA cycles per model cycle?
n
Centralized controller?
y
y
y
n
Chandy-Misra-Bryant Style
B
Unit-Delay Simulation
C
Dynamic Barrier Simulation
D
A-Port Network
F

Fig. 6. Overview of simulation techniques for FPGAs.

on FPGAs. We compare these techniques to each other in Figure 6 and refer to this figure throughout this section.

3.1 The Emulation Approach

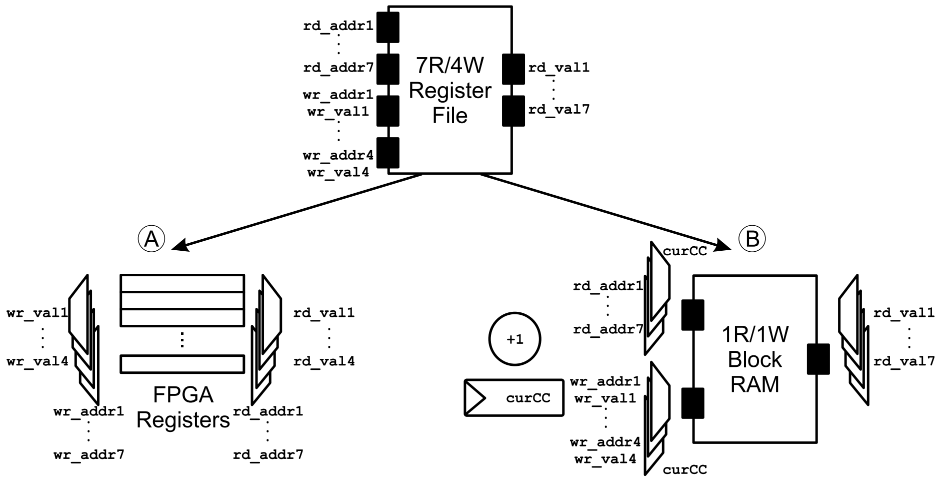
The first approach we consider is to use the FPGA clock to represent the model clock directly. In such a system running the model for *t* clock cycles would simply require ticking the physical FPGA clock *t* times. We refer to this approach as *direct emulation*, Node A in Figure 6.

The main problem with the emulation approach is that it requires each module in the system to complete all of its work in a single FPGA clock cycle. If the target ASIC employs structures that do not map well onto FPGAs (e.g., multiported register files, or content-addressable memories) then the resulting FPGA clock period is likely to be poor, slowing the rate of simulation. For example, consider the register file of our target processor. As stated above, this register file requires 7 read ports and 4 write ports. Implementing this on an FPGA directly would be very expensive, as shown in Figure 7, design A.

A better approach is to disassociate the FPGA clock cycle from the *model clock cycle*—a *simulation* rather than an emulation, in our terminology. Thus we may replace the register file with a space-efficient FPGA structure, a synchronous BlockRAM with one read port and one write port. Now we use 7 FPGA cycles to simulate the behavior of the target register file, as shown in Figure 7, design B, which can represent a significant savings. (We can overlap the writes with the reads because we have higher-level knowledge that the addresses are guaranteed to be distinct within one model cycle.)

3.2 Analyzing Simulation Approaches

While separating the model clock from the FPGA clock can save area, its effect on performance is less clear. While it can increase frequency, we must also take into account the number of FPGA cycles required to simulate a model cycle, which we call the FPGA-cycle to Model cycle Ratio (FMR). FMR is similar to the microprocessor performance metric Cycles Per Instruction (CPI) in that one can observe the FMR of a run, a region, or a particular class of instructions



	LUTs	BlockRAMs	Delay (ns)	FMR
Direct Emulation	6990	0	8.6	1
Simulation	160	1	3.4	7

Fig. 7. FPGA resources can be saved by simulating the target register file.

in order to gain insight into simulator performance. The FMR of a simulator combined with its FPGA clock rate gives us simulation rate:

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

The simulation approach is only useful if the gains to $frequency_{FPGA}$ are not offset by a large FMR. In practice we find that simulator Hz is not the best metric to measure performance models of processors on FPGAs. This is because models often require fewer cycles to simulate pipeline bubbles than heavy activity, and thus these idle cycles lower FMR. A better metric is to evaluate simulators on their simulated Instructions Per Second (IPS). For a software simulator this is calculated as:

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

Plugging in our above equation gives us the means to calculate the IPS of an FPGA performance model:

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

In addition to improving performance, we must ensure that the simulation approach does not introduce any *temporal violations*. Such a violation occurs when a value from model cycle $n + k$ is accidentally used to calculate a value on model cycle n. In highly parallel environments such as FPGAs, this 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*.[3]

The goal of a distributed simulation technique is to maximize simulator IPS while avoiding temporal violations and minimizing the overhead in terms of FPGA resource utilization. Classically, techniques fall into two broad categories: those which track time explicitly (also called "event-driven" simulation) and those that track time implicitly (also called "continuous" simulation).

3.3 Simulation with Explicit Timekeeping

Distributed simulation techniques that explicitly carry time are variants of the Chandy-Misra-Bryant simulation technique [Chandy and Misra 1981; Bryant 1979], Node B in Figure 6. In such schemes all data in the system is associated with a timestamp. Operations on data also increment the timestamp by the appropriate amount.

