HAsim: Cycle-Accurate Multicore Performance Models on FPGAs

by

Michael Pellauer

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2011

Author .... [illegible]
Department of Electrical Engineering and Computer Science
November 1, 2010

Certified by .... [illegible]
Arvind
Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .... [illegible]
Joel Emer
Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .... [illegible]
Terry Orlando
Chairman, Department Committee on Graduate Students

HAsim: Cycle-Accurate Multicore Performance Models on FPGAs

by

Michael Pellauer

Submitted to the Department of Electrical Engineering and Computer Science
on November 1, 2010, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

The goal of this project is to improve computer architecture by accelerating cycle-accurate performance modeling of multicore processors using FPGAs. Contributions include a distributed technique controlling simulation on a highly-parallel substrate, hardware design techniques to reduce development effort, and a specific framework for modeling shared-memory multicore processors paired with realistic On-Chip Networks.

Thesis Supervisor: Arvind
Title: Professor, Electrical Engineering and Computer Science

Thesis Supervisor: Joel Emer
Title: Professor, Electrical Engineering and Computer Science

Acknowledgments

I'm lucky enough to have had the experience of working on a large, multi-institution, multi-company project for my thesis. As such I hope the reader will bear with me as I thank those people who have contributed to this project's success. These few paragraphs feel like an insufficient way to thank them for their help and support over the years.

Let me start by thanking Dr. Joel Emer of Intel Corporation. Joel served for many years an unofficial co-supervisor, and I'm glad MIT will now formally let me acknowledge the role he played. Joel's years of experience in the computer architecture community brought an extremely practical point of view to the project. This project would not have been so successful without Joel's ideas and guidance.

Also at Intel Corporation, I should need to acknowledge the help and support provided by Michael Adler of the VSSAD group. Michael spent a lot of time wrestling with the down-and-dirty technical aspects of interfacing with an FPGA, which freed me up to concentrate on the more academic side of things. Michael also contributed code to the HAsim functional partition, and designed a very elegant and useful scratchpad memory abstraction. For this help I am deeply grateful.

Additionally, I must thank Angshuman Parashar, also of the VSSAD group. He built the virtual platform infrastructure (tentatively named LEAP) which allows HAsim to run on different FPGAs without recoding. This work has been invaluable and should prove to be applicable to applications beyond HAsim. Other people at Intel who contributed feedback or code include Guan-Yi Sun, Tao Wang, Zhihong (George) Yu, and Artur Klauser. Martha Mercaldi, Nikhil Patil, and Abhishek Bhattacharjee served alongside me as summer interns in the Intel VSSAD group at various points throughout the years. They contributed code to various parts of the HAsim infrastructure.

At the MIT Computation Structures Group I must start by thanking Arvind, who used infallible good cheer to cover an infallible demand for precision, which improved the ultimate quality of this thesis immensely. Other contributions were made by

Muralidaran Vijayaraghavan, who designed the original out-of-order timing model and contributed to the A-Ports formalism, and Elliott Fleming, who contributed code to the virtual platform. Michel Kinsy allowed us to use his Heracles platform as the basis for our experimental comparison. Richard Uhler wrote a timing model of a simultaneous multi-threaded processor. Nirav Dave provided years of service as a sounding board, and even contributed some code to early versions of the project. Chris Batten wrote code that became some early benchmarks and testcases. Additional feedback was provided by Alfred Ng, Myron King, Asif Khan, and Abhinav Agarwal. Charlie O'Donnell served the role of making me laugh when needed.

Support was provided at Bluespec Inc by several hard-working employees. In particular Ravi Nanavati and Rishiyur Nikhil contributed compiler support and project feedback. Other support was also provided at various times by Jacob Schwartz, Jeff Newbern, Joe Stoy, Don Baltus, Ed Czeck, and Todd Snyder.

Through the RAMP project I was lucky enough to get the opportunity to present my work for feedback to some of the best minds in the field. Let me start by thanking Derek Chiou of the University of Texas, who always believed in the potential of FPGAs as accelerators. The original RAMP vision articulated by Krste Asanovic, John Warzwyrnik, and Greg Gibeling of UC Berkeley helped form this project's early direction. (I hope that HAsim has had some reciprocal influence on their direction over the years.) At Carnegie Mellon, Professor James Hoe and his student Eric Chung's work on the ProtoFlex had a great influence on our thinking on time-division multiplexing.

Last but not least, I should thank my parents Mary and David Pellauer, and my wife Shauna. I beg forgiveness for any and all times I put research before family.

Michael Pellauer

Contents

List of Figures

Chapter 1

Introduction

1.1 The Processor Simulation Problem

The processor design flow begins when the architect is given a set of requirements — e.g., construct 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. Eventually the target architecture is finalized and the costly and time-consuming step of Register Transfer-Level (RTL) hardware description can begin.

These early studies are carried out using simulators called *performance models*, so named because they give the architect a detailed trace of the dynamic behavior of the target system from one clock cycle to the next, but generally do not give insight into static circuit characteristics such as area or clock frequency.[1] In order to be successful, a performance model must meet three criteria:

1. Be accurate enough to give architects confidence in their decisions.
2. Be flexible enough to allow the architect to explore a wide range of potential targets.

3. Total time of modeling (time to develop all interesting targets and total benchmark execution time) must be short enough to keep the architects at the head of the design cycle.

Currently design teams write most such models in software, using home-brewed C/C++ simulators or frameworks such as SystemC [41]. 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 [13]. The following table shows an overview of typical simulation speeds of performance models constructed using Intel's Asim framework [20]:

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

Such low performance can limit the variety and length of benchmark runs, thus reducing confidence in architectural conclusions. Faced with this researchers have explored three complementary approaches, outlined in Figure 1-1. The first is to transform the benchmarks so that a detailed model only needs to be run on a representative part of the dynamic execution. The second is to transform the model, reducing detail in order to facilitate exploration of a wider space. These approaches are useful for initial architectural pathfinding, or if the phenomenon being studied is not dependent on the cycle-by-cycle behavior of the cores. From a practical perspective, these kinds of studies must still be backed up by high-detail models if architects are going to convince their skeptical managers, especially for more radical proposals.

This leads us to the third approach: accelerating high-detail performance modeling by parallelizing the simulator itself. Although this kind of parallelization can help somewhat, the increasing popularity of multicore architectures will actually widen the gap between simulator speed and target speed. This is because of a variety of factors.

(A) Transform the benchmarks:

SMARTS	Wunderlich et al. [60]	Systematic sampling of execution runs alternating between detailed and functional modes.
SimPoint	Perelman et al. [49]	Automatic extraction of representative regions and weighting of their frequency.
Trace-Graph Workloads	Isshiki et al. [27]	Transform benchmark into optimized trace-graph based on branches.

(B) Abstract the model:

QEMU	Bellard [4]	Functional emulator: no timing details of target.
Regression Models	Lee et al. [34]	Represent cores and contention as functions and explore space using regression modeling.
Interval simulation	Genbrugge et al. [23]	Characterize architectural performance based on intervals between major events.
Adaptive Models	Jones et al. [28]	Use JIT compilation and code-caching to amortize simulation overhead.

(C) Parallelize the model:

SlackSim	Chen et al. [12]	Allows slack synchronization between host threads to tradeoff accuracy and performance.
Graphite	Miller et al. [39]	Scales slack synchronization technique across multiple host machines.
DARSIM	Lis et al. [37]	Multi-threaded simulator with configurable slack synchronization.

(D) Accelerate the model using FPGAs:

Protoflex	Chung et al. [15]	Time-multiplexed functional emulator to accelerate functional modes of SMARTS-based simulation.
UT-FAST	Chiou et al. [13, 14]	Uses FPGA to add timing information to trace generated by modified QEMU that is able to rollback.
RAMP Gold	Tan et al. [55]	Time-multiplexed model with detailed caches but no core pipeline timings or OCN.
LI-BDN PowerPC	Vijayaraghavan et al. [57]	Automatic transformation of implementation to model.
HAsim	Pellauer et al. [46, 47]	General framework for time-multiplexed modeling with emphasis on core detail and OCN, as well as ease-of-use.

Figure 1-1: Comparison of approaches to the simulation modeling rate problem.

First, simulating four cores is fundamentally four times the work of simulating one core, but running the simulator on a four-core host machine does not in practice result in a four-fold speedup due to communication overheads. Second, next-generation multicores typically increase the number of cores, so that architects may find themselves simulating eight- or sixteen-core target machines on a four-core host. Third, the On-Chip Network (OCN) grows in complexity as the number of cores increase, requiring the simulation of more-complicated topologies and routers. To make matters worse, the age-old problem of increasing cache sizes of next-generation cores is expected to continue, meaning longer-running benchmarks become necessary to fully exercise the machine.

Some kind of sea change is necessary if high-detail modeling speeds are to remain fast enough to keep architects at the head of the design process. Recently several companies have begun producing products that allow a Field-Programmable Gate Arrays (FPGA) to be added to a general-purpose computer via a fast link such as PCIe [25], Hypertransport [18], or Intel Front-Side Bus [40]. These products have led to an interest in the processor modeling community to explore whether they can be used to accelerate performance models, similar to how a modern graphics card accelerates 3D modeling. Distributed logic simulation has been shown to have a degree of parallelism in the hundreds [53], yet these parallel tasks are typically quite small, equivalent to simulating a few gates. The hope is that FPGAs will be better able to exploit this extremely fine-grained level of parallelism. Contemporary efforts to explore FPGA-accelerated processor simulation include Liberty [48], UT-FAST [14, 13], ProtoFlex [15], RAMP Gold [55] and our own HAsim project [46, 47]. Collaboration between these groups is facilitated by the RAMP project [58].

Although FPGAs can improve simulator execution speed, the process of designing a simulator on an FPGA is more complex than designing a simulator in software. FPGAs are configured with hardware description languages, and are not integrated into most modern debugging environments. There is a danger that increased simulator development time will offset any benefit to execution time. We believe that no discussion of FPGA-accelerated simulators is complete without presenting techniques that address this problem.

This thesis presents HAsim, an approach to using FPGAs as an execution platform to accelerate high-detail performance models, while attempting to address development effort by building on techniques that have been successfully applied in Asim.

1.1.1 Summary of Contributions

HAsim is a collaborative project that spans industry and academic efforts. Within the context of this project, this thesis makes several specific contributions:

- A-Ports, a framework for distributed control of simulation on FPGAs without a centralized controller. (Chapter 3)
- A scheme for fine-grained time multiplexing of cycle-accurate processor models on a module-by-module basis, including a scheme for multiplexing the on-chip network via permutations. (Chapter 4)
- A hardware implementation of a generalized functional partitioning scheme. (Chapter 5)
- A library of reusable "plug-and-play" components for rapidly constructing processor models (Chapter 5).
- The Soft Connections abstraction to increase modularity in hardware description languages. Because this is a general technique that is applicable beyond HAsim, it is presented in Appendix A.

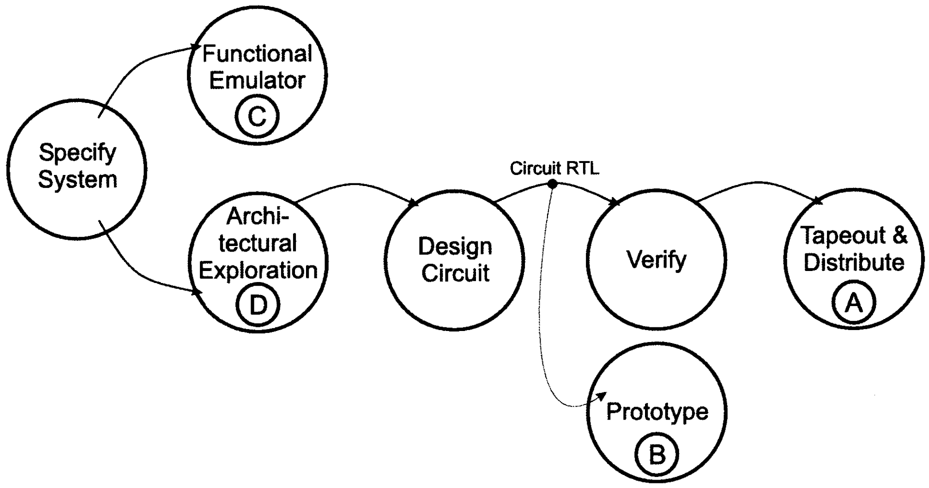


Figure 1-2: The circuit design flow. Traditionally FPGAs have been used in (A), (B), and (C). (D) represents the emerging usage model that HAsim explores.

1.2 FPGAs as Architectural Simulators

Using FPGAs for architectural simulation presents both new opportunities and new challenges. In this section we explain exactly how this use of FPGAs differs from traditional uses such as circuit prototyping. We present new metrics for reasoning about FPGA simulators and concerns about their correctness. Finally we discuss the increased development effort that comes from using FPGAs and argue for why reducing development effort is key if FPGA accelerators are to gain any success.

1.2.1 Traditional uses of FPGAs

Traditionally FPGAs have occupied three positions in the circuit design flow, as shown in Figure 1-2. The first is to distribute pre-configured FPGAs in place of a custom-fabricated chip (1-2A). In this case the RTL description of the circuit is designed with FPGAs in mind, thus can take advantage of FPGA-specific structures such as Xilinx synchronous block RAM resources.

The second use is circuit prototyping (1-2B), where FPGAs are used to aid verification before the costly step of fabrication. In this use the RTL was designed with ASICs in mind, and thus may contain circuit structures—such as multi-ported register files or content-addressable memories—that are appropriate for ASICs, but result in inefficient FPGA configurations.

The third use is functional emulation (1-2C), where a design is created which implements the functionality of the final system, but contains no information on the expected timings of the various components. Usually the goal of such an emulator is to produce a version which is functionally correct with minimal design effort. These designs may use FPGA-specific structures and avoid FPGA-inefficient ones, as they are under no burden to create a circuit related to the final ASIC.

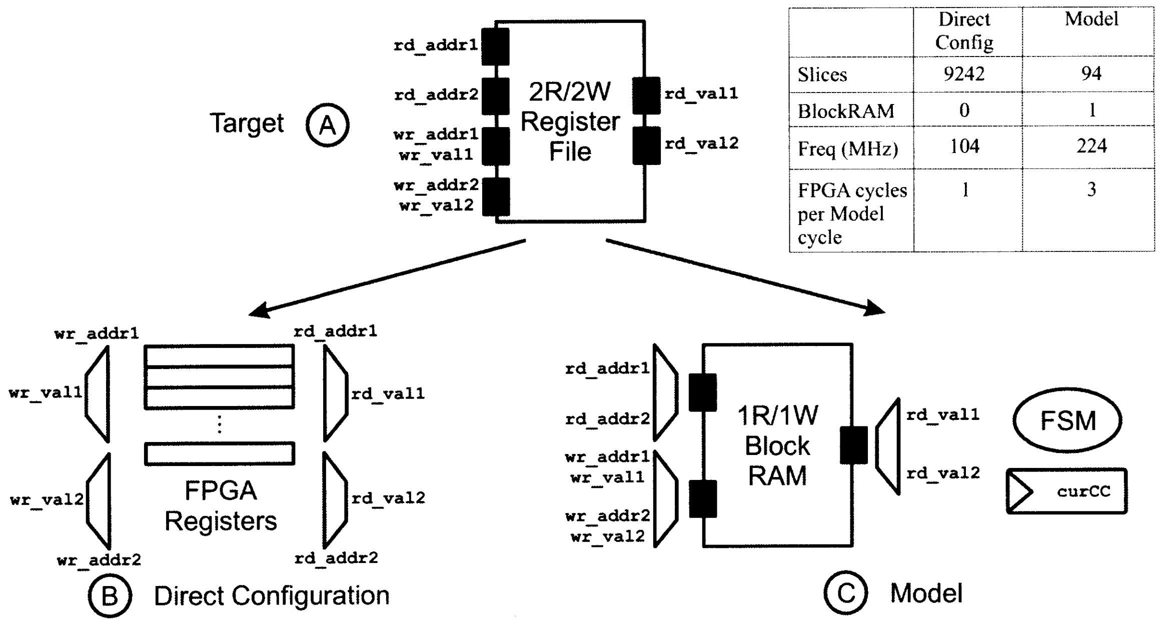
These uses are in contrast to the emerging field of using FPGAs to accelerate architectural simulation (1-2D). A simulator helps the architect to make key architectural decisions via exploration, thus it must combine the functionality of the system with some notion of expected timings of its final components, but these timings may vary wildly from those of the FPGA substrate. Similarly, if the target is expected to be implemented as an ASIC, then FPGA-inefficient structures should not be ruled out.

1.2.2 The Model Clock Versus the FPGA Clock

The key insight is that an FPGA-accelerated simulator must be able to correctly *model* the timing of all structures, but does not have to accomplish this by directly configuring the FPGA into those structures. The FPGA is used only as a general-purpose, programmable substrate that implements the model. This allows the architectural simulator to simulate FPGA-inefficient structures using FPGA-specific components, while *pretending* that their timings match their ASIC counterparts.

To illustrate this, consider the example in Figure 1-3. The architect wishes to simulate a target processor that contains a register file with two read ports and two write ports (1-3A). Read values appear on the same target cycle as an address is asserted. External logic guarantees that two writes to the same address are never asserted on the same model clock cycle.

Directly configuring the FPGA into this structure would be space inefficient because it cannot use built-in block RAM resources. Block RAM typically only has two total ports, and has different timing characteristics than the proposed register file—read values appear on the next FPGA cycle after an address is asserted. Thus



	Direct Config	Model
Slices	9242	94
BlockRAM	0	1
Freq (MHz)	104	224
FPGA cycles per Model cycle	1	3

Figure 1-3: Separating the model cycle from the FPGA cycle helps the simulator to take advantage of synchronous block RAM.

a direct emulation would use individual registers and multiplexers (1-3B), which can be quite expensive.

An FPGA-based simulator can separate the FPGA clock from the simulated *model clock*. Such a simulator could use a block RAM paired with a small finite state machine (FSM) to model the target behavior (1-3C). In this scheme the current cycle of the simulated target clock is tracked by a counter. The FSM ensures that the cycle counter is not incremented until two reads and two writes have been performed. Thus we are able to design a simulator with a high frequency and low area, at the expense of now taking 3 FPGA cycles to simulate one model cycle.^{2}

1.2.3 Space-Time Tradeoffs in FPGA Modeling

Separating the model clock from the FPGA clock allows the simulator architect to trade time for area, using efficient circuits that may take multiple FPGA cycles to simulate the target structure. For example, a Content-Addressable Memory (CAM) would be inefficient to implement directly on an FPGA, resulting in high area and a long critical path. However we can simulate a CAM using a single-ported Block RAM

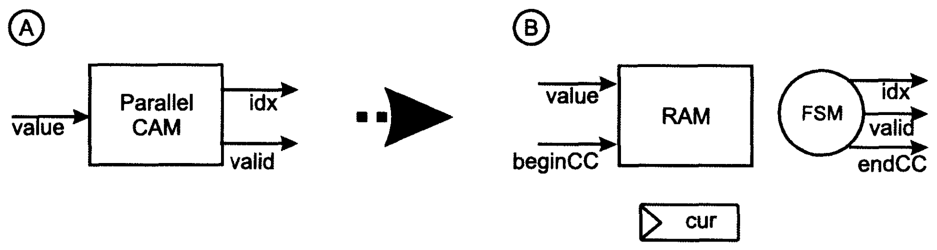


Figure 1-4: (A) CAM Target (B) Simulating the CAM with a RAM and FSM.

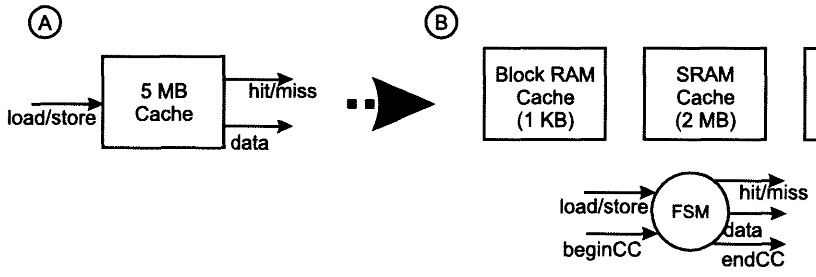


Figure 1-5: (A) Large Cache Target (B) Simulating the cache using a memory hierarchy.

and an FSM that sequentially searches the RAM, as shown in Figure 1-4. The FSM may take more or fewer FPGA cycles to search the RAM, depending on occupancy. However the model clock cycle is not incremented until the search is complete, making the structure *simulate* a parallel CAM. Taking more or fewer FPGA cycles affects the *rate* of simulation, but does not affect the *results.*

Similarly, the FPGA is inserted into a host platform that contains large amounts of host memory. Separating the model clock from the FPGA clock allows the simulator to leverage this memory, even though the sizes and latencies may be radically different than those being simulated. In Figure 1-5 the simulator is run on a platform that has three levels of memory: on-FPGA Block RAM, on-board SRAM, and DRAM managed by the OS running on the host processor. The simulator wishes to use this hierarchy to simulate a 5 MB last-level cache. It can accomplish this by allocating space in the Block RAM, the SRAM, and host DRAM—essentially using 3 caches in place of a single large cache. To simulate an access of the target cache the FPGA first checks if the line is resident in the Block RAM. If it is, the simulator can quickly determine if the access hit or missed. Otherwise, it must access the SRAM or DRAM, and possibly add the response to the BRAM. In this case, in the rate of simulation will be slower, dependent on the distance of the memory where the line resides. But note that the level of physical memory accessed affects only the *rate* of simulation, and is orthogonal to whether or not the simulated 5MB target hit or missed.

An FPGA-accelerated simulator is composed of many parallel modules, each of which can take an arbitrary number of FPGA cycles to simulate a model cycle. The problem now becomes connecting them together to form a consistent notion of model time.

1.2.4 Reasoning About 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, HAsim has developed a series of metrics for reasoning about FPGA performance models that can aid simulator architects in making judicious tradeoffs. We will use these metrics to analyze our simulator throughout this thesis.

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 1-3C, the model takes 3 FPGA cycles to simulate one model cycle, for an FMR of 3. 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.

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 cycles-per-second (Hertz). In practice we find that simulated cycles-per-second 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 and improve Hertz. 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 the above formula, we can deduce the IPS of an FPGA performance model:

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

1.2.5 Logic Emulation and FPGA/Model Cycle Separation

The field of *logic emulation* attempts to use multiple FPGAs to simulate a prototype of a circuit netlist. Communication between FPGAs can take several FPGA cycles. Therefore, if different FPGAs house logic that communicate in fewer cycles in the original netlist, then some sort of separation between the FPGA cycle and model cycle is necessary. To this end, logic emulators developed mechanisms whereby each FPGA could separately perform logic emulation, then stall the FPGA's clock, then perform synchronization between the various FPGAs, and then proceed with logic emulation, and so on. The Standard Co-Emulation Modeling Interface (SCE-MI) [1] is an industry-standard mechanism for doing this kind of coordination.