Any FPGA-optimized circuit may be used to perform the operations—the number of FPGA cycles that such a circuit requires to compute will have no impact on the results of simulation, but only the FMR of the simulator. Additionally, this scheme enables playing "what if" games with the simulated timings without substantial code changes.

The main benefit of explicit-time schemes is that model cycles with no activity do not need to be simulated explicitly. For example, on FPGA clock cycle 300 we may be simulating model time t, but by adding 1000 to the timestamp we would be simulating time $t + 1000$ on FPGA cycle 301. This is why such simulation schemes are referred to as "event-driven," as idle model cycles are passed over until an event occurs.

The disadvantage of such techniques is the overhead of explicitly storing, transmitting, and manipulating timestamps. Practice has shown that performance models—which simulate the core pipelines of synchronous systems—do not generally demonstrate enough idle areas of the system to compensate for this overhead. It is significant to note that the major performance models written in software use continuous simulation techniques rather than event-driven techniques.

3.4 Simulation with Implicit Timekeeping

Continuous simulation techniques make use of the fact that the target system is a synchronous system with only a single (or a small number of) distinct clock domains. These techniques are able to make the timekeeping implicit, using the coordination of behavior among the simulated modules to simulate the target clock.

One straightforward way to coordinate distributed modules is to assign each module n FPGA cycles to simulate one model cycle. This is *unit-delay*

[3] Note that this is distinct from a model-level deadlock, which results when the target design is faulty. If the target enters a deadlocked state, then the performance model should correctly simulate the machine remaining in that state as model time continues to advance.

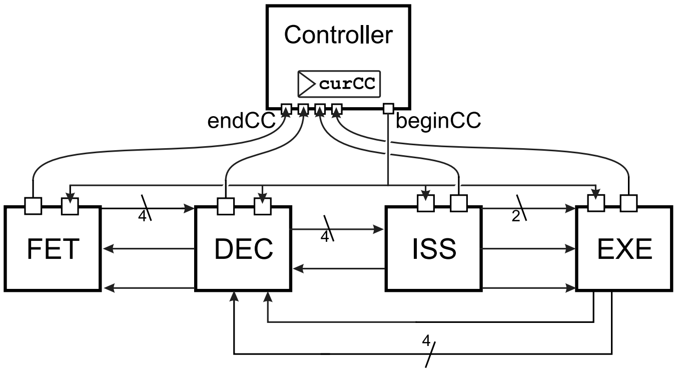


Fig. 8. Dynamic barrier synchronization with centralized controller.

simulation (Node C of Figure 6), historically used in projects such as the IBM Yorktown Simulation Engine [Pfister 1982]. This technique retains the benefit that any FPGA-optimized implementation of a circuit may be used, whether or not its cycle-by-cycle behavior matches that of the target circuit.

The advantage of the unit-delay scheme is that there is very little overhead. All modules can be implemented as finite-state machines which read their inputs, calculate for n cycles, and write their outputs. Temporal deadlocks are impossible, and temporal violations can be easily avoided by restricting producers to write their outputs only on the final FPGA cycle of a model cycle. We can create a snapshot of the system on model cycle t by observing the state of the system on FPGA cycle $n \times t$.

Such a simulator would simulate at a rate of $frequency_{FPGA}/n$. Thus unit-delay simulation is appropriate when the static worst-case n is small. In practice, however, there are likely to be rare, exceptional events that require a large amount of time to simulate. 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.

An alternative is to have the FPGA-to-model cycle ratio determined dynamically. This would be a dynamic barrier synchronization (Node D in Figure 6), where all modules coordinate dynamically on when to move to the next model cycle. As is shown in Figure 8, 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. We may create snapshots of our system by observing the state only on model cycle boundaries. Temporal deadlock is possible if an individual module does not terminate a model cycle, though this is avoidable in practice.

One example of a circuit that can take a dynamic number of FPGA cycles to simulate is a content-addressable memory (CAM). Directly implementing

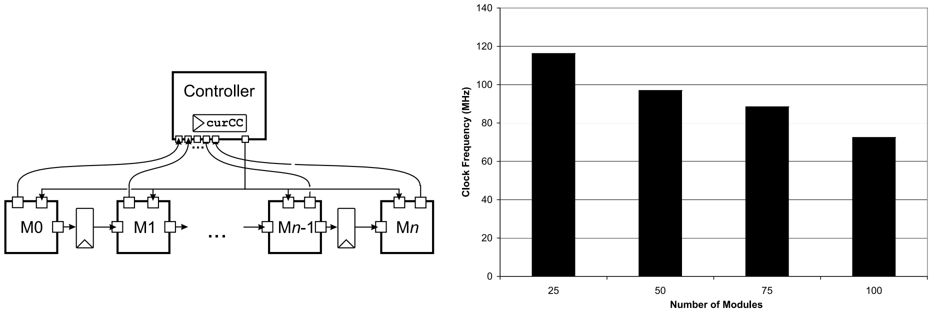


Fig. 9. Dynamic barrier synchronization's centralized controller limits scalability.

such a circuit on the FPGA can be prohibitively expensive. One alternative is to use a synchronous BlockRAM and sequentially search the memory. Under the unit-delay scheme we would have to bound n as the worst case—searching the entire RAM, which is a rare occurrence. In general, in dynamic barrier simulation we take the average number of cycles required to simulate a model cycle, while still tolerating rare worst cases when they occur. The result can be a significant decrease in FMR.

The main problem with barrier synchronization is the scalability of the central controller. 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. 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, as shown in Figure 9. In addition, we observed that the execution time of the FPGA place-and-route tools increased 20-fold over these same data points, in spite of the fact that the largest target used less than 10% of FPGA slices. We conclude that the dynamic barrier synchronization technique offers benefits over the unit-delay case, but also faces scaling issues which limit it to a small numbers of 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 FPGA frequency 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-Port Networks, a distributed simulation technique we developed for the fine-grained parallelism of FPGAs. A-Port Networks do not require explicit timestamps, static rates, or centralized barriers. We quantitatively demonstrate a performance improvement for simulating our target processor of up to 19% over dynamic barrier synchronization using the A-Ports scheme.

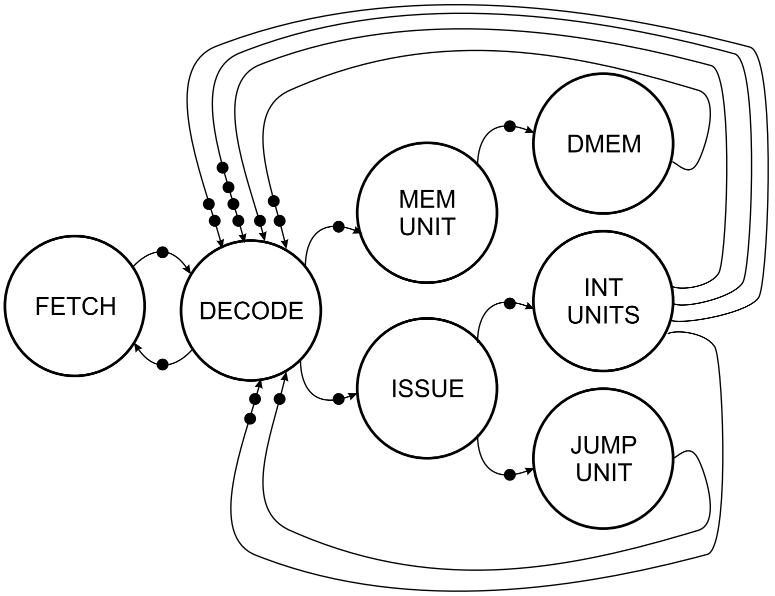


Fig. 10. An A-Port Network is a restricted Kahn process network.

4. A-PORT NETWORKS

As explained in Section 2, software Asim performance models use an explicit representation of time and a centralized controller to coordinate simulation. As we noted in Section 3, both of these choices would carry a large overhead on the FPGA. To this end we developed a novel scheme tailored to the particulars of an FPGA. We name our scheme A-Port Networks, to distinguish it from prior work on Asim ports, and to emphasize the generality of the approach.

4.1 Distributed Simulation Scheme

As shown in Figure 10, a simulation of a port-based model can be viewed as a Kahn process network [Kahn 1974]. The initial placement of tokens is derived from the latencies of the ports themselves. We can exploit the parallelism in this model if we can allow each node, or module, to proceed to the next model cycle when all incoming edges contain data, in the standard dataflow manner.

Our simulator is not an arbitrary process network. It is a reflection of a particular synchronous system. Therefore, we must restrict the nodes' behavior beyond that of general process networks in order to avoid temporal violations. Specifically, each node must always be at an identifiable model cycle k. Furthermore, the nodes at model cycle k may only observe the kth element of their incoming message streams, and may only produce the $k + 1$th element of their outgoing data streams. The key insight of the A-Port Network is that we can accomplish this by making each node behave as follows:

—Each time a node processes it must consume exactly one input from each incoming edge, and write exactly one output to each outgoing edge.

This represents a restriction over generalized process networks, where nodes can dynamically choose how many inputs to consume, and how many outputs to write. As a result of this restriction, an observer can deduce what model cycle a node is simulating by counting the number of times it has executed this simulation loop. Thus the A-Ports scheme (Node E in Figure 6) is an implicit tracking of the model clock. Additionally, no temporal violations are possible as long as nodes do not "peek" at the next values in the message stream. Also, temporal deadlocks are avoided as long as each node takes a finite amount of wall-clock time to simulate each model cycle, and sufficient buffering is present, as we discuss in Section 5.

In order to accommodate this restriction we must change the semantics of classical Asim ports. As described in Section 2, in the sequential simulator each module is told the current model cycle by a centralized controller, thus there is no issue if a module does not write one of its output ports. In the distributed A-Port Network, neglecting to write a port is no longer an option. To resolve this we introduce a special value called NoMessage, which indicates the lack of data at a particular location in the data stream. (We also use NoMessage as the initial tokens in the system.) Thus the complete distributed simulation loop is as follows:

—When all incoming A-Ports are not empty, a module may begin computation. Note that some of its inputs may be NoMessage, and that this is explicitly different from an empty port.