In such a situation, the physical pins leaving the FPGA become a precious resource. Virtual Wires [2] is a technique developed to enhance the utilization of each physical pins by multiplexing several wires in the model onto a single physical pin. In this scheme, the FPGA performs a logic emulation step, then stalls while the multiplexed communication is carried out, then resumes logical emulation, and so forth.

This scheme was later extended to the TIERS scheduling algorithm [52] which allowed for intelligent scheduling of the logic emulation across multiple FPGAs, effectively reducing the FPGA-cycle-to-model-cycle ratio by pipelining.

These techniques differ from the techniques we present in two key ways. Specifically, while logic emulation techniques do separate the FPGA cycle from the model cycle, the logic emulation step itself always takes a single tick of the FPGA clock—it is coordination between the different logic emulation steps that requires multiple FPGA cycles. In the scheme presented in Chapter 3 there is no such restriction, which allows our work to make more efficient use of a single FPGA. The multiplexing scheme presented in Chapter 4 is also performed for the logic emulation pipelines themselves, rather than for off-FPGA communication.

Our work limits itself to the consideration of a single FPGA. A potential approach to expand our techniques to multiple FPGAs, would be to combine the Virtual Wires/TIERS schemes with our approach.

1.2.6 FPGA Models and Target Circuit Characteristics

It is important to note one way in which FPGA performance models share the same restriction as software simulators: they give little insight into the physical properties of the target design. Because the RTL used to configure the FPGA into the simulator makes many accommodations for FPGAs, its device utilization and critical path are unlikely to give insight into those characteristics of the target design. We note that there are scenarios in which the FPGA simulator characteristics may give some degree of insight into the corresponding characteristics of the final design. However, this should not be one of the goals of using an FPGA for architectural simulation, as it is by no means guaranteed.

1.3 Related Contemporary Approaches

The HASim project started at a time when research into FPGA-accelerated simulation was just beginning to emerge as a field. This is because of a confluence of factors.

First, FPGAs themselves became markedly better, passing a threshold whereby their increased capacity and speed made them more attractive as a platform. (Contrast the differences between Xilinx Virtex II generation, and its Virtex 4 and 5 generation.) Second, a new class of products emerged that allowed FPGAs to be added to host computers via fast links such as PCIe, Hypertransport, or Intel Front Side Bus. This allowed a shift in thinking from FPGAs as stand-alone systems to accelerators of general-purpose computation. Finally, the difficulty of simulating multicore processors led to simulator architects to explore if FPGAs were suitable for more than just prototyping or emulation.

The approach taken by Penry et al.'s accelerator for Liberty [48] is a good example of an early approach to FPGA-accelerated simulation. Liberty is an existing software simulator that included a mechanism for execution on a parallel host via barrier synchronization. Penry et al.'s work allowed a software thread to be replaced with a PowerPC from a Xilinx Virtex IIPro FPGA. Additional logic was synthesized around the processor to correctly integrate it with Liberty's parallel task scheduler, and stall it when input was not ready. The PowerPC cores executed instructions faster than a pure software model of an equivalent processor, however did not allow architects freedom to change the characteristics of the pipeline model. Liberty's approach successfully demonstrated that large speedups could be gained from FPGAs over software simulation.

1.3.1 The RAMP Project

The Research Accelerator for Multicore Processors (RAMP) project was founded in 2005 as an umbrella project to explore using FPGAs to accelerate architecture research [58]. RAMP brought together collaborators from different universities with the goal of developing a common shared infrastructure for FPGA usage, as well as a cross-pollination of ideas and approaches. HAsim was an early member of RAMP and continued as a participant throughout the project's lifetime. In the following sections we discuss these projects and contrast their approaches with that of HAsim. A timeline of the various projects that participated in RAMP is given in Figure 1-6.

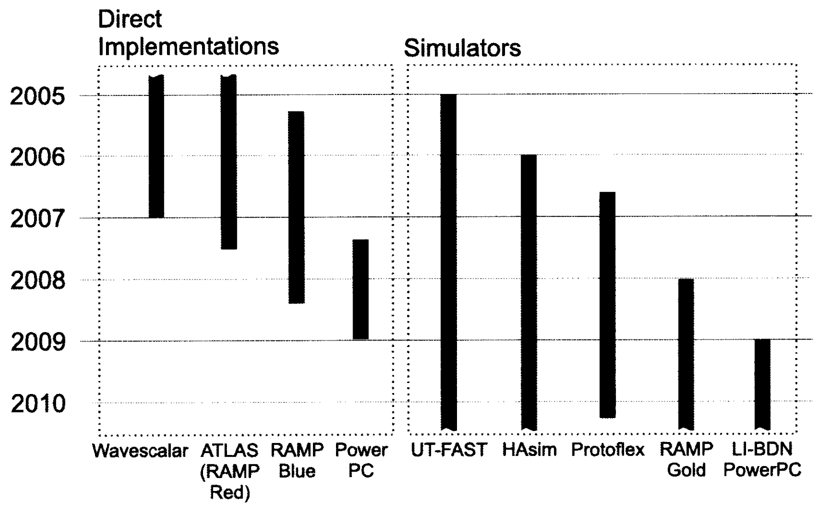


Figure 1-6: Timeline of projects participating in the RAMP collaboration. Source: Derek Chiou

The earliest goal of RAMP was to provide multicore implementations that could be used by software implementors and OS researchers as a platform that would be orders of magnitude faster than software-emulated multicores [45]. To this end RAMP projects turned to the Berkeley Emulation Engine 2 (BEE2) board [11], which was originally designed for wireless research. A key contribution of the RAMP project was the creation of the BEE3 board, [17], which was designed specifically with architectural modeling as an intended application.

1.3.2 RAMP Blue

RAMP Blue [32] was an early result of the RAMP collaboration. RAMP Blue connects multiple BEE2 boards, each of which contains multiple FPGAs. The FPGAs themselves are configured into several Xilinx MicroBLAZE soft cores connected into a disjoint-memory, message-passing network. RAMP Blue met the goal of being several orders of magnitude faster than software emulation, however a downside of the approach was that it did not represent any particular future multicore processor that would be fabricated, which reduced the incentive for software implementors to target the platform.

1.3.3 ATLAS (RAMP Red)

ATLAS [59]—also known as RAMP Red—is a multicore processor that implements a transactional memory multicore using the BEE2. Similar to Liberty, ATLAS uses the hard PowerPC processors on the FPGA, but augments them with a transactional memory component. Execution of code on ATLAS was several orders of magnitudes faster than under software emulation, which aided software implementors tremendously. However the timings of ATLAS's network communication were dictated by the physical links of the BEE2 boards and the PowerPC processors, which reduced utility for computer architects.

1.3.4 Beehive

The Beehive multicore [56] is the latest generation processor to take the RAMP Blue/Red approach: it implements a multicore on an FPGA, but does not strive to model any particular ASIC. Beehive's goal is to be accessible to a large audience by moving away from the expensive BEE2/3 platforms, aiming instead for the widely available Xilinx University Program (XUP) Virtex 5 board.

1.3.5 Towards Flexible Architectural Models

Within the RAMP community a realization arose that while these projects were useful to software implementors looking for large multicore platforms, they were less interesting to computer architects looking to model a next-generation ASIC [31]. The key idea that emerged was the notion that the *target* or *model* cycle can be separate from the cycles of *host* or *FPGA* clock. This insight allowed the creation of projects that model systems with different timings than those imposed by the FPGA platforms, making the projects a *means to an end*, rather than an end in themselves.

HAsim has made liberal use of this technique (Section 1.2), as did the RDL, UT-FAST, Protoflex, and RAMP Gold projects. An overview of the these projects approaches is given in Figure 1-7. We discuss their approaches in detail and contrast them with HAsim in the following sections.

	Functional Model	Timing Model	Time Multiplexed	Comments
RDL [24]	None	FPGA	None	Language for constructing FPGA-accelerated simulators, separating FPGA cycle from model cycle.
ProtoFlex [15]	FPGA	Software	64	SMARTS-style functional/timing split.
UT-FAST [13]	Software	FPGA	None	Software feeds trace to FPGA, which adds timing and may rollback software.
RAMP Gold [54]	FPGA	FPGA	64	Loosely-coupled partitioning focuses on efficient in-order core implementation.
HAsim	FPGA	FPGA	16	Closely-coupled partitioning enables modeling out-of-order cores.

Figure 1-7: Comparison of FPGA-based processor simulators.

1.3.6 RAMP Description Language

As explained in Chapter 3 HAsim's A-Ports arose as a way to coordinate distributed simulation between many modules, each taking a different number of FPGA cycles per model cycle. Gibeling's RAMP Description Language (RDL) [24] attempted to solve both the FPGA development-time problem and the simulation coordination problem via a language-level solution. RDL modules were connected by *channels* somewhat similar to A-Ports. The main differences between RDL channels and A-Ports were as follows:

- RDL channels represent FIFO's in the target, and thus represent both the time to move through the channel, and also the model-level state required to buffer the FIFO. A-Ports simplify this by representing only the time to move through the channel. This simplifies their implementation, and also increases their utility by allowing them to connect modules in the target system even if these modules are not connected by FIFOs.

- RDL channels impose a credit-based protocol on the users. This protocol is convenient for modules that are separated by long latencies, but has less utility for modules that are closely coupled.

Note that an RDL channel can be implemented using four A-Ports, and buffing to represent the channel, as shown in Figure 1-8. RDL channels are well-suited for large-scale modules that must be connected by a credit-based buffering. A-Ports finer granularity makes them more widely applicable in a wider range of modeling circumstances.

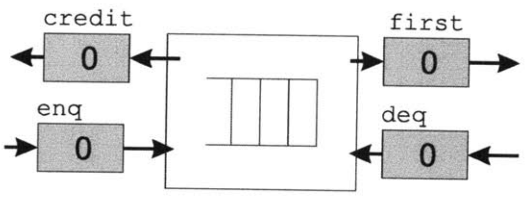


Figure 1-8: Implementing an RDL Channel with 4 A-Ports.

1.3.7 Protoflex

Chung et al.'s Protoflex [15] is designed to accelerate SMARTS-style simulation [60]. In this style, a detailed timing model is run only on small samples of a large benchmark run. For the rest of the run, a functionally accurate emulator is used, but detailed timings are not needed. Using this style of simulation, it is not uncommon for the functional emulation portion of the simulation to become the rate-limiting step of simulation—although emulation executes instructions faster than cycle-accurate modeling, the emulator must run on orders of magnitude more instructions than the cycle-accurate model.

Given this, Protoflex attempts to accelerate the functional-emulation of a multicore processor using an FPGA. However the result of this emulation can later be assigned a timing under SMARTS-style simulation, and this timing can be orthogonal to the timings of the FPGA platform. Therefore we categorize Protoflex as a simulator overall, even though it uses the FPGA to implement an emulator-like component that does not track model time.

Not having to worry about FPGA timings allowed Protoflex to make two key contributions. The first is *migration* of emulation between the FPGA and the CPU in the host computer. Protoflex observed that there are a class of instructions that are quite rare, but also expensive to implement on the FPGA, both in terms of development effort and FPGA area. Rather than wasting resources on these, Protoflex detects their presence, and migrates the state of emulation to the CPU. The instructions are emulated in software, and the resulting state is passed back to the FPGA. This migration is slow, but if these events are rare enough than impact on simulation rate is minimized.

The second contribution is *time-multiplexing.* Protoflex's implementors observed that the physical pipeline was idle during migration events, and thus could be used to execute other threads. This observation was extended to note that the processor pipeline could be greatly simplified if each pipeline stage was executing a different thread, as no hardware for hazard detection or stalling the pipeline was necessary. Protoflex uses one physical emulator to sequentially execute 16 threads.

HAsim has been greatly influenced by Protoflex, using a technique similar to migration to interact with a software simulator (Section 5.2.2). HAsim uses the multiplexing approach but extends it to a fine-grained timing-accurate pipeline where the cycle-by-cycle behavior is being continuously modeled (Section 4.1).

1.3.8 UT-FAST

Chiou et al.'s UT-FAST [13, 14] makes different placement decisions than Protoflex: it uses a software functional emulator to generate an instruction stream which is then fed to an FPGA that adds cycle-by-cycle timings to the stream. The functional emulator is a version of QEMU [4] which has been modified to support checkpointing and rollback. This allows the timing model to redirect the functional emulator when the architectural stream diverges from the timing model's path, similar to a timing-directed simulator (Section 5.3.1). Unlike in a traditional software timing-directed simulator, the long latency between the FPGA and CPU means that the functional emulator cannot stall for feedback from the timing model after producing every instruction. Instead, UT-FAST uses a *speculative* functional emulator whereby the instruction stream is pre-computed along the path that the functional emulator predicts the timing model will take. For example, the functional emulator uses a branch predictor *predictor* to model which path the timing model takes.

As described in Section 5.3.1, HAsim also uses a timing-directed approach with a functional partition and a timing partition. The main difference is that in HAsim both partitions are placed on the FPGA. This leads to a more traditional partitioning, whereby the functional partition does not need to speculate as to the timing model's direction, as it can receive feedback directly after every instruction.

1.3.9 RAMP Gold

Tan et al.'s RAMP Gold [55] is a partitioned, time-multiplexed simulator. Like HAsim, RAMP Gold places both the timing and functional partitions on an FPGA. Where HAsim and RAMP Gold differ is in the granularity of the partitioning. RAMP Gold places an emphasis on scaling cache studies to large multicores. As such, it focuses on efficiency of implementation and scaling, at the cost of reduced detail in the model of the core pipeline. Because RAMP Gold does not model a realistic core pipeline, it is able to scale its time multiplexing to 64 virtual instances. However, this scaling comes at the cost of generality in the functional partition. RAMP Gold's functional partition does not support rollback, and thus does not allow modeling of micro-architectures that use branch prediction or out-of-order execution.

In contrast, HAsim's general partitioning scheme (Section 5.4) allows it to model speculative, out-of-order, and superscalar cores, but at the cost of being able only to scale to 16 virtual cores on the current generation of FPGAs. Which approach is more appropriate depends only on the level of detail that the architect requires in the core model in order to conduct their study.

1.3.10 LI-BDN PowerPC

The RAMP PowerPC project represents an interesting case of a project beginning as an implementation, before moving into a simulator. The project originally began as a direct implementation of a PowerPC processor on an FPGA [19]. This implementation was found to have unfavorable characteristics on the FPGA, particularly for structures such as the multi-ported register file. Vijayaraghavan developed a technique known as Latency-Insensitive Bounded Dataflow Graphs [57] which allowed the separation of the FPGA cycle from the model cycle, and the transformation of the implementation into a model.

Vijayaraghavan was an original collaborator on the A-Ports simulation scheme (Chapter 3) and his LI-BDN work can be seen as an extension of that collaboration. The main contribution of LI-BDNs is the ability to begin with an original circuit description (such as the PowerPC) and automatically transform it into a latency-insensitive circuit. This then allows the designer to alter the FPGA implementations of problematic modules in the system without needing to coordinate the different FPGA-to-model cycle ratios. This technique is discussed more thoroughly in Section 3.7.4.

1.4 Discussion

The idea of using FPGAs as accelerators for architectural simulation is an emerging and active field of research. Unfortunately, developing an FPGA-accelerated simulator currently requires substantially greater engineering effort than developing a traditional software simulator. If development effort is too high, there is a danger that FPGA-accelerated simulation will fail because the total time of modeling (meaning the time to develop the model plus the time to run the benchmarks) still exceeds software simulation. HAsim treats development effort as a first-level concern and has developed several novel contributions to address the development effort problem.

Document Outline

The rest of this document is organized as follows:

- Chapter 2 presents the FPGA development effort problem, and HAsim's specific contributions to address this problem.

- Chapter 3 presents A-Port Networks, HAsim's novel fine-grained distributed simulation scheme, which allows different parts of the FPGA to simultaneously simulate different model clock cycles.

- Chapter 4 presents HAsim's technique for fine-grained time-multiplexing, allowing a single physical core on the FPGA to be simultaneously simulating different parts of many virtual cores. HAsim also uses a novel technique to simulate the interaction between the virtual cores in the simulated on-chip interconnect.

- Chapter 5 presents details of HAsim's implementation on FPGAs.

- Chapter 6 assesses the impact of the various techniques, both on simulation performance and development effort. We present a small case study that argues the merits of high-detail simulation. We discuss potential future work, and conclude.

- Appendix A presents Soft Connections, a general technique for improving modularity in hardware designs. HAsim uses soft connections extensively, but the technique is general, and thus we have reserved it for an Appendix.

Notes

[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.

[2]This is 3 instead of 4 because the simulator can perform the first write on the same FPGA cycle as the second read to the synchronous block RAM.

Chapter 2

The HAsim Approach to Reducing Development Effort

2.1 The FPGA Development Effort Problem

There is a danger that FPGAs as architectural simulators might fail not because of simulator performance, but because the increased development time means that the total time of modeling still exceeds that of slower-performing software models. Given this, HAsim treats development effort as a first-order concern and applies several techniques to minimize the development effort problem. This chapter presents an overview of HAsim's approach, and presents two specific contributions designed to reduce development effort.

2.1.1 FPGA Development Versus ASIC Development

Let us begin by clearly acknowledging that developing an FPGA accelerator is significantly simpler than ASIC hardware description and manufacture. This is for three main reasons:

1. FPGA developers do not have to worry about physical circuit characteristics, such as parasitic capacitance, that complicate ASIC development.

2. The reconfigurable nature of the FPGA allows an iterative approach to development, rather than working towards an irrevocable tapeout.

3. Because FPGA accelerators are added to existing general-purpose computers, the capabilities of the host computer can be used to aid in debugging and to perform functions too complicated to implement on an FPGA.

Development of a manufacturable ASIC requires thousands of engineering-hours. Even though FPGA development may be several orders of magnitude cheaper than this, the effort required may still exceed what is considered acceptable for an architectural simulation.

2.1.2 FPGA Development Versus Software Development

Most contemporary architectural simulators are constructed in software. This eases development time and allows computer architectural studies to be conducted by small teams of skilled architects. Unfortunately, developing FPGA simulators currently requires more investment of development effort than software, for the following reasons:

1. FPGAs are configured using hardware description languages, and are currently not integrated into debugging environments.

2. There is no equivalent of software's standard library infrastructure. This complicates printout-oriented debugging, among other problems.

3. The long running times of FPGA synthesis and place-and-route tools can lengthen the compile-run-debug loop.

4. FPGAs have finite capacity, and so the design must fit in the FPGA or the synthesis tools will fail.

2.2 HAsim Overview

Given the significance of the development effort problem, the goal of the HAsim project is to create a framework for constructing efficient simulators out of a library of

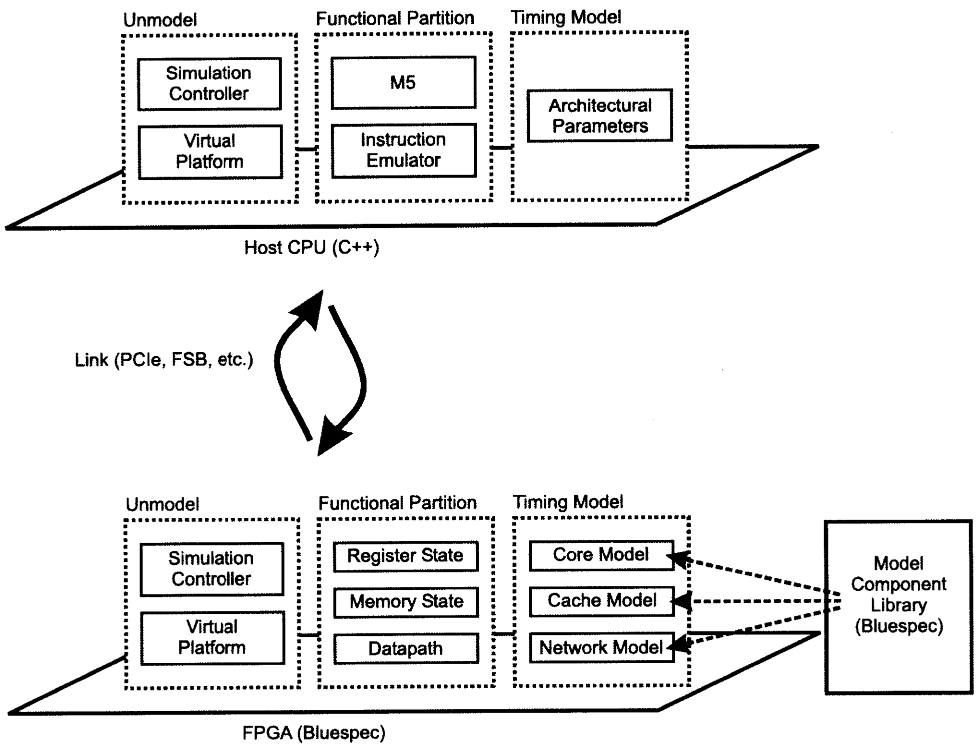


Figure 2-1: Overview of a HAsim model.

reusable components, rather than emphasizing any particular target processor. Where possible, HAsim leverages techniques developed historically for the Asim simulator [20]. In other places, HAsim has developed novel techniques that distinguish it from contemporary FPGA-accelerated performance models.

Figure 2-1 shows an overview of a model written in HAsim. The details of many aspects of this picture will be explained over the course of this document. The most important thing to note is that HAsim is a *hybrid* model, consisting of code running on a general purpose CPU as well as an FPGA. In this scheme we can leverage the strength of each physical platform: the FPGA for fine-grained parallelism, and the CPU for rare-but-difficult-to-implement events, such as system calls.

A key way to reduce development effort is to reduce the amount of code that the architect must change in order to construct their design space exploration. HAsim is divided into four major components:

- The *functional partition* is responsible for correct ISA-level execution of the instruction stream.

- The *timing partition* (or *timing model*) is responsible for tracking microarchitectural-specific timings, such as branch predictors and cache misses.

- A library of pre-defined modeling components, such as branch predictors and caches.

- The *unmodel* component refers to all functionality not directly related to simulation, including the ability to track statistics and parameters, as well as the *virtual platform* necessary to interact with the host CPU.

Under most scenarios, a computer architect using HAsim is required only to change the timing model. Additionally, the library of pre-defined modeling components can reduce this even farther, by allowing architects to adapt pre-existing modules for their experiments.

In the remainder of this chapter we present detailed discussions of HAsim's approach to lowering development effort. We begin by discussing the application of existing design engineering practices in the context of an FPGA accelerator. Techniques described in this chapter is joint work with Angshuman Parashar and Michael Adler. We will discuss this thesis's novel contributions in the domain of development effort separately.

2.3 High-Level Hardware Description Languages

Structural hardware description languages such as VHDL and Verilog give designers precise control over their microarchitectures. However, this control often comes at the cost of complex, low-level code that is unportable and difficult to maintain.

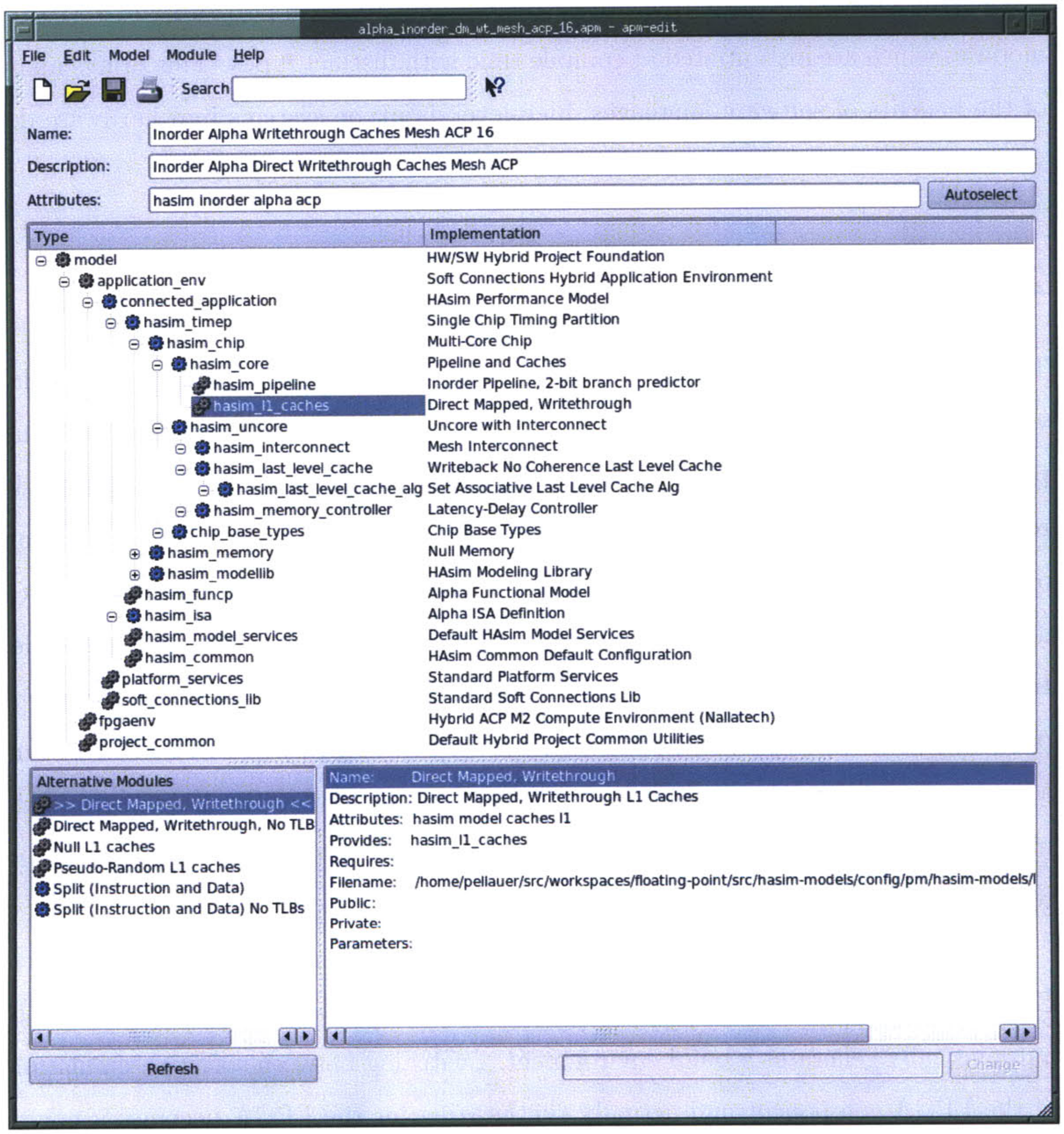
When using an FPGA as a simulator the architect is not describing a final product, and thus does not need such exacting control. Thus high-level hardware description languages such as Bluespec SystemVerilog [7], HandelC [42], or SystemC [41] can be a good fit.[3]

The HAsim [47], UT-FAST [13], and ProtoFlex [15] FPGA simulators are all written in Bluespec SystemVerilog. The benefits of Bluespec are similar to those for using high-level languages in software development: raising the level of abstraction improves code development time and reuse potential, while simultaneously eliminating many low-level bugs caused by incorrect block interfacing. Bluespec also features a powerful *static elaborator* that allows the designer to write polymorphic hardware modules which are instantiated at compile time with distinct types. This brings many of the benefits of software languages' high-level datatype systems into hardware development. Static elaboration is discussed in more detail in Appendix A.

2.4 Architect's Workbench and Modularity

The Asim Architect's Workbench (AWB) [21] is an existing framework for the development of performance models. It aims to improve the performance modeling process, especially in the early exploratory stage, by supporting modularity and code reuse. This support is provided at two levels: First, AWB supports a representation of a model as a hierarchical tree of modules, where each module can be replaced with alternative implementations that satisfy the interface requirements of the module. In fact, these replacements allow complete control of the structure of the tree for a particular model. This allows a wide variety of different models to be constructed out a common pool of modules. At a second level of modularity, AWB allows these modules to be obtained from an arbitrary set of independently-maintained source-code repositories.

HAsim has integrated itself into AWB. This allows computer architects to use a GUI to configure their modules. Once a model configuration has been selected, AWB automatically generates a build environment. Switching between software simulation of the FPGA component and actually synthesizing for the FPGA becomes a point-and-click choice. A typical HAsim model in AWB is shown in Figure 2-2.


alpha_inorder_dm_wt_mesh_acp_16.apm - apm-edit
File Edit Model Module Help
Search
Autoselect
Refresh
Change

Name: Inorder Alpha Writethrough Caches Mesh ACP 16

Description: Inorder Alpha Direct Writethrough Caches Mesh ACP

Attributes: hasim inorder alpha acp

Type	Implementation
model	HW/SW Hybrid Project Foundation
application_env	Soft Connections Hybrid Application Environment
connected_application	HAsim Performance Model
hasim_timep	Single Chip Timing Partition
hasim_chip	Multi-Core Chip
hasim_core	Pipeline and Caches
hasim_pipeline	Inorder Pipeline, 2-bit branch predictor
hasim_l1_caches	Direct Mapped, Writethrough
hasim_uncore	Uncore with Interconnect
hasim_interconnect	Mesh Interconnect
hasim_last_level_cache	Writeback No Coherence Last Level Cache
hasim_last_level_cache_alg	Set Associative Last Level Cache Alg
hasim_memory_controller	Latency-Delay Controller
chip_base_types	Chip Base Types
hasim_memory	Null Memory
hasim_modellib	HAsim Modeling Library
hasim_funcp	Alpha Functional Model
hasim_isa	Alpha ISA Definition
hasim_model_services	Default HAsim Model Services
hasim_common	HAsim Common Default Configuration
platform_services	Standard Platform Services
soft_connections_lib	Standard Soft Connections Lib
fpgaenv	Hybrid ACP M2 Compute Environment (Nallatech)
project_common	Default Hybrid Project Common Utilities

Alternative Modules
- >> Direct Mapped, Writethrough <<
- Direct Mapped, Writethrough, No TLB
- Null L1 caches
- Pseudo-Random L1 caches
- Split (Instruction and Data)
- Split (Instruction and Data) No TLBs

Name: Direct Mapped, Writethrough
Description: Direct Mapped, Writethrough L1 Caches
Attributes: hasim model caches l1
Provides: hasim_l1_caches
Requires:
Filename: /home/pellauer/src/workspaces/floating-point/src/hasim-models/config/pm/hasim-models/l
Public:
Private:
Parameters:

Figure 2-2: A HAsim model in the AWB GUI. Plug-and-play alternatives to the L1 caches are highlighted in the Alternative Modules subpane.

2.5 The LEAP Virtual Platform

Development efforts can be further eased by adopting a standardized set of interfaces for the FPGA to talk to the outside world. This *virtual platform* provides a set of virtualized device abstractions to FPGA developers, enabling them to focus on implementing core functionality without spending time and energy debugging low-level device drivers. Furthermore, most FPGA-based simulators are likely to be hosted on hybrid compute platforms comprising one or more FPGAs, and one or more CPUs. Extending well-understood communication protocols such as remote procedure call (RPC) and Shared Memory to the hybrid CPU/FPGA environment makes the platform more approachable for sharing responsibilities between the FPGA and the CPU.

Figure 2-3 illustrates the structure of HAsim's virtual platform. The primary interfaces between the simulator and the platform are a set of virtual devices and an RPC-like communication protocol called Remote Request-Response (RRR) that enables multiple distributed services on the CPU and the FPGA to converse with each other [43]. The primary benefit of this approach is portability—the virtual platform can be ported to a new physical FPGA platform without altering the application. Only low-level device drivers must be rewritten.

HAsim's virtual platform is a general abstraction for FPGA programming, and is completely independent of architectural simulation. To emphasize this we have renamed the virtual platform to LEAP—Logic-based Environment for Application Programming [44]. The LEAP platform has now been used in an H.264 implementation [22], and an academic study of a solid state disk drive [36].

2.5.1 LEAP Scratchpads

Traditionally, developers accelerating applications on FPGAs have nothing but raw memory devices in their standard toolkits. Each project typically includes tedious development of application-specific memory management which is not reused across projects. Software developers expect a programming environment to include automatic memory management. Virtual memory provides the illusion of very large arrays

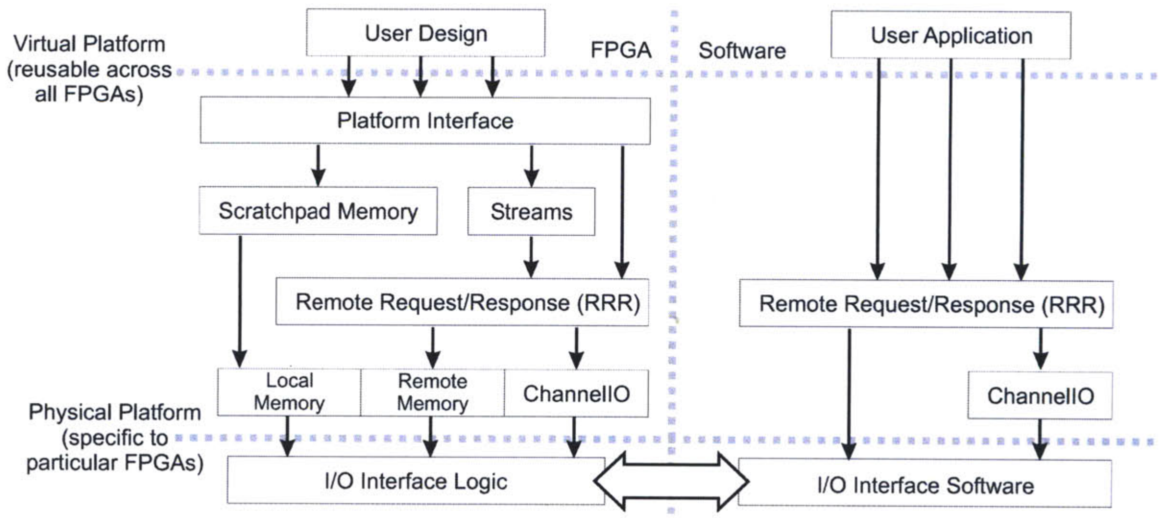


Figure 2-3: HAsim's LEAP Virtual Platform.

and processor caches reduce access latency without explicit programmer instructions.

LEAP *scratchpads* are an abstraction that dynamically allocate and manage multiple, independent, memory arrays in a large backing store. Scratchpad accesses are cached automatically in multiple levels, ranging from shared on-board, RAM-based, set-associative caches to private caches stored in FPGA RAM blocks. In the LEAP framework, scratchpads share the same interface as on-die RAM blocks and are plug-in replacements. Additional libraries support heap management within a storage set. LEAP scratchpads allow accelerator authors to focus more on core algorithms and less on memory management, analogous to software development.

HAsim uses LEAP scratchpads to aid in scaling the simulator to larger multicore targets. The data required to model the caches of the processors does not fit on-board the FPGA. We employ scratchpad memories to store modeled cache states, as discussed in Section 5.6

2.5.2 LEAP Remote Request-Response (RRR)

LEAP provides a typed asynchronous request-response protocol called RRR (for Remote Request Response) to allow typed method-call-like communication between an FPGA and a software process. Similar to Remote Procedure Calls [6], the user defines services whose servers reside on either the FPGA or in software, with the client

```
service ISA EMULATOR
{
    server fpga <- cpu
    {
        method UpdateRegister(in REGINFO rinfo);
    };
    server cpu <- fpga
    {
        method Sync(in REGINFO rinfo);
        method Emulate(in IINFO iinfo, out IADDR newPc);
    };
};
```

Figure 2-4: Example of a LEAP RRR specification for instruction emulation.

residing at the opposite end. The user defines the interface exported by each server. An example interface is shown in figure 2-4.

At compile time, RRR stub compilers generate the marshaling, demarshalling and multiplexing code that plumb the user code into underlying LEAP communication channels. A system like RRR abstracts away almost all of the headache for communicating between an FPGA module and a software module. Most of the LEAP virtual services described earlier, as well as Scratchpads, are layered on top of RRR.

2.6 Further Contributions

The techniques presented in this section represent the application of good engineering practices in the context of FPGA-accelerated simulation. Beyond these, HAsim has developed two specific contributions aimed to address development effort: Soft Connections, a technique for increasing modularity in hardware description languages (Appendix A), and a novel architecture for implementing timing-directed simulation on an FPGA (Chapter 5).

2.7 Discussion

In this chapter we explored HAsim's techniques for offsetting the increased development time that comes with using FPGAs and hardware description languages. HAsim emphasizes plug-and-play modularity and code reuse, enabled by the Asim

Architect's Workbench. The LEAP virtual platform is a general abstraction and has found applications beyond architectural simulation.

The problem now becomes assembling such a model in a way that is able to take advantage of the fine-grained parallelism inherent in the FPGA. To this end we have created an abstraction called A-Ports, which allows for distributed simulation without the need for a centralized controller.

Notes

[3]It should be noted that high-level hardware description languages do not necessarily result in worse FPGA utilization. There are cases where high-level knowledge exposes optimization opportunities [26].

Chapter 3

A-Ports: Fine-Grained Distributed Simulation on FPGAs

3.1 Introduction

Section 1.2.1 demonstrated that separating the FPGA clock from the simulated model clock can result in significant benefits. The problem now becomes taking many modules—representing the various functions of a target processor—each of which takes a different number of FPGA cycles per model cycle, and composing them together in a manner that results in a consistent notion of model time. This section examines how to construct such a simulator while simultaneously enabling the FPGA to take advantage of the fine-grained parallelism available in the target design.

We begin by discussing exactly how a high-detail processor performance model is specified, including a discussion of how parallel software simulators control simulation. We then consider existing simulation techniques and discuss why they are unsuitable for the specific conditions of FPGAs. Finally we present A-Ports: a novel technique for controlling distributed simulation on an FPGA that removes all need for global coordination.

3.2 Latency-Delay Port Specifications

A Latency-Delay Port (LDP) specification is an existing technique for describing a high-detail, cycle-by-cycle model of a target hardware system. This type of specification is used to create the models for simulators such as Intel's Asim [20]. Asim's main goal is to allow architects to develop performance models quickly by reusing existing pieces. To encourage this, the specification of 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 structure known as *ports*, which formalizes the interface and helps separate concerns of timing from functionality. A port is simply a communication channel annotated with a static latency l, representing the amount of model cycles messages take to flow through the port.

In Asim modules are arranged into a directed graph connected by ports. This graph must obey the following rules:

- Each port has a single writer and reader.
- Latencies are statically specified and may not change dynamically.
- The modules may have local state, but may not access each other's state directly—all communication must go through ports.

Each module must now define its "cycle-by-cycle" behavior. This is done by specifying a method called `Clock()`. This method takes as a parameter the current clock cycle, held in a variable `current_time`. A module's `Clock()` method generally does the following:

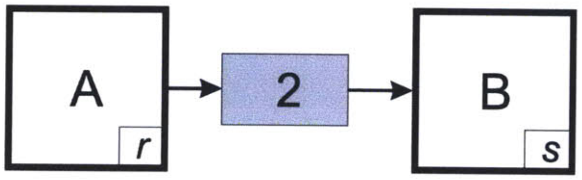
- Queries the module's input ports to determine if they contain any messages which arrive at `current_time`.

- Performs all necessary computations and local state updates based on any messages it received (and its current local state).
- Possibly send messages into its output ports. It may send at most one message on each output port.

Note that the modules themselves have no inherent notion of model time — we can consider their computation to be infinitely fast. Time is represented only in the delay of communication between modules (and in each module's decision whether or not to enqueue a message, as we will see in Section 3.2.4). The ports themselves behave as follows:

- Each port has a method named `HasMessage()` to query if an message arrives at `current_time`.
- Because at most one new message may be added every cycle, at most one message will arrive each cycle. The content of messages may be accessed with a `Receive()` method.
- The `Send()` method uses the port's latency l to record that the message will appear on cycle `current_time`$+l$.
- At the conclusion of cycle `current_time`, all messages arriving on that cycle are deleted—and thus are lost if the receiving module did not observe them.

A simple LDP system is shown in Figure 3-1. The system consists of two modules, A and B. Module A has a local state variable r while B has s. Both variables have an initial value of zero. Each module defines a `clock()` method that describes their cycle-by-cycle behavior. A always increments r, but only sends a message when r is even. B checks for a message, and increments s by that amount if one is present, and by 1 otherwise.



```
port AtoB 2;

module A
    var r = 0;
    method Clock()
        if (r[0] == 1)
            AtoB.Send(r);
        r = r + 1;

module B
    var s = 0;
    var tmp = 0;
    method Clock()
        if (AtoB.HasMessage())
            tmp = AtoB.Receive();
            s = s + tmp;
        else
            s = s + 1;
```

Figure 3-1: Simple example of a complete LDP specification.

Model Cycle	Module Clocked	r	$AtoB$	s
-	-	0	-	0
0	A	1	-	0
0	B	1	-	1
1	A	2	(3, 2)	1
1	B	2	(3, 2)	2
2	A	3	(3, 2)	2
2	B	3	(3, 2)	3
3	A	4	(3, 2), (5, 4)	3
3	B	4	(5, 4)	5
4	A	5	(5, 4)	5
4	B	5	(5, 4)	6

Figure 3-2: Sequential simulation of 5 cycles of the system shown in Figure 3-1

3.2.1 Simulation Model for LDP Specifications

Given an LDP specification of a system, the problem of simulating the system can be reduced to the *dynamic snapshot* problem:

- Given a specification in state s and input i, what is the state of each module in the system at time t?

The execution model for simulating an LDP Specification is to iterate through each module in system, calling clock methods:

```
modelcycle = 0;
while (1)
    foreach M in Modules
        M.Clock(modelcycle);
    foreach P in Ports
        P.dequeue(modelcycle);
    modelcycle++;
```

In order to determine whether or not the ports in the system contain a message on any given cycle, messages in the port are stored as tuples `(receiving_time, msg_val)`, where `receiving_time` equals the clock cycle the message was sent, plus the latency of the port. Figure 3-2 shows simulating 10 cycles of the system previously described in Figure 3-1.

3.2.2 Latency-Delay Port Models of Systolic Pipelines

Latency-Delay Port specifications can be easier to write than a traditional hardware specification if the system contains *systolic* pipelines. To see why, let us construct an LDP specification for the multiplier pipeline of an ALU. Figure 3-3A shows the specification of the *functionality* of the multiplier: it simply takes two numbers and multiplies them together by applying a function called *mul*. The advantage of this approach is that it is simple to describe and easy to understand.

Figure 3-3B shows an implementation of the multiplier as a 4-stage pipeline in a traditional hardware description language (assuming $mul = mul_1 \bullet mul_2 \bullet ...$). This

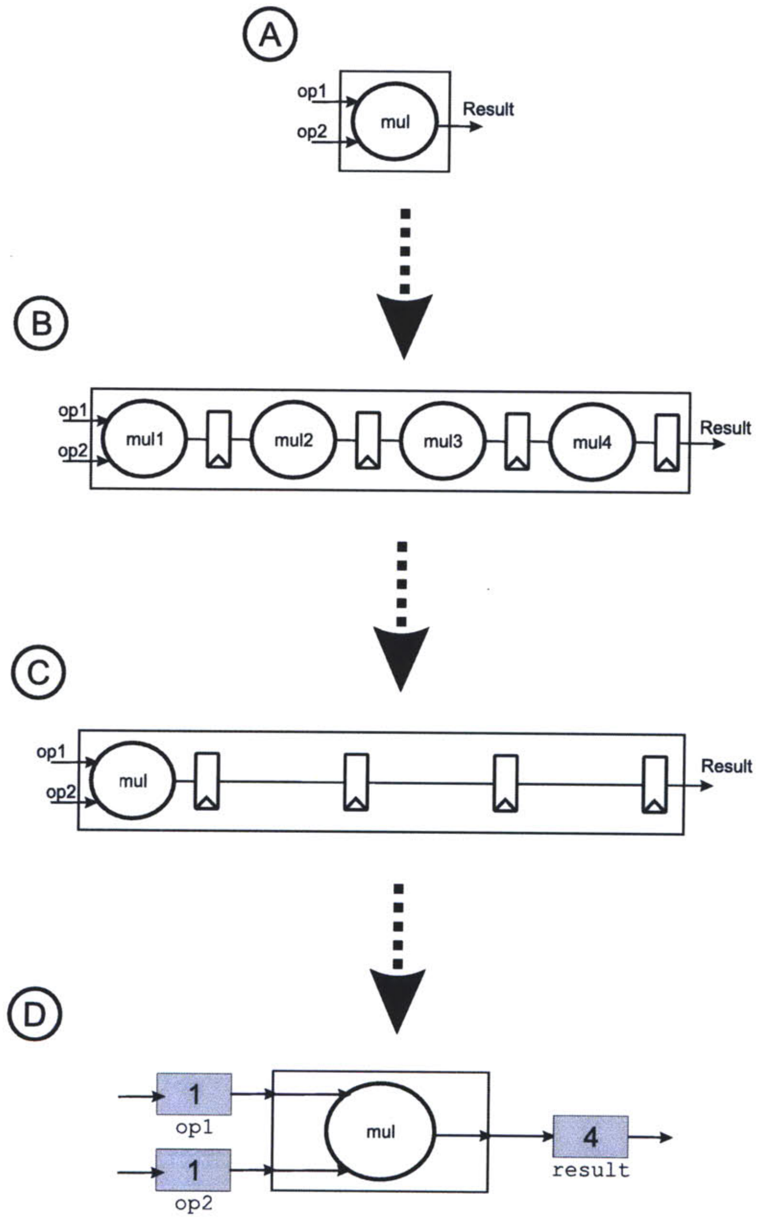


Figure 3-3: (A) Functionality of a systolic multiplier. (B) Traditional hardware description of the multiplier can be tedious and hard to change. (C) Separating the functionality from the timing of the systolic pipeline. (D) Latency-Delay Port Specification recovers a large degree of the simplicity of the original functional specification.

description of the multiplier includes information on both the functionality of the operation, and the precise *timing* that the operation takes. Traditional structural hardware descriptions have several problems:

- Tedious to describe because they require verbose descriptions of each pipeline stage.
- Hard to change, because adding or removing a pipeline stage requires changing the logic functions of every stage.
- Confuses code that is required for functionality, from code that is required for timing.
- Does not differentiate between state which is necessary for functionality, and state that is introduced only to accommodate timing.