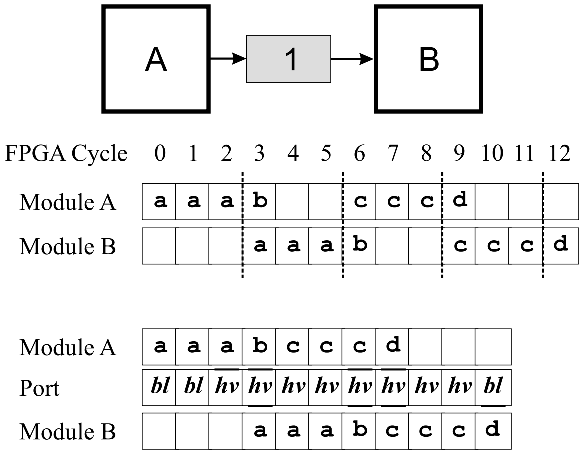
—When computation is complete, the module must write all of its outgoing A-Ports. It may write NoMessage or some other value, but must write all of them exactly once.

—The messages are consumed from the incoming A-Ports and the loop repeats.

The net effect of this simulation loop is to allow every module in the system to produce and consume data at any wall-clock rate, while still maintaining a local notion of a model clock step. To put this another way, an A-Port Network effectively turns a synchronous system into an asynchronous system, while still preserving the timed behavior of the synchronous system with respect to snapshots. In this respect A-Port Networks are similar to the Chandy-Misra-Bryant simulation scheme. The main contribution of A-Port Networks is to do this without explicit timestamps or a central controller, making it amenable to implementation on FPGAs.

Because modules simulate at different wall-clock rates, adjacent modules often are simulating different model cycles. A producer may run into the future, precomputing values as fast as possible. We say an A-Port of latency l is *balanced* when it 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.



FPGA Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12
Module A	a	a	a	b			c	c	c	d			
Module B				a	a	a	b			c	c	c	d

Module A	a	a	a	b	c	c	c	d			
Port	*bl*	*bl*	$\overline{hv}$	$\overline{\underline{hv}}$	*hv*	*hv*	$\overline{\underline{hv}}$	$\overline{\underline{hv}}$	*hv*	*hv*	$\underline{bl}$
Module B				a	a	a	b	c	c	c	d

Fig. 11. A-Port Network can improve FMR over barrier synchronization.

- —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.

We say that simulation via A-Ports is decoupled because a module can "slip" ahead as long as its input data is available. This can result in a performance improvement over barrier synchronization, as demonstrated in Figure 11. In this example, instructions *a* and *c* take more FPGA time to compute compared to *b* and *d*. Observe that on FPGA cycle 4 module A is simulating model cycle 3, whereas module B is simulating model cycle 2.

The amount 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. Selecting the appropriate buffer sizes can have a significant impact on simulator performance, as we will show in Section 5.

4.2 Obtaining Consistent Snapshots

Obtaining a snapshot of relevant state in the A-Ports scheme is complicated by the fact that the decoupled modules may have slipped in time. As we are using an implicit notion of time, the modules themselves may not know what cycle they are simulating.

One possible solution is to observe every module in a distributed fashion, and reconstruct the snapshot from these observations. For instance, an observer of the processor Fetch module could record the Fetch state after model cycle *t*, which would later be combined with the Execute state, etc. The overhead of communicating these distributed observations could become costly, similar to those of dynamic barrier synchronization's central controller. An alternative is to rebalance the decoupled modules to the same model cycle

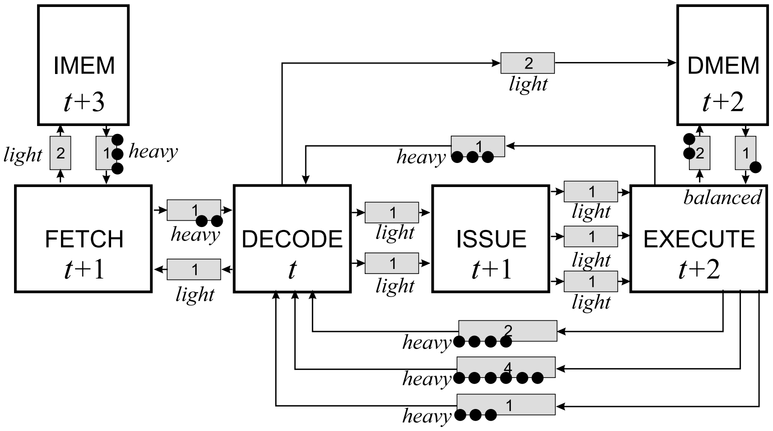


Fig. 12. Obtaining a consistent snapshot from a slipped state.

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, 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 nonempty set of modules that 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 nonempty set of modules that is farthest behind in time, and thus able to simulate the next cycle. Since 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.

Figure 12 shows an example of this quiescing. Our example processor model is in a state where the Decode module has recently had the worst FMR, and thus is simulating the oldest model cycle *t*. Note that the relationship between two modules in model time can be derived by looking at the number of messages in the connecting ports, represented by black circles.