One solution to this problem is to *re-time*[4] the description of the circuit, as shown in Figure 3-3C. This description is much simpler, as the functionality of the multiplication operation is separate from their timing. This also makes the timing easier to change, and thus enables a certain class of architectural exploration (assuming that the system around the multiplier is tolerant to these changes in latency).

An LDP specification of the multiplier is shown in Figure 3-3D. This approach retains the benefits of the re-timing approach, in that the functionality is separated from the timing. Additionally, it adds some new benefits:

- Local state which is stored inside the ports is clearly not relevant for functionality, but only for timing.
- Changing latency involves only changing a single number, rather than adding new registers.
- Provides a clean separation between code for timing and code for functionality.

One criticism of the LDP specification is that systolic pipelines are relatively rare in detailed microprocessor descriptions. Although this is true of the final system, latency-delay ports have proved to be a useful *abstraction* for quickly describing long-latency communication. For example a "magic" memory system that is simply a latency delay can be modeled using a RAM and a port. This enables a certain class of experiments simply by varying the latency of the memory.

3.2.3 Representing Parallel Pipelines as Multiple Ports

Now let us suppose the architect wants to expand the multiplier to the simple ALU shown in Figure 3-4A. In this ALU all instructions are classified as either multiplications or simple arithmetic operations. The desired structural implementation of the ALU is shown in Figure 3-4B. Simple arithmetic operations are expected to take 2 cycles, while multiplications take 5.

An LDP specification of the module is shown in Figure 3-4C. Here, we have added a new port to represent the arithmetic pipeline. The ALU will enqueue at most one message into either of the output ports, depending on the type of instruction it receives.

Notice that this representation says nothing about what happens if the next stage receives a result on both ports on the same cycle. This could result in an error if the register file does not have sufficient ports to perform both writebacks. Presumably, the Issue stage includes logic to ensure that writeback collisions do not occur. One advantage to the LDP specification is that it allows the Issue stage's code to query the latency of the various ports, allowing more concise description of collision detection.

3.2.4 LDP Specifications of Non-Systolic Operations

For operations which are not implemented as systolic pipelines, a simple latency is not sufficient to describe the timing of the system. In this case the latency must be combined with *whether or not* a message should be sent, which is determined by the local state of the system.

Suppose we want to add a divider to our ALU. The functionality of the divider (Figure 3-5A) is simply added to our specification. However, the timing of the divider

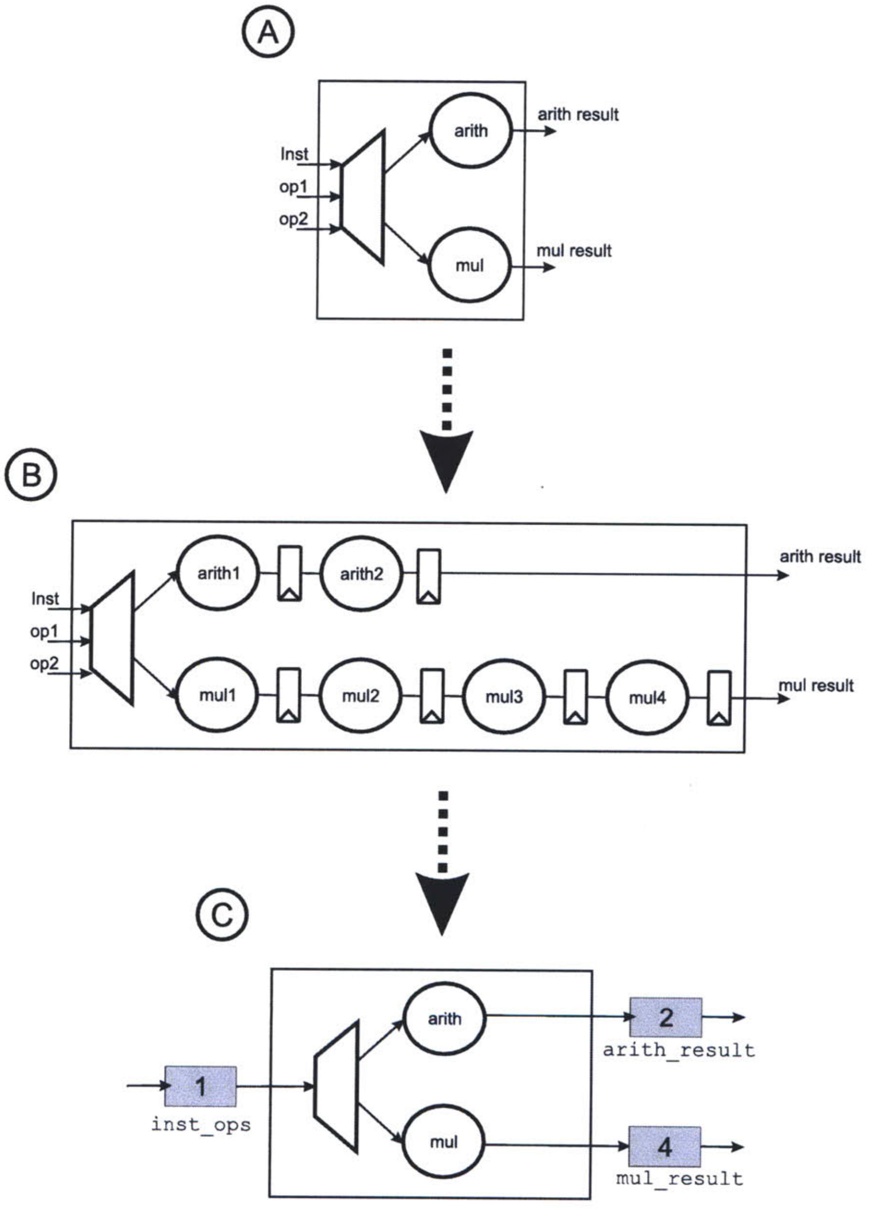


Figure 3-4: (A) Functionality of ALU with separate arithmetic pipeline. (B) Traditional hardware description of the parallel pipelines. (C) Latency-Delay Port Specification representing the parallel pipelines as separate ports.

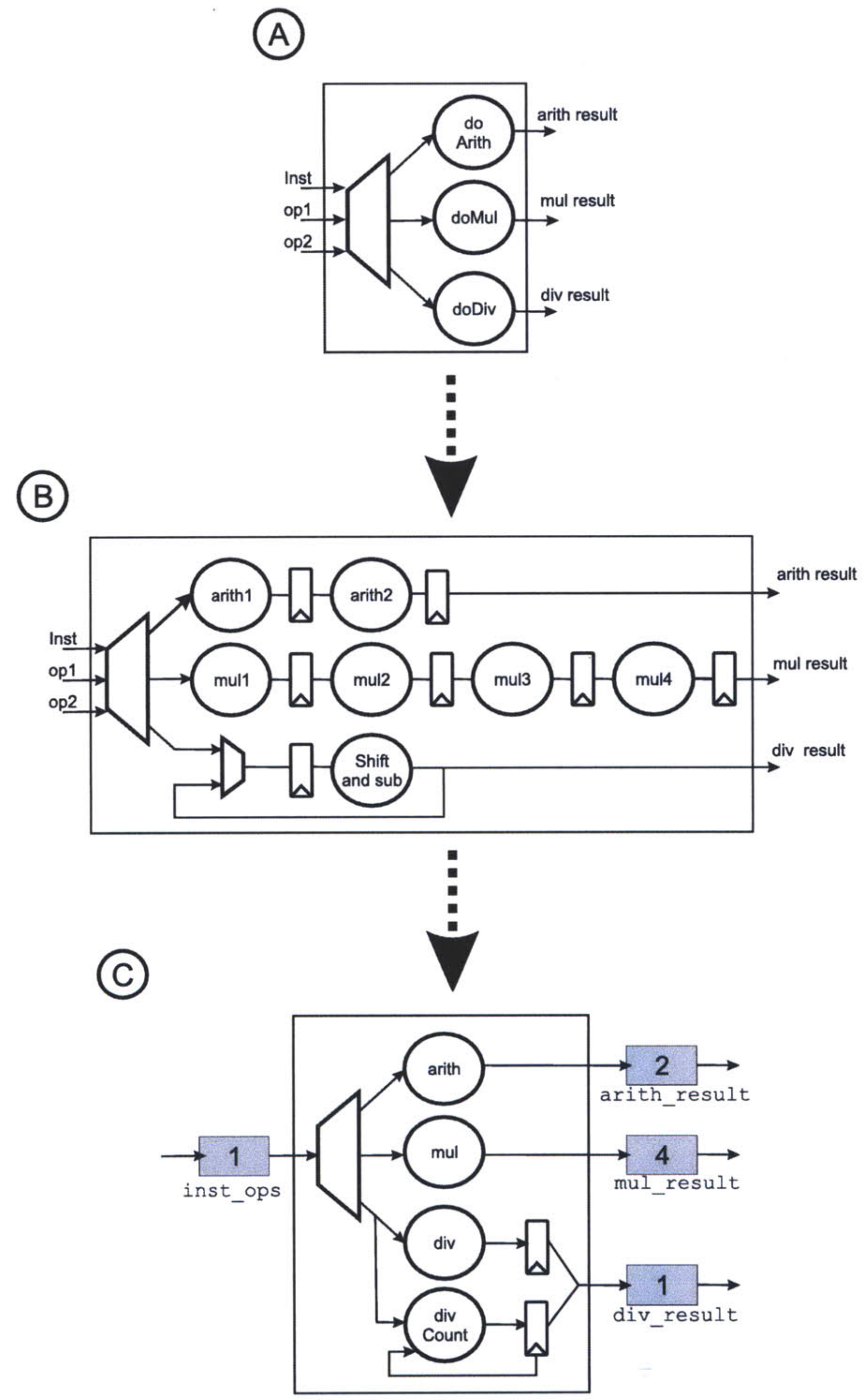


Figure 3-5: (A) Functionality of ALU with division pipeline. (B) Implementation of the division operation as a circular pipeline. (C) LDP Specification uses separate logic to calculate the division's result from the amount of cycles consumed.

is more complex. Because the architect expects that divide operations are rare, she is considering implementing them with a circular shift-and-subtract. (Let us assume the issue stage knows not to place more than one divide instruction in flight simultaneously.) The structural description of the shift and subtract (Figure 3-5B) iterates the operation through the pipeline a number of times that is *dynamically* dependent on the input data to the ALU. Again, this confuses issues of timing with functionality.

In contrast, an LDP model of our circular divider is shown in Figure 3-5C. In this model the functionality of doing the division operation (`div`) is separated out from the timing. However, we cannot simply send the result on an output port, as the latencies of ports are fixed statically. Instead, the result is *delayed* in local state. A separate operation (`divCount`) sets a counter with the number of iterations that the shift-and-subtract pipeline would take to calculate the result. The port itself is assigned a static latency of 1. Therefore, if the `divCount` operation decides that a particular operation should take 5 clock cycles, then the result of `div` is sent in the output port 4 clock cycles later, so as to arrive on the 5th cycle.

In this way, the LDP specification is able to retain separation of timing and functionality, but unlike in the systolic case the description of the timing is not as easy as placing a message into a port. Similar techniques can be applied to make a port (which is fully pipelined) simulate a communication channel which is not fully pipelined.

3.2.5 Complete LDP Specifications

Figure 3-6 shows a complete target processor recast as a port-based model. 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 example.

It is important to note that the graph of modules and latency-delay ports is *not* a complete specification of the timings of the system. As with the circular divider above,

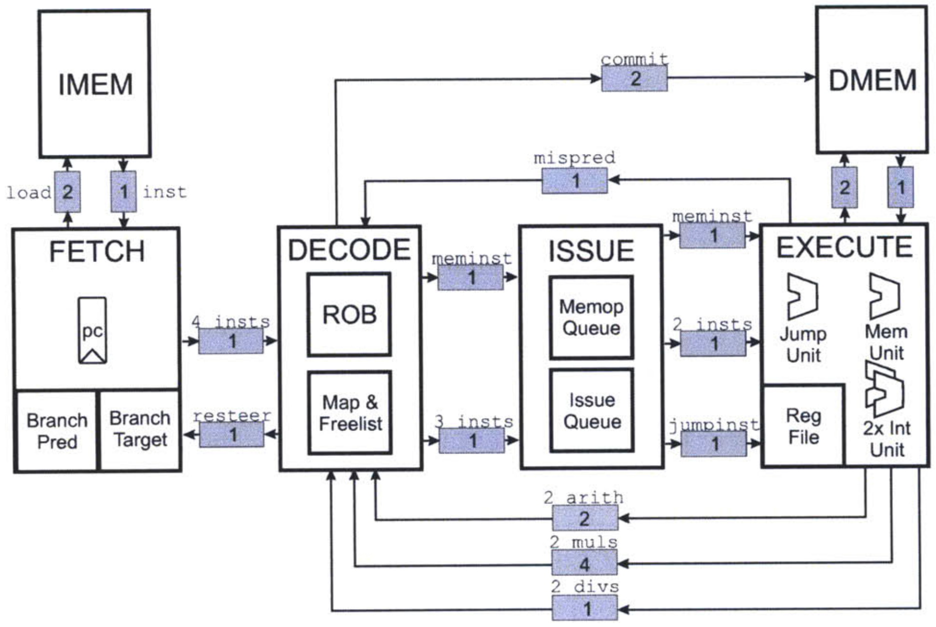


Figure 3-6: LDP Specification of out-of-order, superscalar processor.

a complete specification must also include code for each modules' local behavior, as this determines *if* these modules send a message for a given cycle.

For example, the ROB in Figure 3-6 is connected to the Issue stage by a port of latency 1. However, an instruction may be resident in the ROB for dozens (or maybe even hundreds) of cycles before being issued. In fact, wrong-path instructions may be dropped from the ROB before ever being issued at all. The only information that the LDP graph conveys is that *when* an instruction does leave the ROB, it arrives at Issue 1 cycle later.

In aggregate, the precise timings of any given instruction are dependent on the instruction itself, the local state of the ROB and other modules, and the static latencies of the ports. The only way to determine this is to simulate the system in full.

3.2.6 Considerations for Zero-Latency Ports

It is legal to use zero-latency ports in an LDP Specification, so long as those ports are not arranged into "combinational loops" — a familiar restriction to hardware designers. Formally, every cycle in the graph must contain at least one port of latency 1 or more.

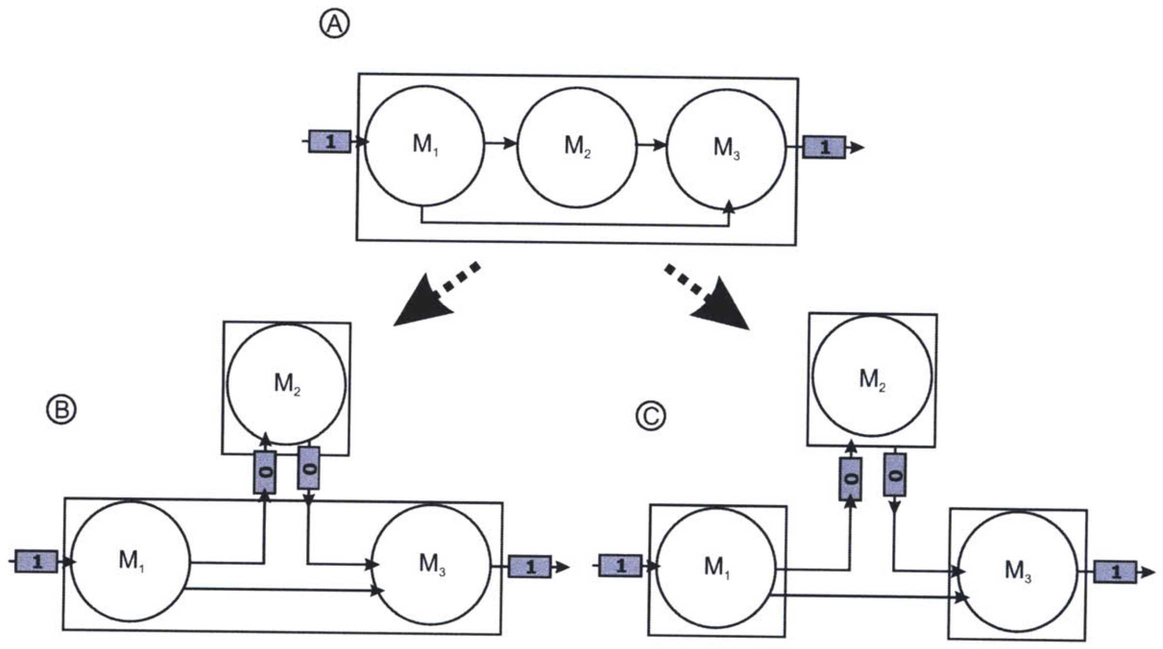


Figure 3-7: Re-cutting a module with zero-latency ports. (A) Original module with three sequential operations. (B) Illegal re-cutting with a "false" loop. (C) Legal re-cutting removes the false loop.

Because of this restriction, a situation can occur whereby it can seem that the LDP specification is fighting against modularity. To illustrate this, consider the example shown in Figure 3-7. The original module contains three sequential operations M_1, M_2, and M_3. The designer wishes to place M_2 into a separate module, perhaps in order to facilitate swapping in alternative modules. Unfortunately, doing this naively results in a loop of zero-latency ports, as shown in Figure 3-7B. However, clearly this represents a case where the modeler was describing legal hardware: a false loop rather than a true combinational loop.

In this case the solution is to *re-cut* the graph so that the original is now three modules, as shown in Figure 3-7C. This allows M_2 to be isolated, as intended.

Re-cutting also applies when the module contains local state that is read by different operations. Figure 3-8A shows a more realistic example of a Fetch unit. The original module has the following functionality:

1. Receive incoming credits from the instruction queue.

2. If we have 1 or more credits, send current PC to the line predictor and send the

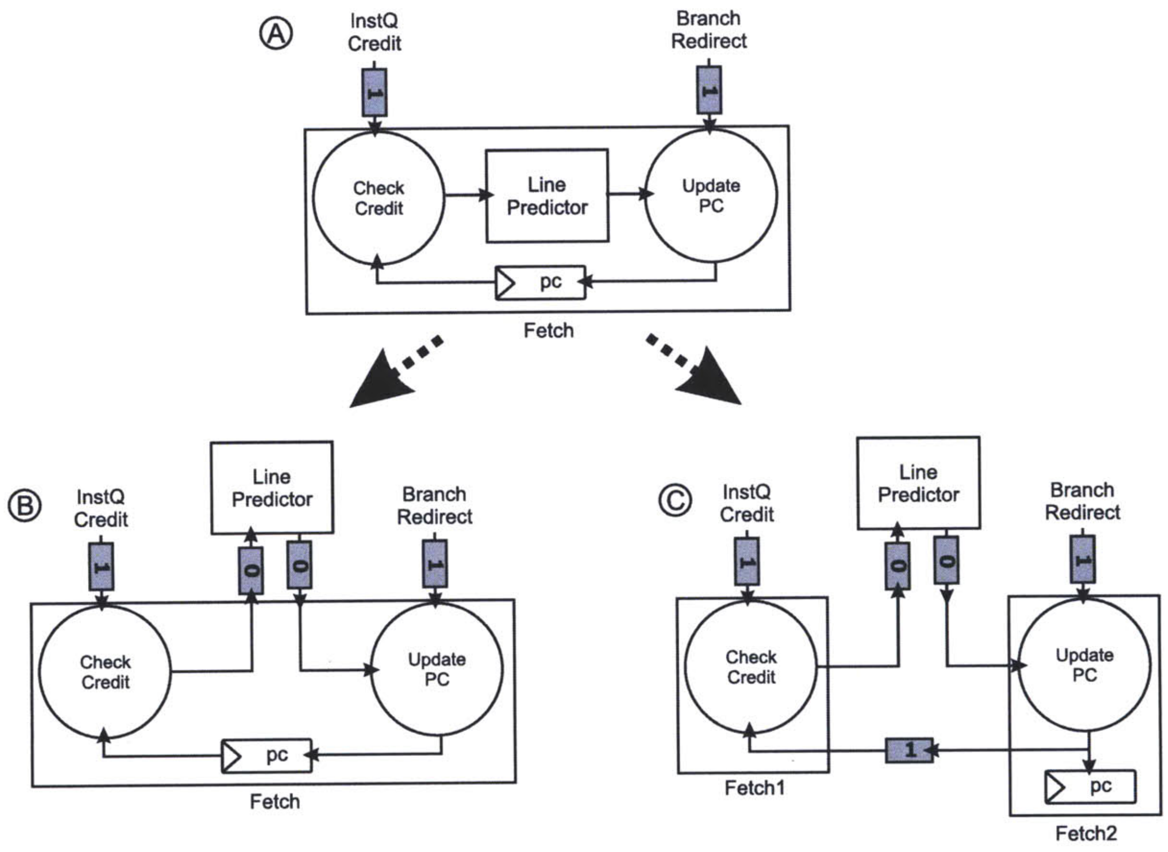


Figure 3-8: Re-cutting a module which includes local state. (A) Original fetch module. (B) Illegal re-cutting. (C) Legal re-cutting with transmission of new state to first stage.

prediction to the ITLB.

3. If we receive a redirect from the back end, write that to the PC, otherwise write the line prediction to the PC.

The architect wishes to move the line predictor out of the Fetch unit, in order to facilitate "plug-and-play" replacement of different line prediction schemes. Unfortunately, this would create a loop of the zero-latency ports (Figure 3-8B). The re-cutting solution now must include a backwards path which carries the updated value of the program counter, as shown in Figure 3-8C). This backwards path must not be zero-latency, and must be configured to contain the initial value of the program counter at simulator startup.

An alternative approach is to change the module simulation semantics to query input ports in a fixed dependency order. This is the approach taken by LI-BDNs, as we discuss in Section 3.7.4. However, we believe the re-cutting solution is sufficient, as there is guaranteed to be at least one legal cut of any hardware system that does not contain combinational loops.

Finally, the execution model of the simulator must be updated to deal with zero-latency ports. Originally, the order that we iterated through the modules in the system was unimportant. If the system contains zero-latency ports, then they must be done in causal order. 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 non-zero-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.

3.3 Parallel Simulation of LDP Specifications in Software

The Asim simulator is a straightforward implementation of the sequential latency-delay port simulation model. This results in the simulator performance shown in Figure 3-9, whereby modules are simulated one-at-a-time on the host computer.

Within a target processor there is a large degree of fine-grained parallelism, as all modules that are not connected (or transitively connected) by zero-latency ports may be simulated in parallel. As the unit of parallelism represented a few instructions, the implementors of Asim were concerned about whether this approach could be exploited to improve simulation rate, as the overhead of running multiple threads on a parallel *host* computer would overwhelm this granularity of parallelism.

However, when simulating a multicore *target* computer the target cores themselves may be partitioned into separate threads, and the threads run in parallel. This

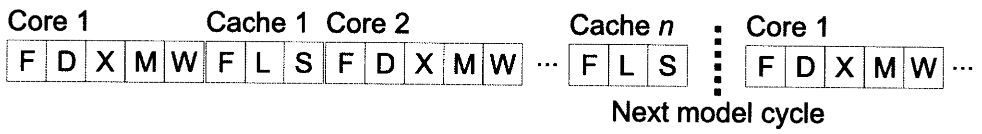


Figure 3-9: Sequential Simulation Scheme of LDP Specification

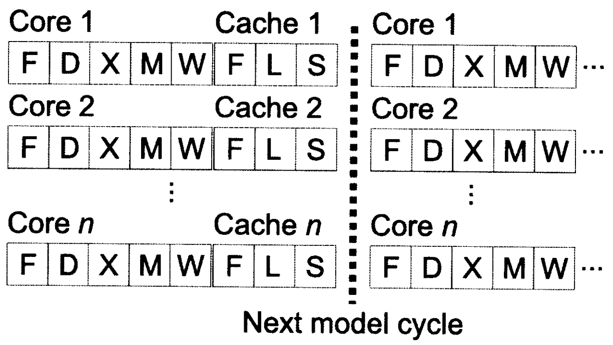


Figure 3-10: Parallel Simulation Scheme of LDP Specification on Multicore Host

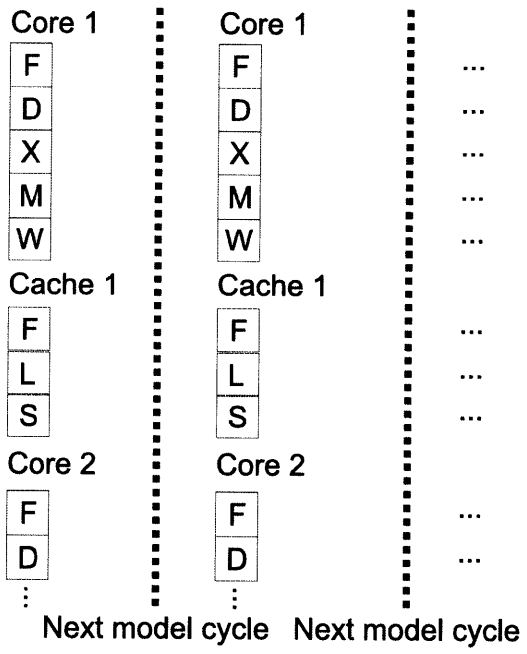


Figure 3-11: Parallel Simulation Scheme of LDP Specification on FPGA

degree of parallelism is much more suited to running on a multicore host. This led to Parallel Asim, an implementation where the centralized clock server runs in a thread, and uses barrier synchronization to coordinate between a separate simulation threads, as shown in Figure 3-10. The modules of the target are statically partitioned between the threads (usually one-to-two target processors and their associated caches are assigned per thread). The threads advance a model cycle, and stall on a barrier when complete. Simulation of the on-chip network (the OCN, or so-called *uncore*) is handled by a dedicated thread. This is because the OCN represents communication between the various threads, and so must be synchronized to the barrier. This is discussed in-depth in Chapter 4.

Assuming that the user has defined a partitioning of modules to threads, the resulting simulation algorithm can be described as follows:

```
modelcycle = 0;
Threads = partition(Modules);
while (1)
    parallel foreach T in Threads
         parallel foreach M in T
             M.Clock(modelcycle);
    wait_for_barrier();
    modelcycle++;
```

Barr [3] had earlier 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 controller-less 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 that 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, as shown in Figure 3-11. The problem now becomes coordinating when the distributed parallel modules can advance to the next model cycle. This will require additional hardware overhead to perform this coordination. In the next section, we explore the overhead incurred by existing parallel simulation techniques.

3.4 Applicability of Existing Distributed Simulation Techniques to FPGAs

In this section we discuss various existing simulation techniques with the goal of exposing as much parallelism as possible when implementing an LDP specification on FPGAs. We compare these techniques to each other in Figure 3-12 and refer to this figure throughout this section. The goal is to find a distributed simulation technique that maximizes simulator performance minimizing the overhead in terms of FPGA resource utilization. The technique must not introduce any errors into the simulation results.

3.4.1 Correctness Issues of Modeled Clocks

As a performance model is a simulation rather than a direct implementation, we must be concerned with both the correctness of the target specification, and of the simulator's implementation. 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,

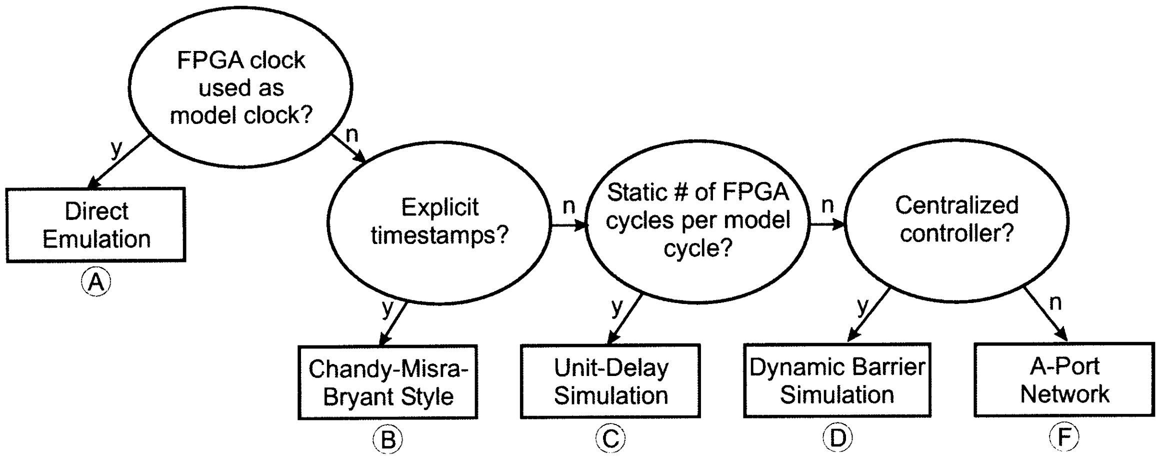

FPGA clock used as model clock?
y
n
Direct Emulation
A
Explicit timestamps?
n
y
Chandy-Misra-Bryant Style
B
Static # of FPGA cycles per model cycle?
n
y
Unit-Delay Simulation
C
Centralized controller?
y
n
Dynamic Barrier Simulation
D
A-Port Network
F

Figure 3-12: Overview of simulation techniques for FPGAs.

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 preceding 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 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.

3.4.2 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 3-12.

As discussed in Section 1.2.1, if the target ASIC employs structures that do not map well onto FPGAs (e.g., multi-ported register files, or content-addressable memories) then the resulting FPGA clock period is likely to be poor, and use a lot of resources, as shown previously in Figure 1-3. We include this technique only for completeness.

A better approach is to use a distributed simulation technique that separates the model clock from the FPGA clock. Classically, such existing 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).

3.4.3 Simulation with Explicit Timekeeping

Distributed simulation techniques that explicitly carry time are variants of the Chandy-Misra-Bryant simulation technique [10, 8], Node B in Figure 3-12. 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 processor 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. Additionally, modules which have changes to internal state triggered autonomously without external events (such as counters) need to have an activation mechnasim provided to them. It is significant to note that the major performance models written in software use continuous simulation techniques rather than event-driven techniques.

3.4.4 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 simulation* (Node C of Figure 3-12), historically used in projects such as the IBM Yorktown Simulation Engine [50]. 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. As the FMR=n, 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 3-12), where all modules coordinate dynamically on when to move to the next model cycle. As

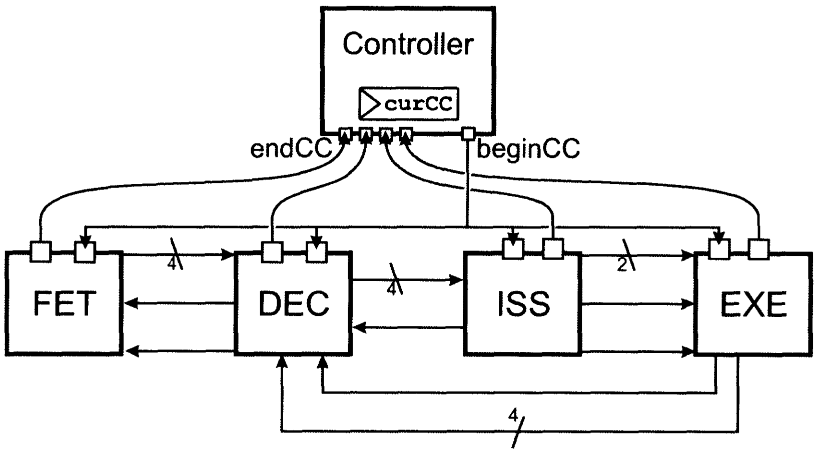
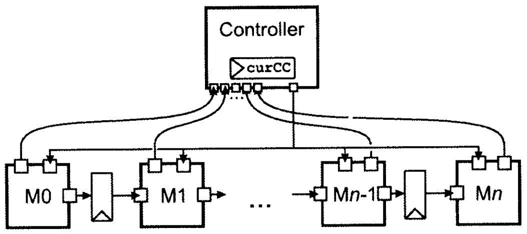


Figure 3-13: Dynamic barrier synchronization with centralized controller.

is shown in Figure 3-13, 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 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



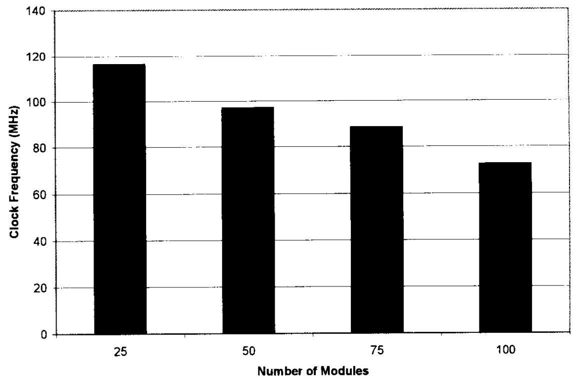


Figure 3-14: Dynamic barrier synchronization's centralized controller limits scalability.

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 3-14. 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 number 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 developed for HAsim specifically 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% in FMR over dynamic barrier synchronization using the A-Ports scheme.

3.5 A-Port Networks

As explained in Section 3.2, Asim performance models are specified using an explicit representation of time and implemented using a centralized controller to coordinate simulation. As we noted in Section 3.4.3, 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.

3.5.1 Developing a Distributed Simulation Scheme

As shown in Figure 3-15, a simulation of a latency-delay port specification can be viewed as a Kahn process network [29]. The initial placement of tokens is derived from the latencies of the ports themselves (n tokens are placed on a port of latency n). 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 k a node is

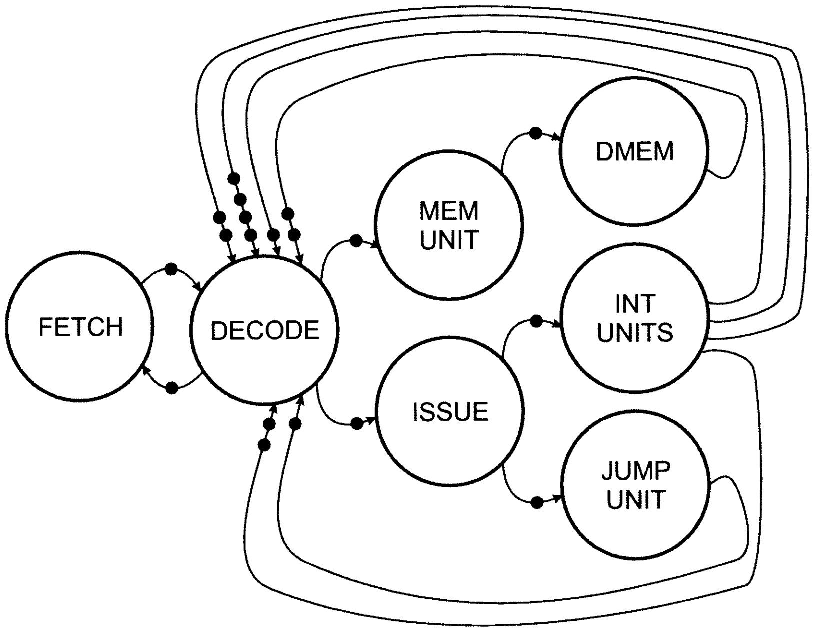


Figure 3-15: An A-Port Network is a restricted Kahn process network.

simulating by counting the number of times it has executed this simulation loop. Thus the A-Ports scheme (Node E in Figure 3-12) 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 3.6.

In order to accommodate this restriction we must change the semantics of classical Asim ports. As described in Section 3.3, 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 default initial tokens in the system.) Thus the complete distributed simulation loop is as follows:

1. 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.

2. 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.

3. One message is consumed from each incoming A-Port and the loop repeats.

3.5.2 Simulator Slip

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. This is called simulator *slip*. A producer may run into the future, pre-computing values as fast as possible. Similarly, a fast consumer can drain its input buffers.

Interestingly, observing the state of the A-Port itself is sufficient to determine the relationships of its producer and consumer. Observe:

- On any given FPGA cycle, if an A-Port of latency l contains l elements, the modules it connects are simulating the same model cycle.

- If the A-Port contains more than l elements, the producer module is simulating into the future compared to the receiving module.

- If the A-Port contains fewer than l elements, the situation is reversed.

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.

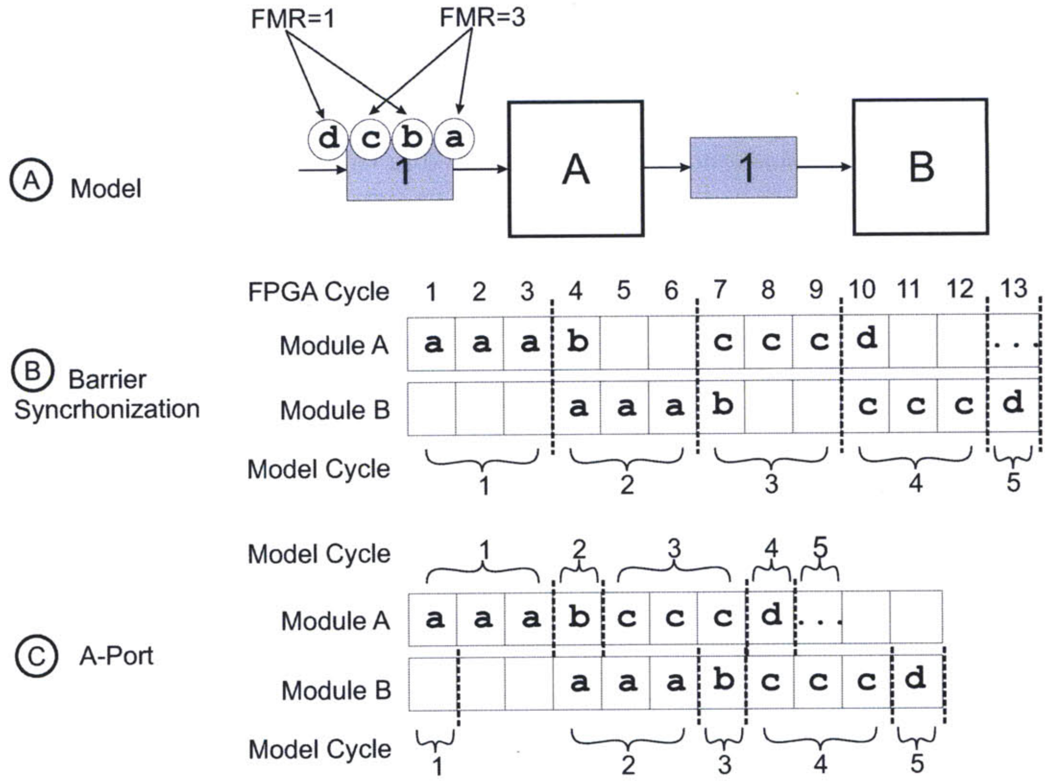


Figure 3-16: Demonstrating how an A-Port implementation can result in a performance improvement over barrier synchronization.

This slipping does not alter results of simulation, but it can improve simulation rate over barrier synchronization, as demonstrated in Figure 3-16. In this example, instructions `a` and `c` take more FPGA time to simulate compared to `b` and `d`. Observe that on FPGA cycle 6 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 show in Section 3.6.

3.5.3 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 before enabling the result capture. Similarly, such a rebalancing scheme should not rely on an expensive global communication network.

To this end, we have designed a distributed protocol that modules may use to resynchronize a slipped system. This involves changing rule 1 of the simulation scheme above to the following:

1. 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 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. Since the graph is connected, any module that 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 3-17 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 3-18 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 outputs, 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, 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.

3.6 Implementing A-Port Networks on FPGAs

As shown in Figure 3-19, we implement an A-Port of message type t as a FIFO of $sizeof(t) + 1$ 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 buffer-

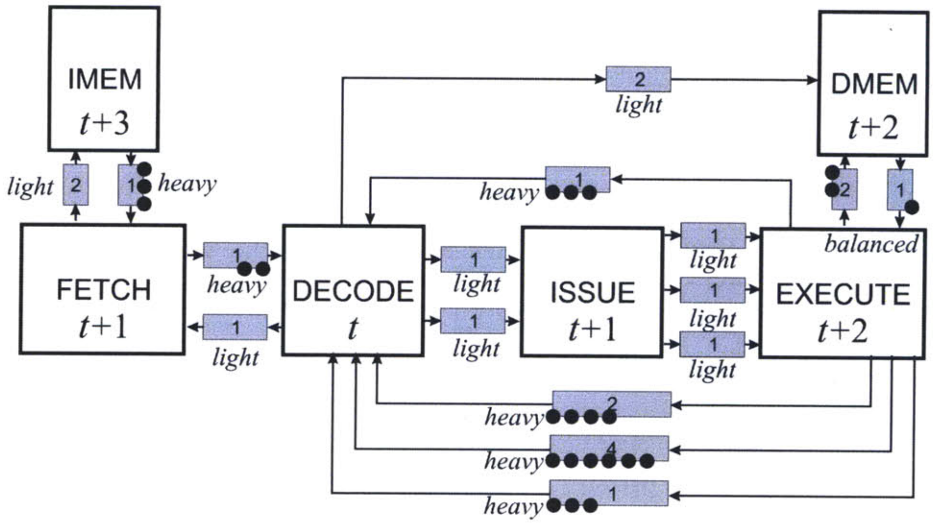


Figure 3-17: Obtaining a consistent snapshot from a slipped state.

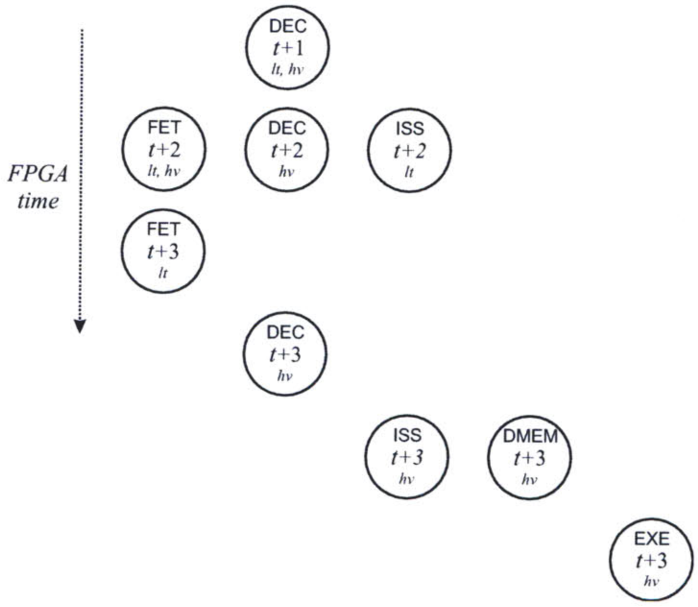


Figure 3-18: Execution order to quiesce Figure 3-17.

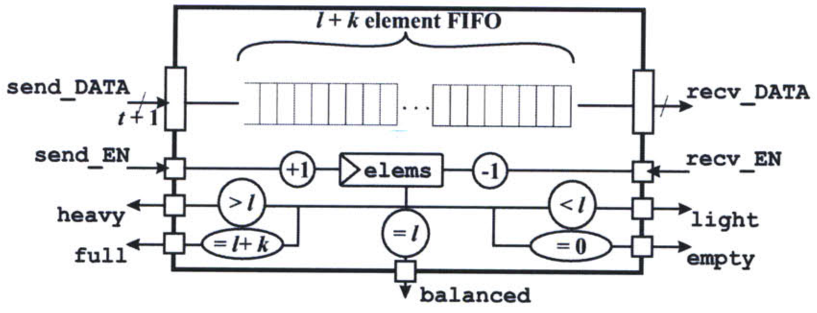


Figure 3-19: A-Port implementation on FPGAs.

ing. 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.

3.7 Related Work

3.7.1 Synchronous Dataflow

The conditions for initial token placement in an A-Port Network are closely related to the correctness conditions of Lee's [35] static synchronous dataflow graphs. 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 modeled synchronous system, rather than requiring the user to determine buffer sizes or placement of tokens manually.

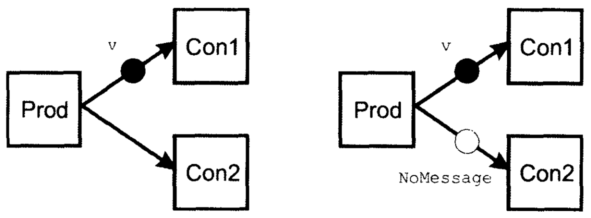


Figure 3-20: In A-Port Networks, the NoMessage value is used in place of not sending a message.

3.7.2 Process Networks and the NoMessage Value

As already noted, an A-Port network is a restricted case of a general Kahn process network [29], 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 Commoner's marked directed graphs [16]. As shown in Figure 3-20, 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 [10]. 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 [35]. 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.

3.7.3 Latency-Insensitive Design

The theory of latency-insensitive design developed by Carloni et al. [9] 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.

3.7.4 Latency-Insensitive Bounded Dataflow Networks

Latency-Insensitive Bounded Dataflow Networks (LI-BDNs) are an abstraction created by Vijayaraghavan and Arvind [57] as an expansion of A-Port Networks and Vijayaraghavan's work on HAsim. This work adds three main contributions over A-Port Networks.

- The LI-BDN technique takes a specification of a circuit as a synchronous state machine (SSM), rather than an LDP specification. The technique then automatically transforms the SSM into an asynchronous specification suitable for implementation on FPGAs. This resulting specification shares some properties with LDP specifications, as it allows users to then refine modules by hand so that they take more FPGA cycles to simulate one model cycle. This can help development effort because the user can focus on modules that need to be optimized for FPGAs and leave others to automatic transformation.

- LI-BDNs define specific conditions that guarantee that the dataflow network avoids temporal violations and temporal deadlock. One restriction, called *no extraneous dependencies*, states that each module should only read its input ports as late as possible when simulating a model cycle. This allows LI-BDNs to simulate descriptions that have "combinational loops" of dependencies, where these dependencies are actually false dependencies in the target circuit. In an A-Port Network, the solution would be to "re-cut" the module boundaries (Section 3.2.6). Although a legal re-cut is always guaranteed to exist, LI-BDNs make it so recutting is not necessary.