Figure 13 shows the progression of the modules. Initially, only Decode will proceed to the next model cycle ($t + 1$, which it will do because it has heavy inputs and light ouputs, as indicated by *hv* and *lt* in the figure). Then Fetch, Decode, and Issue will proceed to cycle $t + 2$. Every A-Port is now balanced,

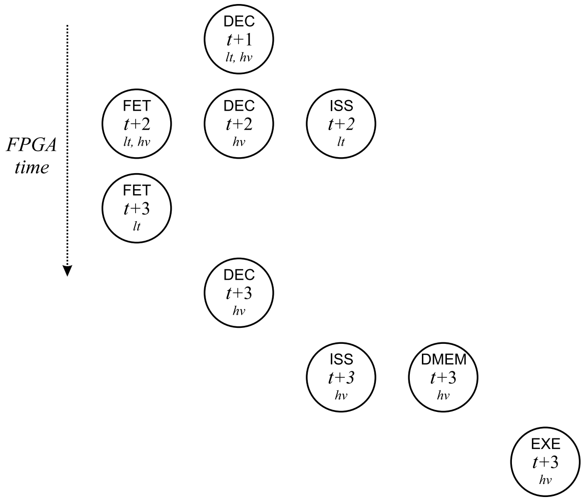


Fig. 13. Execution order to quiesce Figure 12.

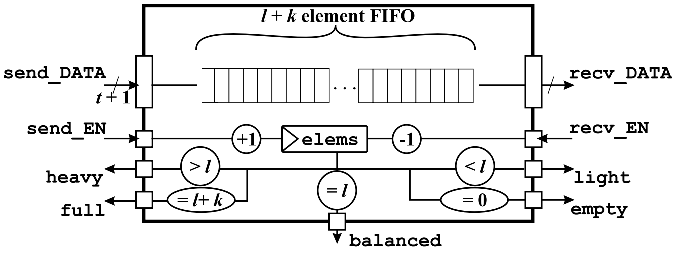


Fig. 14. A-Port implementation on FPGAs.

except for the ones between IMem and Fetch. If the modules were using the normal protocol then IMem would attempt to proceed into the future, but in this case it has no heavy inputs or light outputs. As a consequence, all the other modules will proceed one more cycle in causal order, as shown. At this point every A-Port in the system will be balanced, so the system will quiesce until it receives a command to resume simulation using the normal protocol. Note that in this state the number of messages in each A-Port matches the initialization conditions, so simulation is guaranteed to be able to resume.

As an additional benefit, 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. IMPLEMENTING A-PORT NETWORKS ON FPGAS

As shown in Figure 14, 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). On an FPGA each A-Port must have finite buffering.

In order to guarantee the absence of temporal deadlock, the following sufficient conditions must be met:

—Each A-Port of latency l must contain at least $l + 1$ buffering.

—Each A-Port of latency l is initialized to contain l copies of NoMessage at simulator startup.

—Modules should be arranged in a connected graph.

To see why this prevents temporal deadlock, 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.

These conditions are closely related to the correctness conditions of Lee’s [1987] static synchronous dataflow graphs, as we discuss in Section 6. The primary difference is that in A-Ports Networks the buffering requirements and initial placement of data is derived from the latencies of the A-Ports themselves. Thus the properties of the asynchronous implementation are correct because they reflect properties of the target synchronous system, rather than requiring the user to determine buffer sizes or placement of tokens manually.

5.1 Quantitative Assessment

In order to assess our A-Ports implementation we identified two target processors. First, a traditional five-stage in-order microprocessor pipeline. Second, the more realistic out-of-order superscalar processor, which we have used as an ongoing example. As the instruction set is not the focus of this research we chose a subset of the MIPS ISA. To maximize the impact of the processor pipeline itself, the core is assumed to be paired with one-cycle “magic” memory rather than a realistic cache hierarchy.

As shown in Figure 15, the processors were decomposed into modules and connected both using barrier synchronization and A-Port Networks. Our implementation of the model focused on efficiency of FPGA configuration. To this end we used BlockRAMs for every large structure in the processor, including the branch predictor, branch target buffer, and register file. In the superscalar processor we implemented only a single ALU and multiplexed it to simulate the four physical pipelines. The effect of these transformations was to reduce implementation effort and increase area efficiency, at the cost of using more FPGA cycles per model cycle.

The designs were implemented using Bluespec SystemVerilog, and were synthesized for a Xilinx Virtex II Pro platform and assessed for simulation speed and efficiency. We measured the targets 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

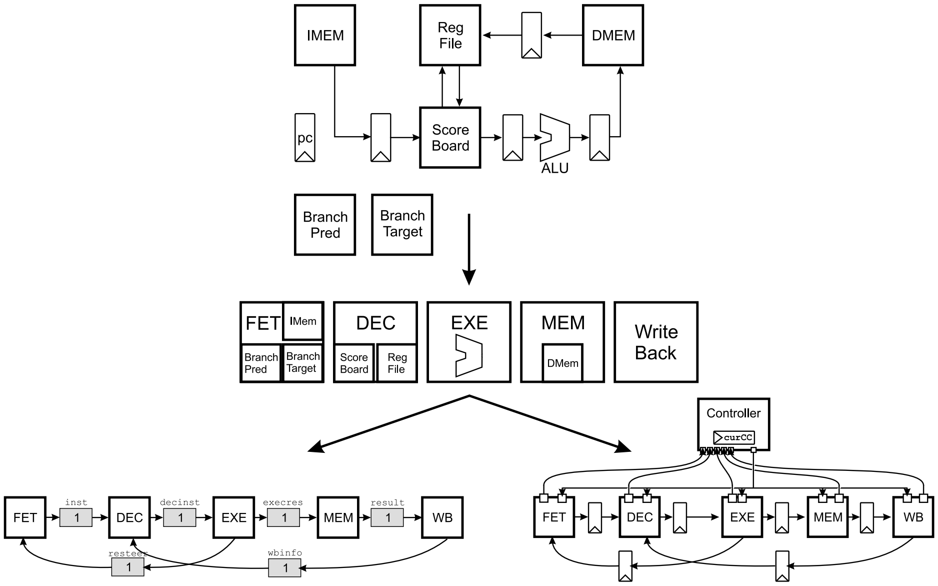


Fig. 15. Assessment methodology showing the in-order target.

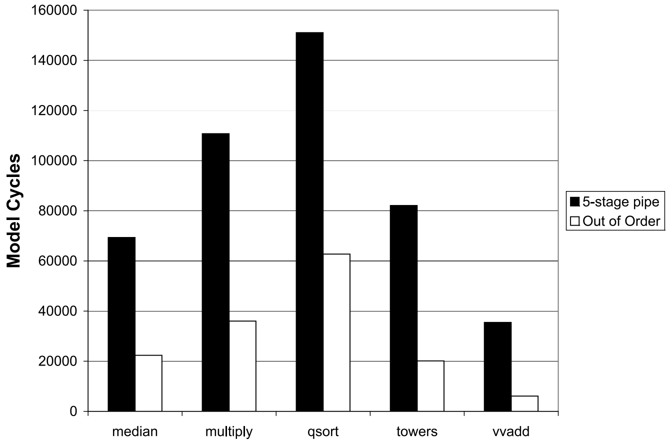


Fig. 16. Assessing the target processors as a sanity check.

conclusions from small benchmarks running on processors not paired with a realistic memory hierarchy, the results (Figure 16), show 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. These results match our intuition that the out-of-order processor is a better architecture—it would execute substantially faster (assuming the circuit design team was able to achieve an equivalent clock speed, and that the area overhead was not prohibitive).

These results represent the insights into the target design that most users of performance models care about. However, as simulator architects, we are

	5-Stage A-Ports	OOO 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

Fig. 17. Simulator synthesis results for Virtex II Pro 70.

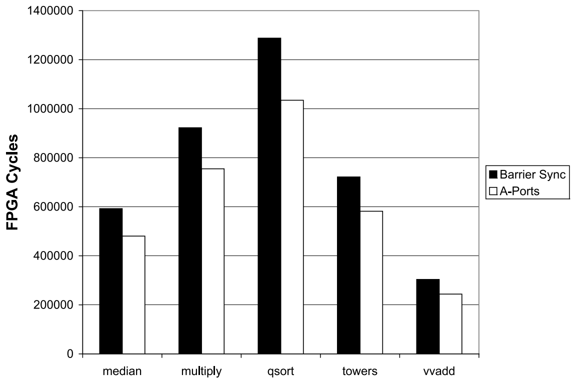


Fig. 18. Assessing the in-order simulators.

also interested in comparative simulator performance. The physical properties of the simulators are given in Figure 17. These results demonstrate that when we consider simulator performance the situation is reversed—the five-stage simulator can simulate model clocks more than twice as fast (14 MHz vs 6 MHz), due to the multiplexing of the ALU which the out-of-order superscalar model does during every model cycle. However when we consider simulated Instructions per Second, 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.

The results comparing barrier synchronization to A-Ports are shown in Figures 18 and 19. These results show that the in-order 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, the A-Ports buffer size limits the amount adjacent modules can slip in model time. Figure 20 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.

Although these assessments were done on relatively simple cores without a memory hierarchy, our hypothesis is that adding detail to these models will not

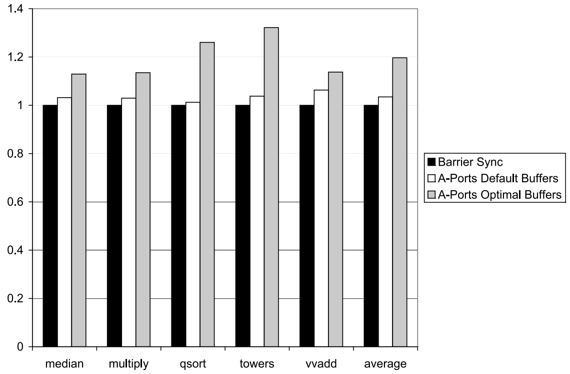


Fig. 19. Assessing the out-of-order simulators.