- LI-BDNs transform synchronous state machines into *self-cleaning* asynchronous dataflow representations. This restriction ensures that tokens are only produced if they will be consumed later. This relaxes the A-Port restriction of reading/writing every A-Port exactly once for every model cycle, and allows LI-BDNs to coordinate simulation without a `NoMessage` value. Although this does not add expressivity to the specification, it does allow for more flexibility in the implementation.

Since LI-BDNs have been developed, HAsim has incorporated some of these insights into the way the simulator uses A-Port Networks, in particular using the relaxed dependency checking to deal with false loops.

3.8 Discussion

In this chapter we explored FPGAs as a platform for executing cycle-accurate performance models. We discussed how performance models are created in software and why contemporary multicores 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 chapter 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. In Chapter 6 we demonstrate that this technique can lead to an average improvement in simulation rate of 19% over traditional dynamic barrier synchronization.

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.

Notes

[4]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.

Chapter 4

Time-Multiplexed Simulation of Multicores and On-Chip Networks

In this chapter we extend the techniques presented in the previous chapters to simulating multicore processors paired with realistic On-Chip Networks (OCNs). Using FPGAs to simulate such a multicore immediately presents the challenge of fitting the target system into the FPGA's capacity. Experience has shown that directly configuring the FPGA into multiple cores, caches, and interconnect is too expensive to be useful. (We explore a specific example of this in Section 6.3.1). Given this, *time multiplexing* is a technique that can help enable scaling our models to larger multicores.

In a time-multiplexed scheme a single *physical* core is used to sequentially simulate several *virtual instances* that flow through the pipeline in sequence, as shown in Figure 4-1. Internal core state such as the program counter (PC) or register file (RF) is duplicated, but the combinational logic used to simulate each pipeline stage is not.[5]

The disadvantage to time-multiplexing is that it can reduce simulation rate, as a single physical pipeline is being used sequentially to do the work of many. In HAsim, the performance reduction from time-multiplexing is minimized. Chapter 3 established that on any given FPGA cycle module in HAsim may be *utilized*, meaning it is doing useful work for the simulation of a model cycle, or *un-utilized*, meaning that it is unable to simulate the next cycle—either because an input port is empty,

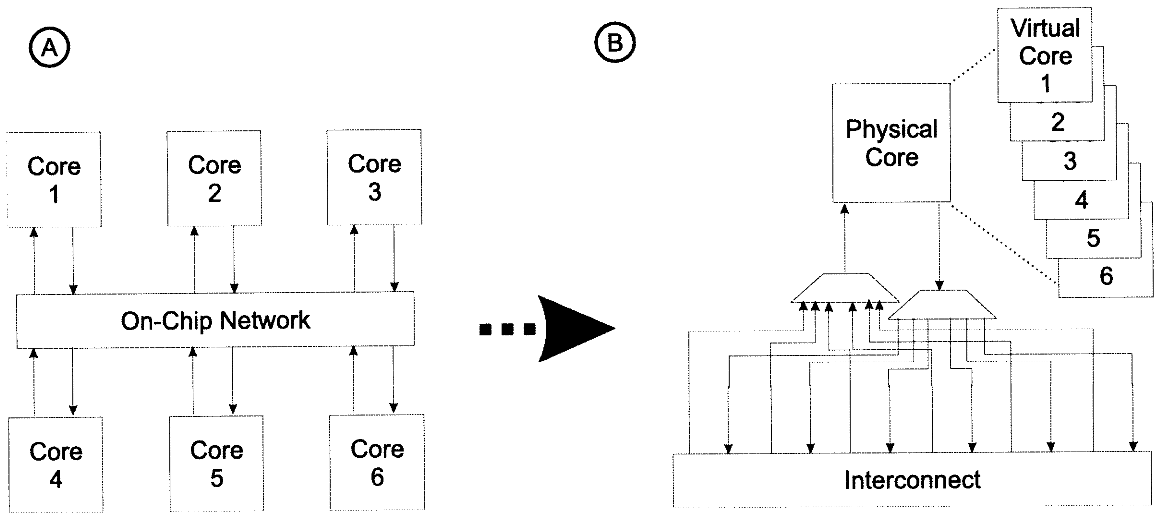


Figure 4-1: Modeling a multicore and OCN via (A) direct implementation (B) time-multiplexed core.

or because an output port is full. We found that in a typical HAsim configuration modules were utilized on an average of 13% of FPGA cycles over the course of a benchmark run. This is because simulation rate was limited by accesses to off-chip memory. In this situation, if we chose to duplicate the core models we would be paying for modules that would often be unable to proceed to the next model cycle because one of the cores was waiting for an off-chip response.

The time-multiplexing approach was first used in the Protoflex simulator [15] (Section 1.3.7). Protoflex multiplexes a functional model between 16 threads, but does not support any timing model on the FPGA. RAMP Gold [54] (Section 1.3.9) is another FPGA-accelerated simulator that uses a coarse-grained approach whereby a scheduler chooses a virtual instance to simulate, and performs the functional emulation of that instance without adding any timing model of the core. RAMP Gold does support timing models of caches, but does not currently support simulations of on-chip networks.

In this chapter we address these two problems. First, we modify HAsim's A-Ports scheme to time-multiplex the core pipeline. HAsim's scheme is unique in that the fine-grained nature means that different pipeline stages can be simulating different cores. We then extend this scheme to the simulation of a realistic on-chip interconnect via a novel use of permutations. We generalize our technique and extend it to cover heterogeneous topologies. HAsim is the first FPGA-accelerated simulator to allow for the modeling of realistic on-chip networks in a time-multiplexed fashion.

4.1 Fine-Grained Time-Multiplexed Simulation

4.1.1 Port-Based Multiplexing

A major contribution of HAsim is to extend the multiplexing to detailed timing models of core pipelines. HAsim uses the A-Ports between modules to implement time-multiplexing. In order to accomplish this we simply change the A-Port initialization rule. At simulator startup a port of latency n is initialized with n message tokens from each virtual instance, as shown in Figure 4-2. Within a module, local state is duplicated. The simulation loop for each module becomes:

- When all incoming ports are not empty, they all contain messages of the same virtual instance. The module may simulate the next model cycle for that instance.
- Use the virtual instance identifier as an index to retrieve the local state for that instance.
- Any output messages it produces will be the output for that instance.

The original port-based simulation scheme (Section 3.5.2) allowed adjacent modules to be simulating different module clock cycles. In the time-division multiplexed scheme, adjacent modules may now also be simulating different virtual instances. This helps keep modules busy, as they are more likely to have work in the queue if an up-stream module encounters a rare-but-slow event.

The round-robin ordering can be overly-restrictive as it can result in head-of-line blocking. This means that a virtual instance that is stalled do to a slow event such as an off-chip memory access, cannot be bypassed even in the situation where other virtual instances are ready to simulate. One solution would be to allow a ready

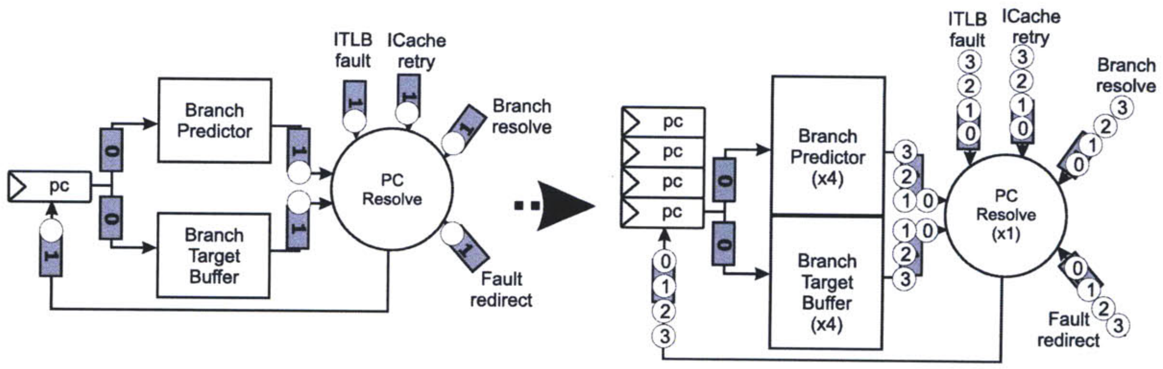


Figure 4-2: (A) Port-based model of a processors' PC Resolve stage. (B) Time-multiplexing between 4 virtual instances.

virtual instance to bypass an idle one at the head of the queue, but so far this scheme has proven too expensive to implement on an FPGA. We have found it to be more expensive than direct module duplication.

4.1.2 Pipelining the Modules

A more practical approach to minimize idle virtual instances is to pipeline the module implementation.[6] Under ideal cases pipelining the simulator modules can entirely eliminate the multiplexing performance penalty, achieving the performance of the original duplicated modules (similar to fine-grained multi-threading in an actual multiprocessor).

To understand why, consider the situation in Figure 4-3A. Module A is faster than module B, which has an FMR of 4 (4-3B). Multiplexing the system 4 ways without pipelining would decrease the overall FMR to $4 \times 4 = 16$ (4-3C). However, if module B can be pipelined into 4 stages, then overall FMR can be reduced back to the original 4 (4-3D). Note that the fourth stage of module B finishes each model cycle for the first core on the same FPGA cycle as the original non-multiplexed design.

Of course, such a scheme will rarely be operating under ideal conditions, and pipeline bubbles will be introduced. Such bubbles can be minimized if there are always ready virtual instances waiting for simulation. Thus a key to achieving simulator performance is keeping module pipeline depths less than or equal to the number of virtual instances.

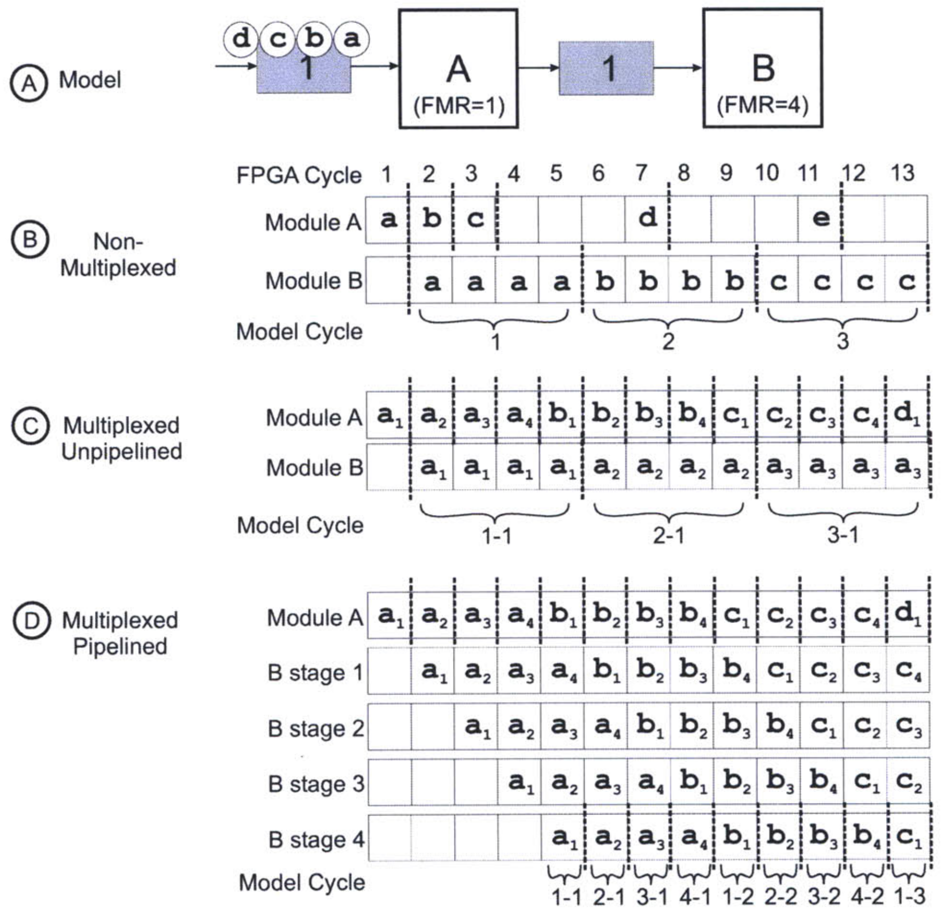


Figure 4-3: Pipelining modules can offset the performance penalty from time-multiplexing.

Note that the time-multiplexing scheme is possible only because the state of the different cores being simulated is independent. That is, the Register File of Core 0 cannot directly affect the Register File of Core 1. Only by going through the OCN can the various cores affect each other's simulation results. Because of this *cross-instance* communication, traditional time-multiplexing is insufficient for modeling the OCN—different techniques are needed that can take the interaction into account while still exploiting fine-grained parallelism.

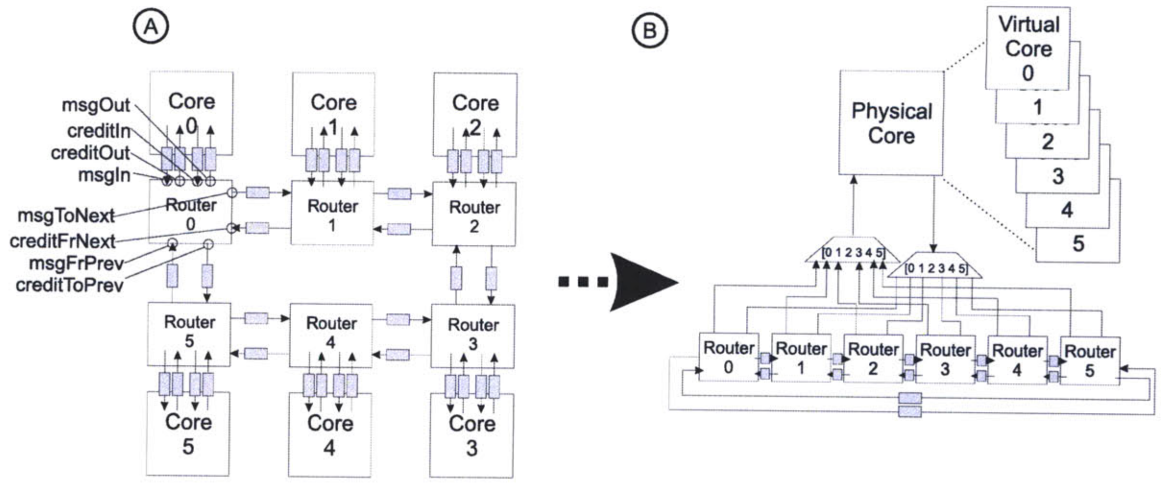


Figure 4-4: (A) Target multicore with uni-directional ring network. (B) Multiplexed core connected to ring-network routers via sequential de-multiplexing.

4.2 Time-Multiplexed Simulation of Networks via Permutations

4.2.1 First Approach: De-multiplexing

The previous section established that time-multiplexing the core works well because it improves both scaling and utilization. Now, the problem becomes attaching a single physical (time-multiplexed) core to an on-chip network. Consider the ring network shown in Figure 4-4A. Each router has 4 ports that communicate with the core: `msgIn`, `creditIn`, `msgOut`, and `creditOut`. Additionally each router has 4 more ports that communicate with adjacent routers: `msgToNext`, `creditFromNext`, `msgFromPrev`, `creditToPrev`.

A baseline approach to simulating this target is to duplicate the routers, and synthesize an on-chip network directly. The messages from the cores are then *sequentially de-multiplexed* and sent to the appropriate router. Each router can now simulate its next model cycle when data arrives. Responses are *re*-multiplexed and returned to the cores. This situation is shown in Figure 4-4B. In this figure and throughout the paper we represent sequential de-multiplexing by augmenting a de-multiplexor with a sequence denoting where each sequential arrival is to be sent. In this case the first arrival is sent to router 0, the second to router 1, and so on.

While this scheme is functionally correct, it presents many practical challenges. Most significantly, the physical core is now no longer adjacent to any particular router. Thus the FPGA synthesis tools are presented with the difficult problem of routing the de-multiplexed signals to and from the individual routers. Second, the routers themselves are under-utilized: at any given FPGA cycle only a small subset of routers are actively simulating the next model cycle—most are waiting for their corresponding virtual core to the produce data for a given model cycle. HAsim solves this problem by extending the time-multiplexing to the OCN routers themselves via a novel use of permutations.

4.2.2 Time-Multiplexed Ring Network via Permutation

If we wish to time-multiplex the ring, observe that the simulation of router n is complicated by the communication from routers $n - 1$ and $n + 1$. It is the ports that cross between routers that present a challenge to time-multiplexing, as shown in Figure 4-5, as they express the fact that the differing virtual instance's behaviors are not independent. How can we ensure that each *cross-virtual instance* port's messages are transmitted to the correct destination?

The key insight, as shown in Figure 4-6, is that we can connect these ports to themselves. That is, the output from `msgToNext` is fed into `msgFromPrev`, and `creditFromNext` produces `creditToPrev`. This makes sense intuitively: messages leaving one router are the input to the next router. However note that simply making the connection is not sufficient: router n produces the message for router $n + 1$, not for router n.

One way to solve this would be to store cross-router communication in a RAM, as shown in Figure 4-7. The index of the RAM to be read and written by each virtual index would be calculated by accessing an indirection table. This approach is similar to the way a single-threaded software simulator simulates an on-chip network. The disadvantage is that a random-access memory is overkill, as the accesses are actually following a static pattern determined by the topology.

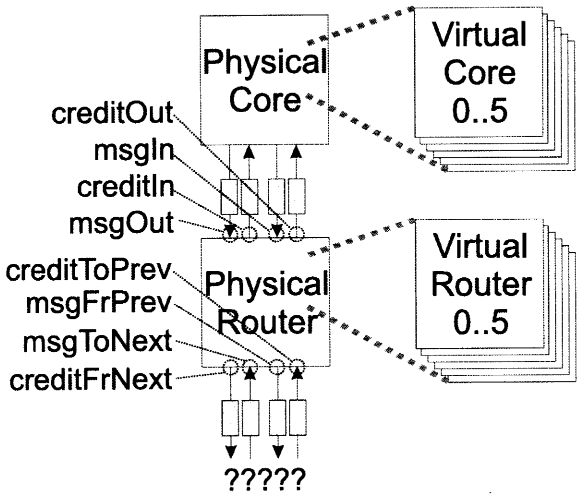


Figure 4-5: Time-Multiplexing the ring is complicated by the cross-router ports/dependencies.

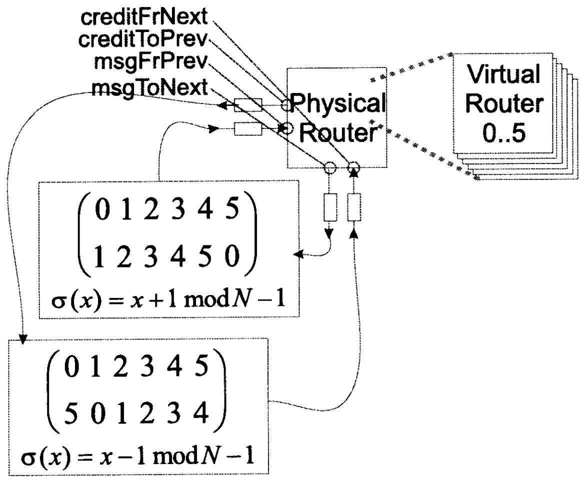

creditFrNext
creditToPrev
msgFrPrev
msgToNext
Physical
Router
Virtual
Router
0..5
$\begin{pmatrix} 0&1&2&3&4&5 \\ 1&2&3&4&5&0 \end{pmatrix}$
$\sigma(x) = x+1 \bmod N-1$
$\begin{pmatrix} 0&1&2&3&4&5 \\ 5&0&1&2&3&4 \end{pmatrix}$
$\sigma(x) = x-1 \bmod N-1$

Figure 4-6: Connecting the credit ports to each other, and the message ports to each other, and applying permutations to the messages that pass through them.

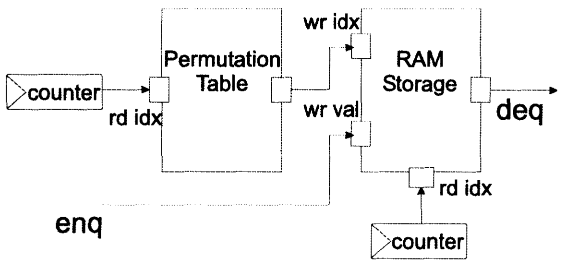


Figure 4-7: General permutations implemented via indirection tables.

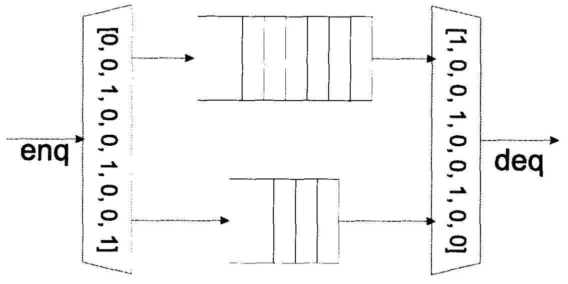


Figure 4-8: Permutations used in ring and torus topologies can be implemented via parallel queues.

HAsim's insight is that the communication pattern can be represented by a small permutation. For the `msg` port the output from router 0 is the input for router 1 (on the next model cycle), 1 is for 2, and so on to $N-1$, which is for 0. For the `credit` port 0 goes to $N-1$, 1 to 0, 2 to 1, and so on. The advantage of this approach is that these permutations can be represented using two queues: a main queue and a side buffer, as shown in Figure 4-8. A small FSM determines which queue will be enqueued to, and which queue will be dequeued from.

Formally, given N cores the permutation σ for the xth input of each port is as follows:

- $\sigma_{msg}(x) = x + 1 \bmod (N-1)$
- $\sigma_{credit}(x) = x - 1 \bmod (N-1)$

In this paper we will express the permutations as shown in Figure 4-6: a concrete table showing that the output for core 0 is sent to core 1, and so on, until core 5's output is sent to core 0. This table is then supplemented with a generalized formula that scales the permutation to any number of routers.

Given these permutations, Figure 4-9 shows a complete example of simulating a model cycle in the ring network. In 4-9A the messages are in their initial configuration. The router simulates the next model cycle, consuming N inputs and producing N new outputs, resulting in the state shown in 4-9B. After the permutation is applied we can confirm that the resulting configuration in 4-9C is correct: on the next model cycle router 0 will receive the message from router 3, and the credit from 1. Router 1

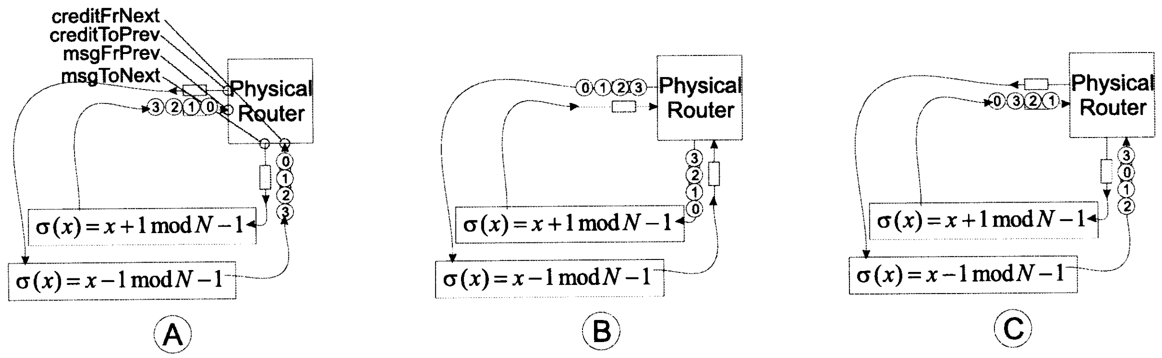


Figure 4-9: Simulating a model cycle for ring network via permutations.

will receive the message from router 0, and credit from router 2, and so on. Although we present this execution as happening in three separate phases, on the FPGA we can overlap the execution.