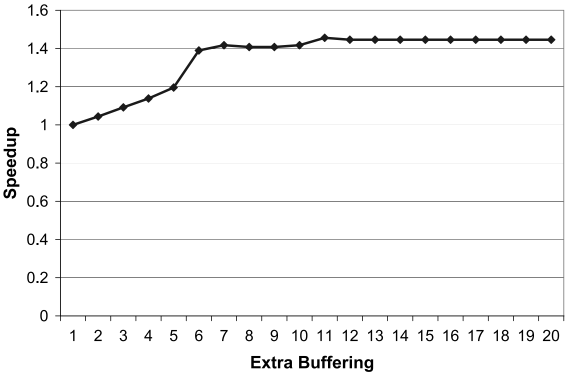


Fig. 20. Out-of-order simulator performance improvement as buffering increases.

significantly impact simulation rate. The reason is that a realistic model will use the FPGA to perform the simulation of the cache hierarchy and interconnect network in parallel with that of the core. Thus while these structures will certainly require FPGA resources, FPGA cycles per model cycle should remain relatively unchanged.

What may require more FPGA cycles to simulate is rare-but-complex target behavior such as exceptions or system call instructions. Taking multiple FPGA cycles to simulate these events can result in a significant saving of FPGA resources. (For example, by communicating with an off-FPGA simulator, as in Chung et al. [2008].) However if these events are rare enough then the impact on simlation rate should be minimized. We believe that the computer architect's principle of "make the common case fast" should be equally applicable to simulations as to the target designs themselves.

6. RELATED WORK

6.1 Performance Models on FPGAs

Early efforts at creating performance models on FPGAs such as Ray and Hoe [2003] and Wunderlich and Hoe [2004] shared the goal of creating a model early in the design process, but these efforts used the FPGA clock itself as the simulation clock, reducing fidelity in order to ease development time and save FPGA resources. Thus these are more closely aligned with what we have termed a direct emulation approach.

An alternative to re-implementing the entire performance model onto the FPGA is maintaining a software simulator and accelerating critical tasks in hardware. Penry et al. [2006] explored using the Power PCs on Xilinx Virtex II Pro 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—an emulation approach, in our terminology.

The approach of taking many FPGA cycles to simulate one model cycle was popularized by the RAMP project [Arvind et al. 2006; Wawrzynek et al. 2007]. RAMP aims to model systems with hundreds of chips in them by spreading them across multiple FPGAs, and across multiple boards. Ramp Description Language [Gibeling et al. 2006], 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 do 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 [Chiou et al. 2007a; 2007b]. 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. Subsequently, UT-FAST developed a more generalized connector which was also inspired by Asim ports, as presented in Chiou et al. [2007b]. The focus of this connector is slightly different, as it reuses the buffering of the channel itself to represent buffering of the target, which mixes concerns of simulator implementation and model properties. Additionally UT-FAST connectors use a protocol which allows them to be time-multiplexed, so that n conceptually different channels can share the same physical buffer for efficient implementation. Currently there is an ongoing collaboration to reach a convergence between UT-FAST connectors and A-Port Networks.

6.2 Process Networks and the NoMessage Value

As already noted, an A-Port network is a restricted case of a general Kahn process network [Kahn 1974], where the buffer sizes are fixed and the nodes must consume and produce exactly one input from each edge. With these restrictions the closest formalism is that of marked directed graphs [Commoner

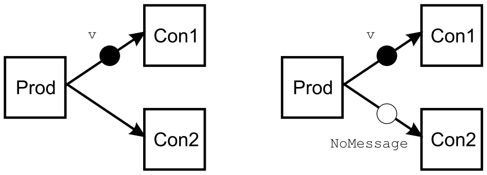


Fig. 21. In A-Port Networks, the NoMessage value is used in place of not sending a message.

et al. 1971]. As shown in Figure 21, the largest difference between A-Port Networks and classic process networks or dataflow graphs is handling the absence of data using the NoMessage value. Classically, a node may choose to send a token on one output but not another. In an A-Port Network this would cause the two recipients to disagree about the current model cycle, as the consumer node cannot distinguish between the "previous node is still computing" and the "previous node is done computing and no message is coming."

In this sense the NoMessage value plays a role similar to the null messages of the Chandy-Misra-Bryant explicit timestamp scheme [Chandy and Misra 1981]. In this scheme the simulation may deadlock unless individual modules communicate messages with a timestamp of the node's local current simulated cycle. A-Port networks can be viewed as a degenerate case of this where the fact that a message (or NoMessage) is sent at every time step replaces the timestamp itself.

A-Port Networks are also a restricted case of Lee's static synchronous dataflow [Lee and Messerschmitt 1987]. In such a system nodes statically declare how many inputs they will produce and consume, and this number need not necessarily be one per edge. It is believed, though not yet proven, that introducing the NoMessage value into an arbitrary static synchronous dataflow graph allows us to transform any synchronous dataflow graph into one where every node only produces and consumes one token on each edge per processing step (though some of those tokens may be NoMessage). If this is true, A-Port Networks represent a complete restriction.

The theory of latency-insensitive design developed by Carloni et al. [2001] shares a great deal of motivation with our work, as it aims to convert an originally synchronous system into an asynchronous system. In a properly latency-insensitive system delay-changing relay stations may be added as necessary in order to break long physical wires into smaller segments. The resulting system is latency-equivalent to the original system, a requirement which is weaker than maintaining the snapshot-equivalence we discuss here. Carloni also uses a null-message τ symbol; however, this is used as a stalling event which signals that a given node is not computing. Thus this symbol is not equivalent to our NoMessage, but is more akin to the FPGA cycles on which a module cannot proceed because one or more input A-Ports are empty. Because of this, latency-insensitive theory also requires that when a module is able to compute it must produce its output within one host clock cycle, whereas A-Port Networks allow the module any number of FPGA clock cycles to compute before producing a result.

7. DISCUSSION

In this article, we explored FPGAs as a platform for executing cycle-accurate performance models. We discussed how performance models are created in software and why contemporary mutlicores are not able to exploit the parallelism inherent in these models. We explored the strengths and weaknesses of existing distributed schemes for synchronous simulation in the particular context of FPGAs. This article, introduced A-Port Networks 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 [Pellauer et al. 2008a; 2008b] is to use A-Ports, combined with other techniques from software performance models [Pellauer et al. 2008b], to create a high-detail model of a chip-multiprocessor (CMP) on an FPGA.

ACKNOWLEDGMENTS

The authors would like to acknowledge the collaboration and valuable feedback of Angshuman Parashar, Zhihong Yu, Tao Wang, and Guan-Yi Sun.

REFERENCES

ARVIND, ASANOVIC, K., CHIOU, D., HOE, J. C., KOZYRAKIS, C., LU, S., OSKIN, M., PATTERSO, D., RABAEY, J., AND WAWRYZNEK, J. 2006. Ramp: Research accelerator for multiple processors—a community vision for a shared experimental parallel hw/sw platform. Tech. rep. University of California, Berkeley.

BARR, K. C., MATAS-NAVARRO, R., WEAVER, C., JUAN, T., AND EMER, J. 2005. Simulating a chip multiprocessor with a symmetric multiprocessor. In *Proceedings of the Boston Area Archictecture Workshop (BARC)*.

BRYANT, R. 1979. Simulation on a distributed system. In *Proceedings of the 1st International Conference on Distributed Systems*.

CARLONI, L., MCMILLAN, K., AND SANGIOVANNI-VINCENTELLI, A. 2001. Theory of latency-insensitive design. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.*

CHANDY, K. M. AND MISRA, J. 1981. Asynchronous parallel simulation via a sequence of parallel computations. *Comm. ACM*, 198–206.

CHIOU, D., SUNWOO, D., KIM, J., PATIL, N. A., REINHART, W. H., JOHNSON, D. E., KEEFE, J., AND ANGEPAT, H. 2007a. FPGA-accelerated simulation technologies FAST: Fast, full-system, cycle-accurate simulators. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'07)*.

CHIOU, D., SUNWOO, D., KIM, J., PATIL, N. A., REINHART, W. H., JOHNSON, D. E., AND XU, Z. 2007b. The fast methodology for high-speed soc/computer simulation. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'07)*.

CHUNG, E., NURVITADHI, E., MAI, J. H. K., AND FALSAFI, B. 2008. Accelerating Architectural-level, Full-System Multiprocessor Simulations using FPGAs. In *Proceedings of the 11th International Symposium on Field Programmable Gate Arrays (FPGA'08)*.

COMMONER, F., HOLT, A., EVEN, S., AND PNUELI, A. 1971. Marked directed graphs. *J. Comput. Syst. Sci. 5*.

EMER, J., AHUJA, P., BORCH, E., KLAUSER, A., LUK, C. K., MANNE, S., MUKHERJEE, S. S., PATIL, H., WALLACE, S., BINKERT, N., ESPASA, R., AND JUAN, T. 2002. Asim: A performance model framework. *Computer*, 68–76.

GIBELING, G., SCHULTZ, A., AND ASANOVIC, K. 2006. The ramp architecture and description language. Tech. rep. University of California, Berkeley.

KAHN, G. 1974. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld Ed., *Information Processing*, North Holland, 471–475.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Static scheduling of synchronous data ow programs for digital signal processing. *IEEE Trans. Comput.*

PELLAUER, M., VIJAYARAGHAVAN, M., ADLER, M., ARVIND, AND EMER, J. 2008a. A-ports: An efficient abstraction for cycle-accurate performance models on FPGAs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'08)*.

PELLAUER, M., VIJAYARAGHAVAN, M., ADLER, M., ARVIND, AND EMER, J. 2008b. Quick performance models quickly: Closely-coupled timing-directed simulation on FPGAs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'08)*.

PENRY, D. A., FAY, D., HODGDON, D., WELLS, R., SCHELLE, G., AUGUST, D. I., AND CONNORS, D. 2006. Exploiting parallelism and structure to accelerate the simulation of chip multiprocessors. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA'06)*.

PFISTER, G. 1982. The yorktown simulation engine. In *Proceedings of the 19th Conference on Design Automation (DAC'82)*.

RAY, J. AND HOE, J. C. 2003. High-level modeling and FPGA prototyping of microprocessors. In *Proceedings of the ACM/SIGDA 11th International Symposium on Field Programmable Gate Arrays (FPGA'03)*.

WAWRZYNEK, J., PATTERSON, D., OSKIN, M., LU, S. L., KOZYRAKIS, C., HOE, J. C., CHIOU, D., AND ASANOVIC, K. 2007. Ramp: A research accelerator for multiple processors. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'07)*.

WUNDERLICH, R. E. AND HOE, J. C. 2004. In-System FPGA Prototyping of an Itanium Microarchitecture. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'04)*.

Received June 2008; accepted July 2008