4.2.3 Time-Multiplexed Torus

Let us extend the permutation technique to another topology, the 2D torus shown in Figure 4-10. Here each router has ports going to/from 4 directions: `msgFromNorth, msgFromEast, msgFromSouth, msgFromWest` and so on, as well as ports/to from the local core. In the time-multiplexed implementation the `msgToEast` port is connected to the `msgFromWest` port and so on, as expected. However, compared to the ring network the permutation is different to reflect the width of the torus. In order to simulate the cores in numeric order, the permutation for the East/West ports for a network of width w is:

- $\sigma_{msgFromEast}(x) = x + 1 \bmod (w - 1)$
- $\sigma_{msgFromWest}(x) = x - 1 \bmod (w - 1)$

Similarly the permutation for the North/South port must take into account the *width* of the network (not the height):

- $\sigma_{msgFromNorth}(x) = x + w \bmod (N - 1)$
- $\sigma_{msgFromSouth}(x) = x - w \bmod (N - 1)$

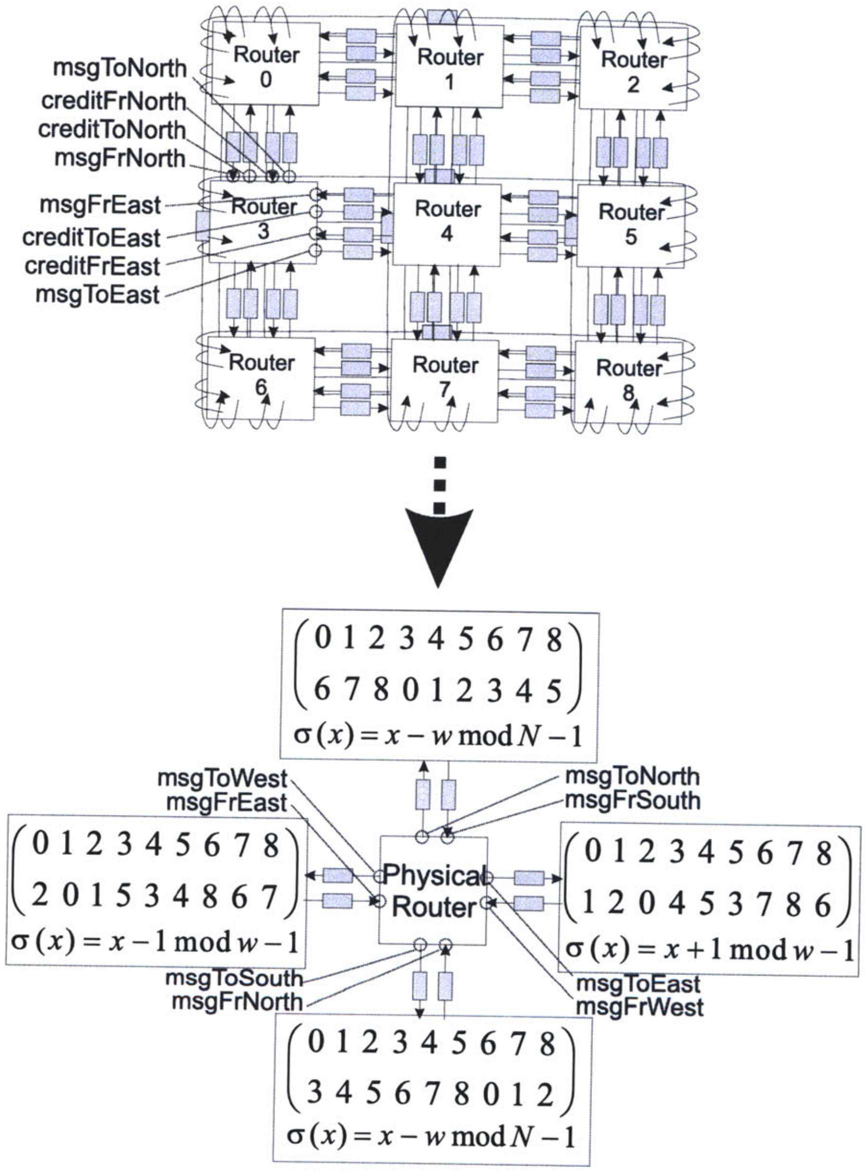

msgToNorth
creditFrNorth
creditToNorth
msgFrNorth
msgFrEast
creditToEast
creditFrEast
msgToEast
Router 0
Router 1
Router 2
Router 3
Router 4
Router 5
Router 6
Router 7
Router 8
Physical Router
msgToWest
msgFrEast
msgToNorth
msgFrSouth
msgToSouth
msgFrNorth
msgToEast
msgFrWest
$\begin{pmatrix}0&1&2&3&4&5&6&7&8\\6&7&8&0&1&2&3&4&5\end{pmatrix}$
$\sigma(x)=x-w \bmod N-1$
$\begin{pmatrix}0&1&2&3&4&5&6&7&8\\2&0&1&5&3&4&8&6&7\end{pmatrix}$
$\sigma(x)=x-1 \bmod w-1$
$\begin{pmatrix}0&1&2&3&4&5&6&7&8\\1&2&0&4&5&3&7&8&6\end{pmatrix}$
$\sigma(x)=x+1 \bmod w-1$
$\begin{pmatrix}0&1&2&3&4&5&6&7&8\\3&4&5&6&7&8&0&1&2\end{pmatrix}$
$\sigma(x)=x-w \bmod N-1$

Figure 4-10: Time-multiplexing an example torus network. Local cores/caches are not pictured. Credit ports are omitted as they use the same permutations.

Note that these permutations mean that the output from router 0 will be sent to routers $\sigma_{msgFromNorth}(0) = 3$, $\sigma_{msgFromEast}(0) = 1$, $\sigma_{msgFromSouth}(0) = 6$, and $\sigma_{msgFromWest}(0) = 2$. Similarly router 0 will receive messages from $\sigma_{msgFromNorth}(6) = 0$, $\sigma_{msgFromEast}(2) = 0$, $\sigma_{msgFromSouth}(3) = 0$, $\sigma_{msgFromWest}(1) = 0$, corresponding exactly to the original target.

4.2.4 Time-Multiplexed Grid

Once we have a torus model it is straightforward to alter this model to simulate a grid topology such as the one shown in Figure 4-11. We will not do this by altering the permutations or physical ports of our network, but rather by just altering the routing tables to send `NoMessage` along the links that do not exist in the grid network. For instance, router 0, in the Northwest corner, will only send `NoMessage` West or North. If other routers obey similar rules then it will only receive `NoMessage` from those directions as well.

The permutations given in this section assume that the first processor that should be simulated (core 0) is located in the upper left-hand corner of the topology. If the architect for some reason desired a different simulation ordering they could accomplish this by changing the permutation — analogous to a software simulator of a torus changing the order of indexing in a for-loop.

4.3 Generalizing the Permutation Technique

The permutations described earlier correspond to picking the simulation order of the routers in the network and properly routing the data between them, similar to how a sequential software simulator cycles through nodes in sequence. It is always possible to create a sequential simulator for any valid OCN topology. In this section we demonstrate that it is similarly always possible to construct a set of permutations to allow any valid topology to be time-multiplexed.

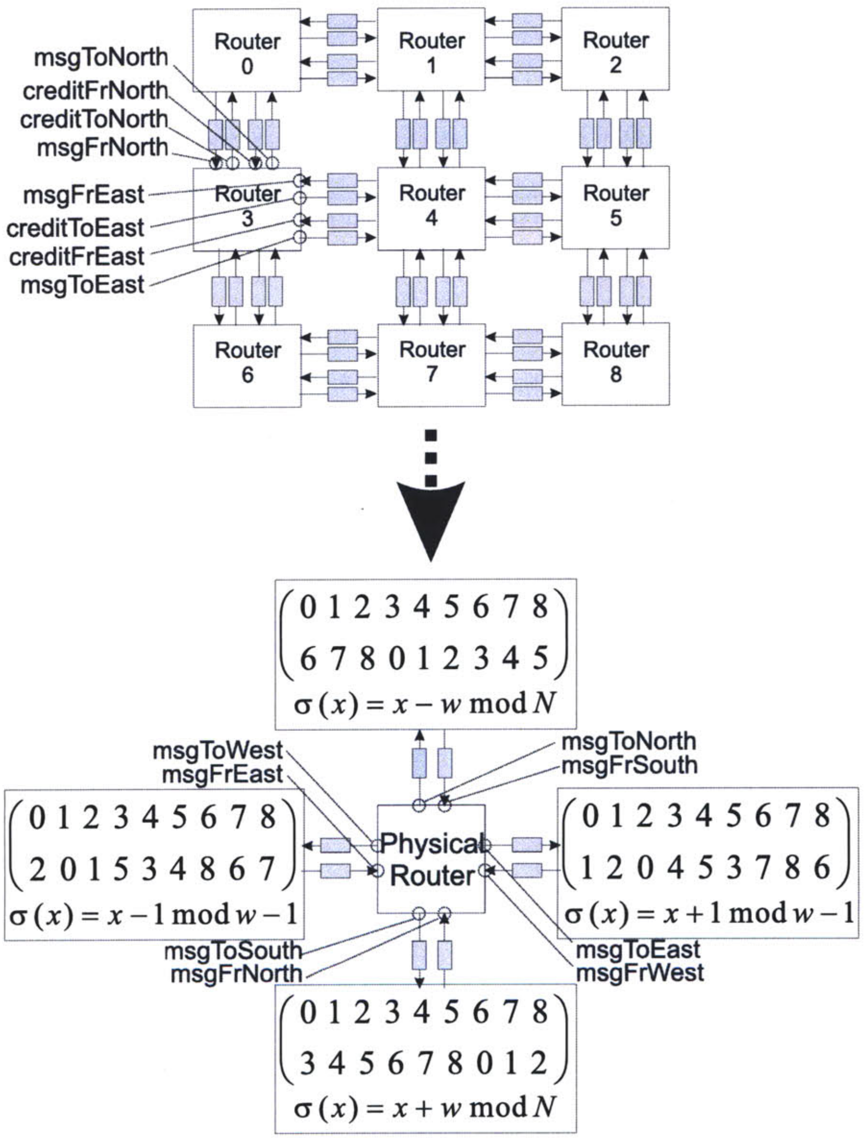


Figure 4-11: Target grid network. This can be simulated using the same permutations as the torus and sending NoMessage values on non-existent edges.

4.3.1 Permutations for Arbitrary Topologies

Assume that the target OCN has been expressed as a port-based model: a digraph $G = (M, P)$ where M is the modules in the system and P is the ports connecting them. Label the modules M with a valid simulation ordering $[0, 1, .., n]$ such that 0 is the first node simulated and n is the last. Note that if the graph contains zero-latency ports then not all simulation orderings will be valid. However if the graph represents valid hardware then there is guaranteed to exist at least one valid simulation ordering.

Once the simulation ordering is picked we must combine the ports into as few time-multiplexed ports as possible. To do this we divide the edges P into the minimum number of sets $P_0, P_1..P_m$ such that each set P_m obeys the following properties:

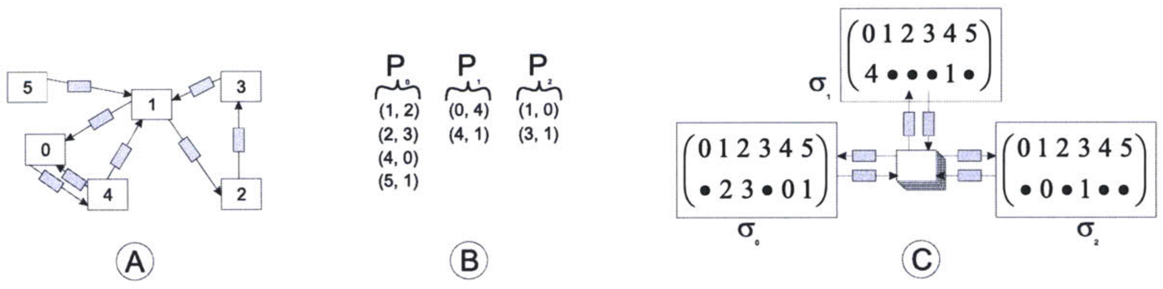
- $\forall\{s, d\} \in P_m, \neg\exists\{s', d'\} \in P_m.s = s'$
- $\forall\{s, d\} \in P_m, \neg\exists\{s', d'\} \in P_m.d = d'$

In other words, no two ports in any given set can share the same source, or share the same destination. Each set P_m corresponds to a permutation that we must construct in our time-multiplexed model. Ensuring that no source or destination appears twice ensures that we will construct a valid permutation. We construct permutations $\sigma_{0..n} : M \rightarrow M$ using the following rule:

- $\forall\{s, d\} \in P_m, \sigma_m(s) = d$

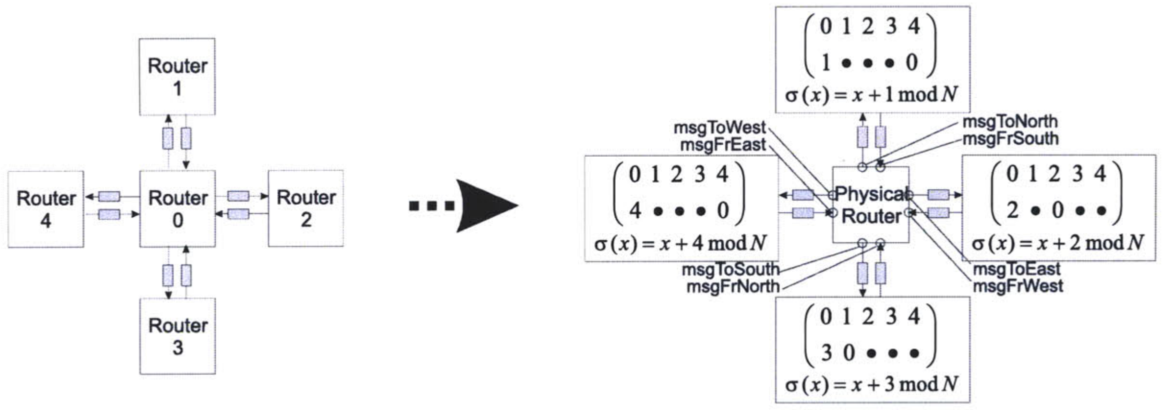
The remaining range of σ_m represent "don't-care" values and so may be chosen in any way that creates a valid permutation. (It is possible that certain permutations will be cheaper to implement on an FPGA than others.)

Finally, each permutation should be associated with a port with a source and destination of the remaining module. This module can be time-multiplexed using existing techniques referenced in Section 4.1, with one additional restriction: the time-multiplexed module should ensure that `NoMessages` are sent on port m for undefined values in the range of σ_m. This represents the fact that these output ports do not exist for a particular virtual instance. The grid versus torus discussion in Section 4.2.3 is an example of this phenomenon.



5
1
3
0
4
2
A
$\underbrace{P_0}$ (1, 2) (2, 3) (4, 0) (5, 1)
$\underbrace{P_1}$ (0, 4) (4, 1)
$\underbrace{P_2}$ (1, 0) (3, 1)
B
σ_1 $\begin{pmatrix}0&1&2&3&4&5\\4&\bullet&\bullet&\bullet&1&\bullet\end{pmatrix}$
σ_0 $\begin{pmatrix}0&1&2&3&4&5\\\bullet&2&3&\bullet&0&1\end{pmatrix}$
σ_2 $\begin{pmatrix}0&1&2&3&4&5\\\bullet&0&\bullet&1&\bullet&\bullet\end{pmatrix}$
C

Figure 4-12: Building permutations for an arbitrary network.



Router 1
Router 4
Router 0
Router 2
Router 3
$\begin{pmatrix}0&1&2&3&4\\1&\bullet&\bullet&\bullet&0\end{pmatrix}$ $\sigma(x)=x+1 \bmod N$
$\begin{pmatrix}0&1&2&3&4\\4&\bullet&\bullet&\bullet&0\end{pmatrix}$ $\sigma(x)=x+4 \bmod N$
$\begin{pmatrix}0&1&2&3&4\\2&\bullet&0&\bullet&\bullet\end{pmatrix}$ $\sigma(x)=x+2 \bmod N$
$\begin{pmatrix}0&1&2&3&4\\3&0&\bullet&\bullet&\bullet\end{pmatrix}$ $\sigma(x)=x+3 \bmod N$
msgToWest
msgFrEast
msgToNorth
msgFrSouth
Physical Router
msgToSouth
msgFrNorth
msgToEast
msgFrWest

Figure 4-13: Multiplexing a star topology results in many undefined values representing non-existent ports.

Figure 4-12 shows an example applying this process to an arbitrary, irregular topology. First a desired simulation order is selected (4-12A). The ports are arranged into three sets (4-12B), the fewest possible for this example. These sets then form the basis of permutations (4-12C). The don't-care values of the permutations can be can be resolved in any way that creates a legal permutation. The router is time-multiplexed across 6 virtual instances, and the virtual instances are arranged to send `NoMessage` values on non-existent ports. For example, instance 0 will send NULLs on two of the output ports, as the original router 0 only had one output port.

The meaning of undefined values in the permutations can clearly be seen when we apply the technique to a star network topology (Figure 4-13). The resulting time-multiplexed network has the same number of physical ports as the grid network, but the permutations themselves are different. Each leaf node only contains a subset of nodes of the hub, and thus will send `NoMessage` on ports that do not exist for them. Given this, the undefined values in the permutations can be filled in using straightforward modular arithmetic.

4.3.2 Heterogeneous Network Topologies

Thus far we have presented OCNs where all of the routers are connected to homogeneous cores. This has kept the examples pedagogically clear, but is unrealistic. Architects often wish to study multicores such as those shown in Figure 4-14, a 3x3 grid that contains a memory controller, 2 last-level caches, and 6 cores. The cores and caches will be simulated using time-multiplexing. How then can we connect them to our permutation-based grid? The answer is to sequentially multiplex the streams together, pass them to the time-multiplexed OCN, and de-multiplex the responses. This is shown in Figure 4-15. Unlike the original de-multiplexing approach presented in Section 4.2.1 this imposes no difficult routing problem on the synthesis tools, as the modules being connected are time-multiplexed physical cores. A key advantage of this technique is that it requires no changes to the individual modules—they can be time-multiplexed independently using established techniques.

This same technique allows for efficient time-multiplexing of *indirect* network topologies such as butterflies, shown in Figure 4-16. The technique can also be extended to topologies where nodes have differing input and output degree, such as the tree network shown if Figure 4-17.

This same technique allows us to support *indirect* network topologies. These are networks that possess intermediate routers that are not connected to any external modules, but only route messages to other routers. We implement this by interleaving messages from the cores with messages from other routers. An example of this for a butterfly network topology is given in Figure 4-16. Interestingly, the butterfly topology can be implemented using only interleavers, without the need for permutations. Each node has four ports: `from0`, `from1`, `to0`, `to1`. The output from the cores and the routers are interleaved with the first 8 outputs from the routers to create the input ports. The remaining 4 outputs are interleaved to form the inputs to the cores.

The interleaving technique can also be used to implement a tree topology (Figure 4-17). In this network each node has ports to/from `North`, `SouthEast`, and

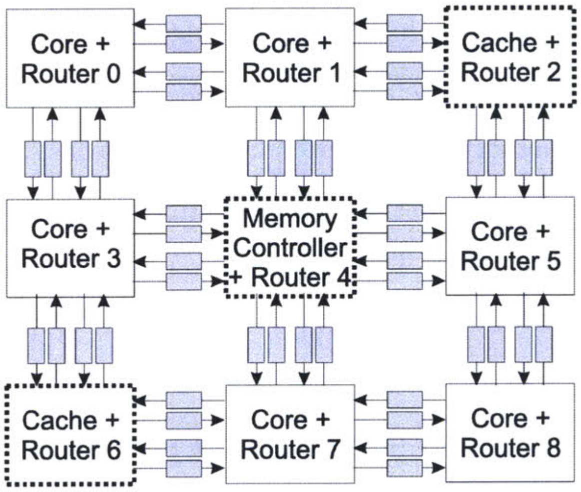


Figure 4-14: A heterogeneous grid, where routers connect to different types of nodes.

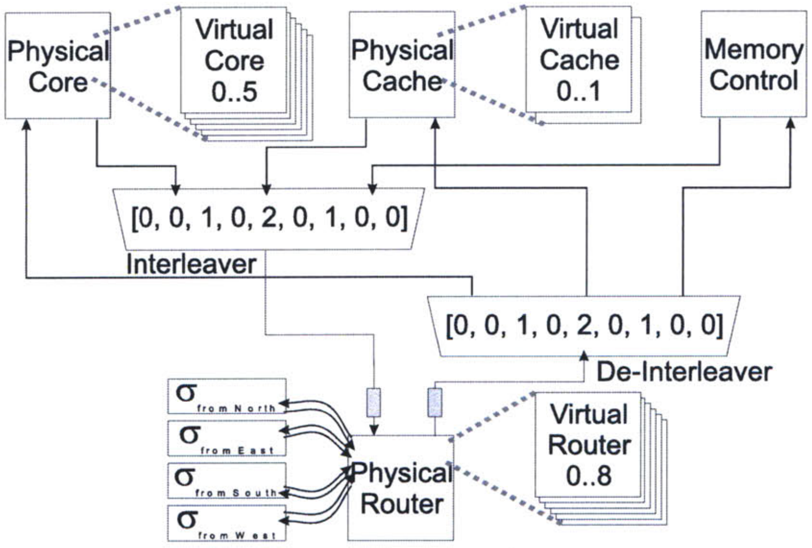


Figure 4-15: Time-multiplexing the heterogeneous network via interleaving.

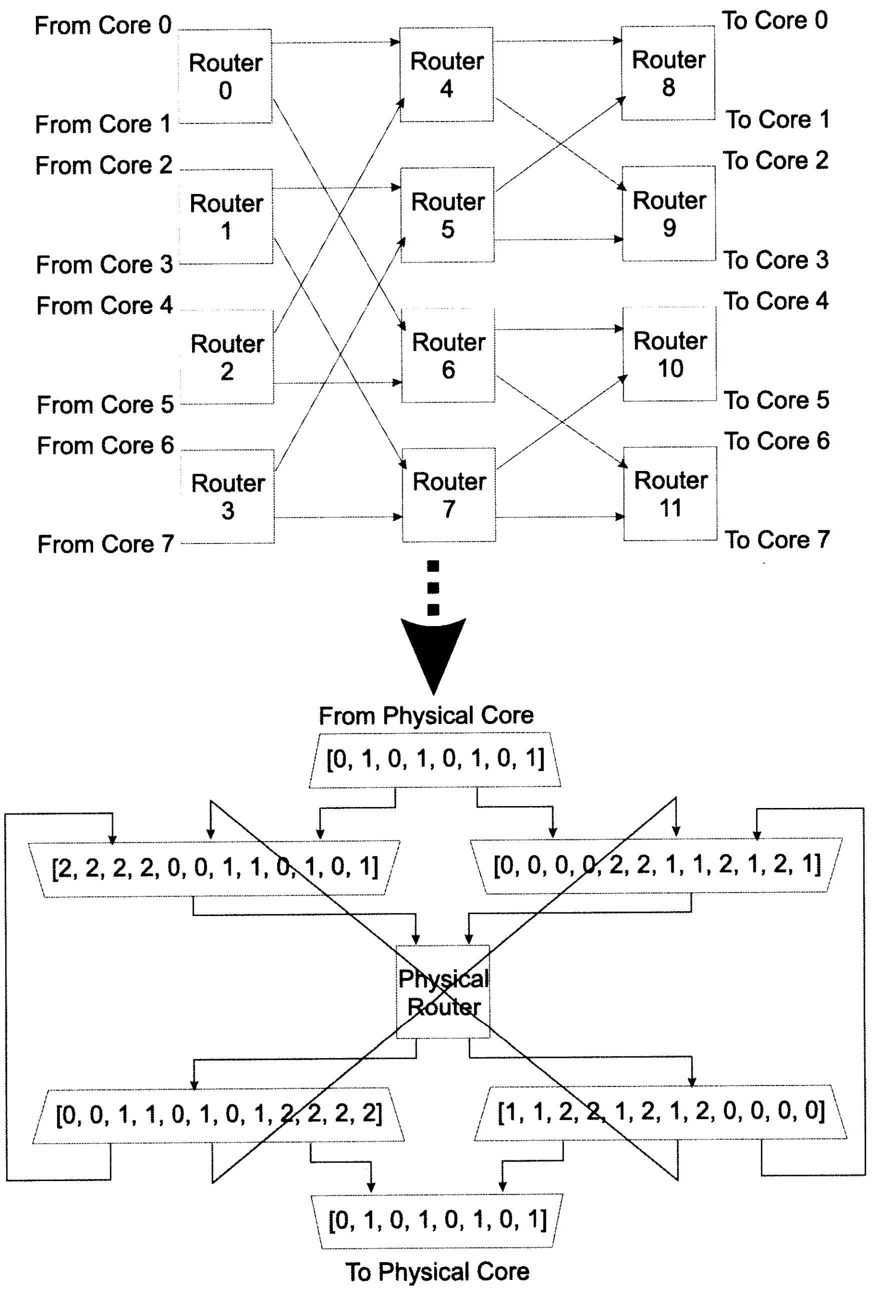


Figure 4-16: Multiplexing an indirect butterfly network via interleaving.

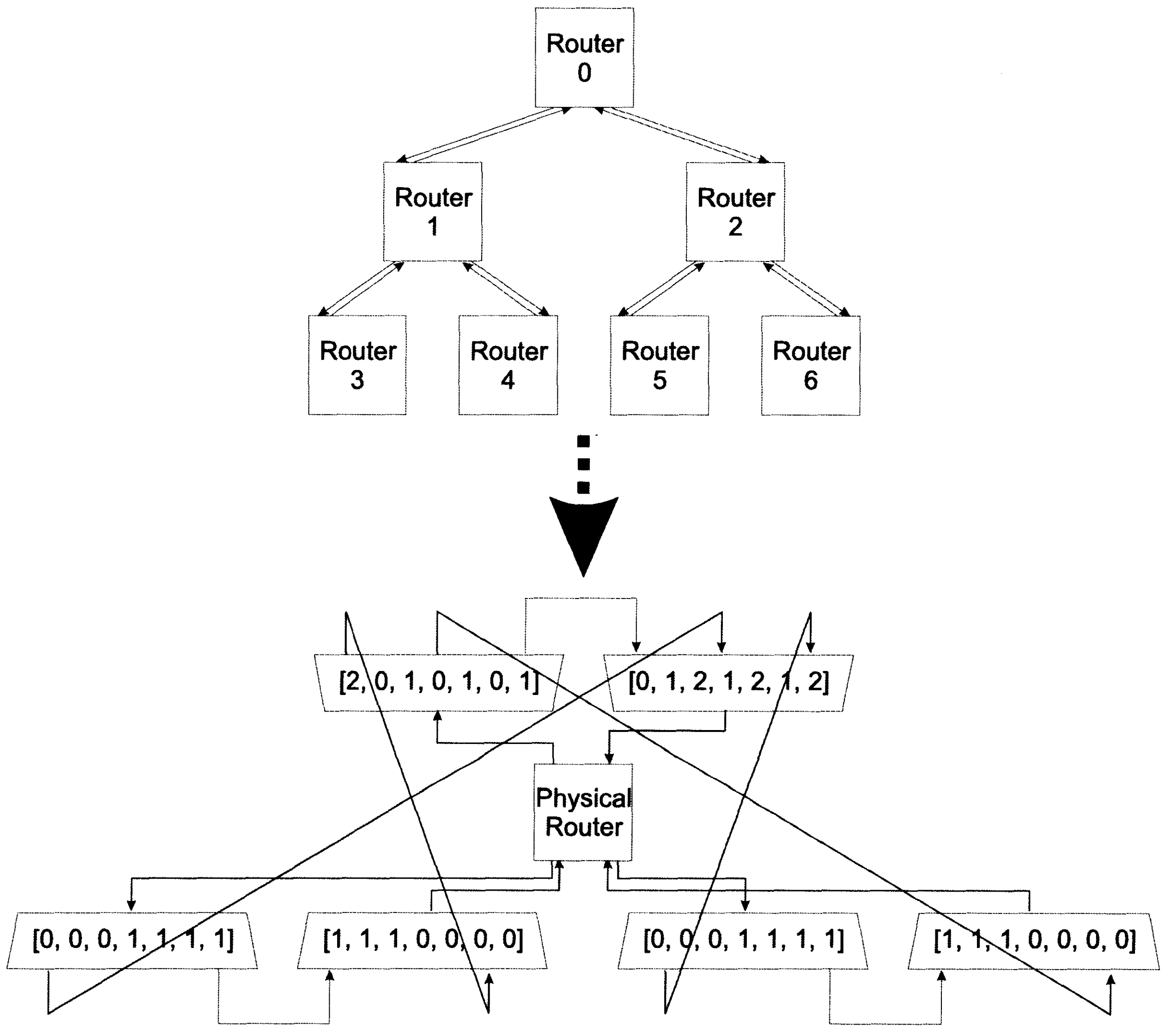


Figure 4-17: Multiplexing an tree network topology.

`SouthWest`. In the target the root node has no to/from North port and leaf nodes have no to/from SouthEast and SouthWest ports. The time-multiplexed version simulates this by using the NULLs leaf nodes send to the South as the NULL inputs from that direction. Non-leaf messages that are sent SouthEast and SouthWest are interleaved to create the `fromNorth` port, with an additional NULL for the root node. Messages sent `toNorth` are de-interleaved and then re-interleaved to form the `fromSouthEast` and `fromSouthWest` ports.

These realistic topology examples demonstrate the generality of the time-multiplexing technique.

4.4 Discussion

Time-multiplexed simulation of detailed multicores using FPGAs represents a new tool in the architect's toolchest of simulation techniques. By trading space-savings for sequentialized simulation, it allows the possibility to free up substantial FPGA area. This critically limited resource can then be utilized to increase fidelity without negatively impacting simulation rate.

Alternatively, a natural extension of the techniques presented in this paper is to store the state of the virtual instances off-chip. Careful orchestration of memory accesses should be able to bury much of this latency and keep the physical pipeline busy. Currently we are aiming to use the techniques discussed here to model a thousand-node on-chip network using only a single time-multiplexed FPGA.

Notes

[5]This kind of multiplexing bears a resemblance to multi-threading in real microprocessors, but it is important to distinguish that this is a simulator technique, not a technique in the target architecture. The cores being multiplexed may or may not support multi-threading.

[6]Again, note that this refers to altering the implementation of the modules on the FPGA, not altering the timing characteristics of the target circuit, which are preserved by the ports.

Chapter 5

Implementing Timing-Directed Simulation on FPGAs

5.1 Introduction

Previous chapters demonstrated that separating the model cycle from the FPGA cycle can lead to benefits for efficiency on the FPGA, increasing the amount of detail that can be fit within an FPGA's capacity. In this chapter, we leverage the latency-insensitive nature of this separation to aid with the development effort problem discussed in detail in Chapter 2.

We begin by demonstrating how this can be used to integrate existing FPGA modules such as floating-point units into our simulator. We then expand this technique to include emulating rare-but-expensive-to-implement instructions. Finally, we extend this technique to *timing-directed simulation*, an existing simulation technique for reducing development effort whereby the simulator is divided into *functional* and *timing* partitions. We present HAsim's specific partitioning scheme, and argue that it is general enough to represent a wide range of cores. We present details of the functional partition's architecture, including efficient implementation on FPGAs. We then cover the implementation of a variety of timing models for cores, caches, and on-chip networks. HAsim is the first project to implement a generalized functional partition on an FPGA.

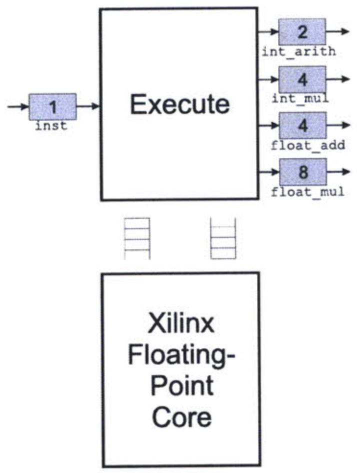


Figure 5-1: Integrating an existing floating point core.

5.2 Leveraging Latency-Insensitivity of A-Ports

5.2.1 Integrating Existing IP Cores

Chapter 3 established that one benefit of the A-Port network is that each module can take a different number of FPGA cycles per model cycle, and that this number can vary dynamically. We can leverage this to aid with development effort by integrating existing modules into our simulator, thus increasing code reuse.

Specifically, there are many existing IP blocks designed for FPGAs. These blocks have particular timings—taking a certain number of FPGA cycles to perform operations—determined by the particulars of the FPGA platform and the goals of the implementation (low area, low critical path, etc.). The A-port scheme allows us to use their functionality while still simulating timings of the ASIC processor we ultimately wish to construct.

For example, consider adding floating point support to our processor. Xilinx provides a configurable IEEE-complaint floating point core as part of their Core Generator utility. These cores use pipelines optimized for FPGAs, and therefore have FPGA-specific timings. Figure 5-1 shows an implementation of a model of a processor's Execute stage which sends floating points to an existing FPU. Because the Execute stage's timings are determined by the A-Ports, the timings of the floating-point pipeline will not affect the results of the simulation, but only its rate.

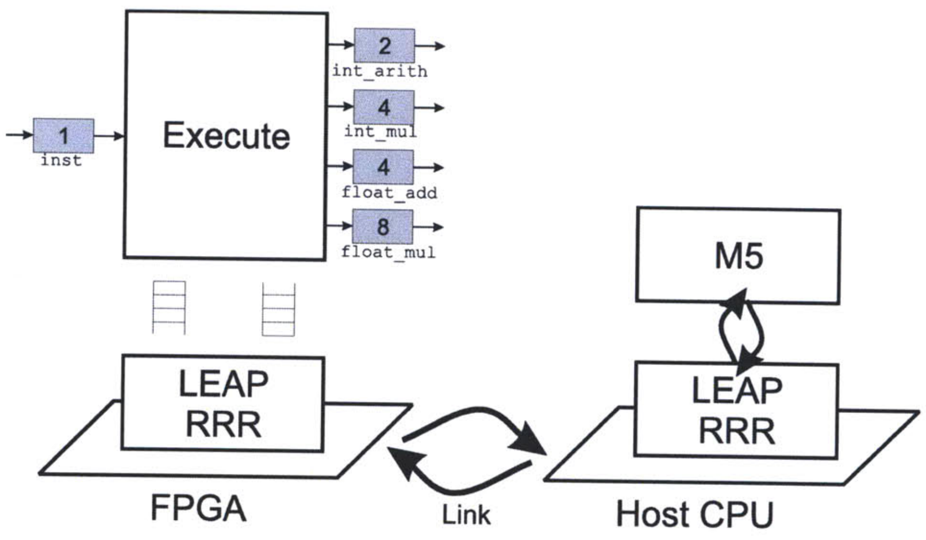


Figure 5-2: Integrating the existing M5 [5] simulator using LEAP RRR (Section 2.5.2).

5.2.2 Interacting with a Software Simulator

Additionally, there are classes of instructions for which no existing IP for FPGAs exists—perhaps because they are too expensive to implement on the FPGA, or because the amount of development effort to implement them would be too high. Examples of this include system calls, CISC-style instructions with multiple side effects, system calls, and non-standard rounding modes for floating point instructions—which are not supported by Xilinx IP blocks.

To handle these classes of instructions we can use LEAP RRR (Section 2.5.2) to communicate with an existing software simulator. As shown in Figure 5-2 HAsim uses the M5 full-system simulator [5] as a software backer to handle system calls and page faults. The modular nature of M5 allows HAsim to tie directly into M5's memory subsystem while ignoring its CPU timing models. When the FPGA detects a system call it transfers the architectural state of the simulated processor to HAsim's software, which invokes M5. After the state update is calculated, it is transmitted back to the FPGA, at which point simulation on the FPGA can resume.

The ProtoFlex project applied these principles to emulation [15]. They demonstrated that if these events are rare enough the impact on performance can be minimized, while still resulting in significant gains in development effort.

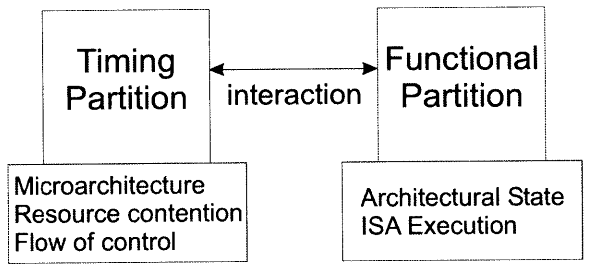


Figure 5-3: Overview of simulator partitioning.

5.3 Timing-Directed Simulation

We can extend this notion of integrating existing IP into our latency-insensitive A-Port network to use *partitioned simulation*, an existing technique used to reduce development effort. In microprocessors, exploring a future generation is often mostly about exploring when things happen (branch predictors, cache strategies, pipeline depths), and only a limited amount of genuinely new functionality (what operations do). A partitioned simulator is divided into *functional* and *timing* partitions, as shown in Figure 5-3. As the functional partition may take an arbitrary number of FPGA cycles to carry out its execution, the A-Port Network scheme's latency insensitivity is necessary to coordinate the timing model while the functional partition operates.

In a partitioned simulator, the functional partition is responsible for correct ISA-level execution[7]. The timing partition (or *timing model*) is responsible for driving the functional partition in such a way as to simulate a particular microarchitecture. Example responsibilities of the functional partition include decoding instructions, updating simulator memory, or guaranteeing that floating point operations conform to standards. Example responsibilities of the timing partition include deciding what instruction to issue next, tracking branch mispredictions, and recording that floating-point multiply instructions take 5 clock cycles to execute.

The goal of this partitioning is to speed development time. The functional partition might be complex to implement, optimize, and verify, but once it is complete it can be reused across many different timing models. The timing models themselves are significantly simpler to implement than simulators written from scratch: they do not need to worry about ISA functional correctness, but only track microarchitecture-specific timing details. Often structures can be excluded from the timing model completely, or modeled only partially. A common example of this is a timing model of a cache that needs to track tags and status bits but does not need to store the instructions or data — the goal being to decide whether a particular load hits or misses, but not actually track the data associated with it.

Most importantly, a large amount of code reuse is available between timing model generations, as only those portions of the microarchitecture that change from one generation to the next need to be re-implemented. Practice with the Asim simulator environment has shown models can be decomposed in such a way as to reuse branch predictors, cache hierarchies, or communication networks with no changes whatsoever.

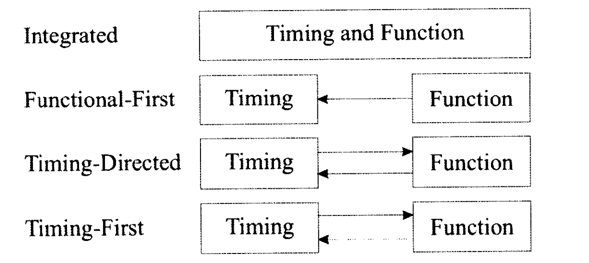
In this section we describe the architecture of HAsim's functional partition, the first functional partition implemented entirely on an FPGA, and a key contribution of the HAsim project.

5.3.1 Partitioning Schemes

Contemporary partitioned software simulators include Asim and MASE [33]. Within partitioned simulation, there are many potential ways for these functional/timing partitions to interact. Mauer, Hill and Wood [38] categorized such simulators as *functional-first* (traditionally called *trace-driven*), *timing-directed*, and *timing-first*, as shown in Figure 5-4. The schemes are differentiated as follows:

- In the functional-first scheme a functional model is used to generate an execution trace that is fed into a timing model, which adds microprocessor-specific timing information to the trace. The advantage is that an existing instruction-set simulator such as QEMU [4] can often be used to generate the trace with little or no modification. The downside of this scheme is that the functional trace contains only correct-path instructions, making it difficult to conduct evaluations on speculative architectures.

- In the timing-first style timing is first calculated, and then a functional model invoked to verify the results. Again, the benefit is that an existing emulator



Arrows indicate inter-simulator interactions per simulated instruction.

Figure 5-4: Mauer, Hill, and Wood's categorization of partitioned simulators. Source: [38]

can be used as the functional model with little modification. This style is rarely used because in practice it results in too many rollbacks from the functional model, which reduces simulation rate.

- In the timing-directed scheme, the simulator is execution-driven, meaning that the timing model invokes operations on the functional model at the right time. The benefit of this scheme is that it allows a full level of detail including speculation, and can easily be adapted to handle superscalar timing models. The disadvantage of this scheme is that it requires a custom functional partition that allows for phased execution, committing, and aborting of instructions.

HAsim is directed at high-detail modeling including speculative architectures. Additionally, in the context of FPGA-accelerated simulation, there are no existing functional partitions for FPGA. Therefore HAsim uses the tightly-coupled timing-directed scheme. The reasoning behind this decision is as follows:

- The fine-grained parallelism of the FPGA can benefit both the timing and functional partitions, which both have high degrees of parallelism.
- The FPGA is able to handle the frequent communication between the partitions.
- Since the simulator can use multiple FPGA cycles per model cycle (Section 1.2), the functional partition can be implemented as multiple-FPGA-cycle pipelines that are optimized by experts for FPGA implementation. The pipelining of the timing model (Section 4.1.2) can help offset the latencies of these pipelines.

- Rare events which are difficult to place on the FPGA, such as system call instructions, can be farmed out to software regardless of whether they occur in the functional or timing partition, similar to Protoflex's migration scheme [15].

In the following sections we give the semantics of our particular timing-directed scheme, give an architecture to implement it on an FPGA, and evaluate the efficiency of our scheme.

5.4 Semantics of the HAsim Functional Partition

At a high level, the job of the functional partition is straightforward: given a machine in a certain state and an instruction, calculate the new state of the machine after executing the instruction. This coarse granularity is sufficient for modeling simple in-order pipelines, but is not a high-enough level of detail to capture the behavior of modern microprocessors which include features like out-of-order execution and speculative execution.

In order to be able to precisely capture control speculation, data speculation, and the timings of interactions between threads, we identified seven operations for our functional partition, as shown in Figure 5-5. These operations roughly correspond to stages in a traditional microprocessor pipeline, including support for controlling the precise timing of store operations in order to simulate thread communication. The effect of these operations is given in Figure 5-6.

The order in which the timing partition invokes these operations determines the state of the machine at any given moment. Figure 5-7 demonstrates how the same functional partition can be reused across three different timing models to simulate different microarchitectures. Each simulator must do the same fundamental amount of work — the only change is how this work relates to model time. Figure 5-7A shows a timing model of a simple in-order pipeline. This machine stalls on a read-after-write hazard, as between instructions 1 and 2. Thus this small program takes 8 model cycles to execute, assuming a perfect memory hierarchy and a one-cycle ALU. Figure 5-7B shows the same basic machine, but now modeling a bypass path which

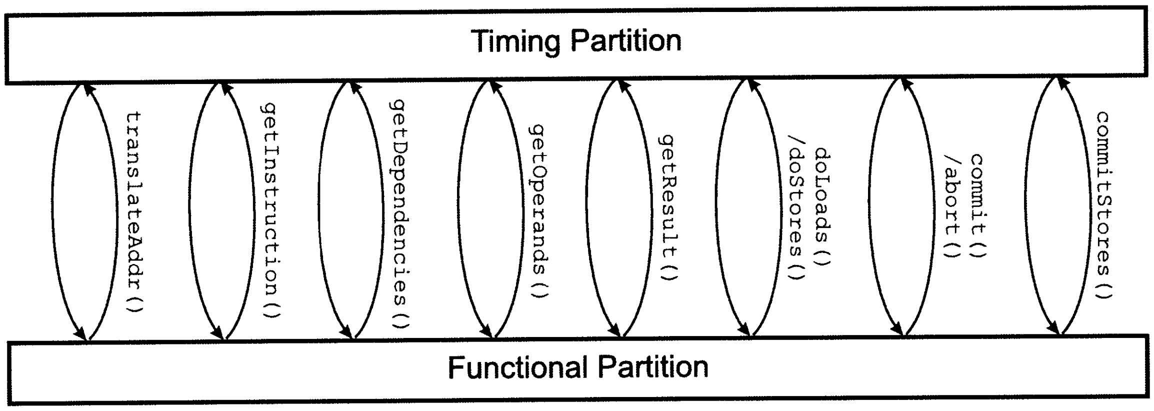


Figure 5-5: HAsim's timing-directed simulation scheme.

Operation	Params	Returns	Effect
`translateAddr`	VAddr	PAddr	Translate a virtual address into a physical address.
`getInstruction`	PAddr	Inst	Fetch the instruction at this address and place it in flight.
`getDependencies`	Inst	Deps	Get the dependencies of this instruction relative to other in-flight instructions.
`getOperands`	Inst	Srcs	Read the register file and prepare the instruction for execution.
`getResults`	Inst	Result	Execute the instruction and return the result, including branch information. For loads and stores, do effective address calculation.
`doLoads`	Inst	Value	Perform all memory reads associated with the instruction.
`doSpeculativeStores`	Inst	-	Make any memory writes visible to local loads.
`commit`	Inst	-	Commit the instruction's local changes and remove it from being in-flight.
`abort`	Inst	-	Abort the instruction's local changes and remove it from being in-flight.
`commitStores`	Inst	-	Make any memory writes globally visible.

Figure 5-6: Overview of functional partition operations.

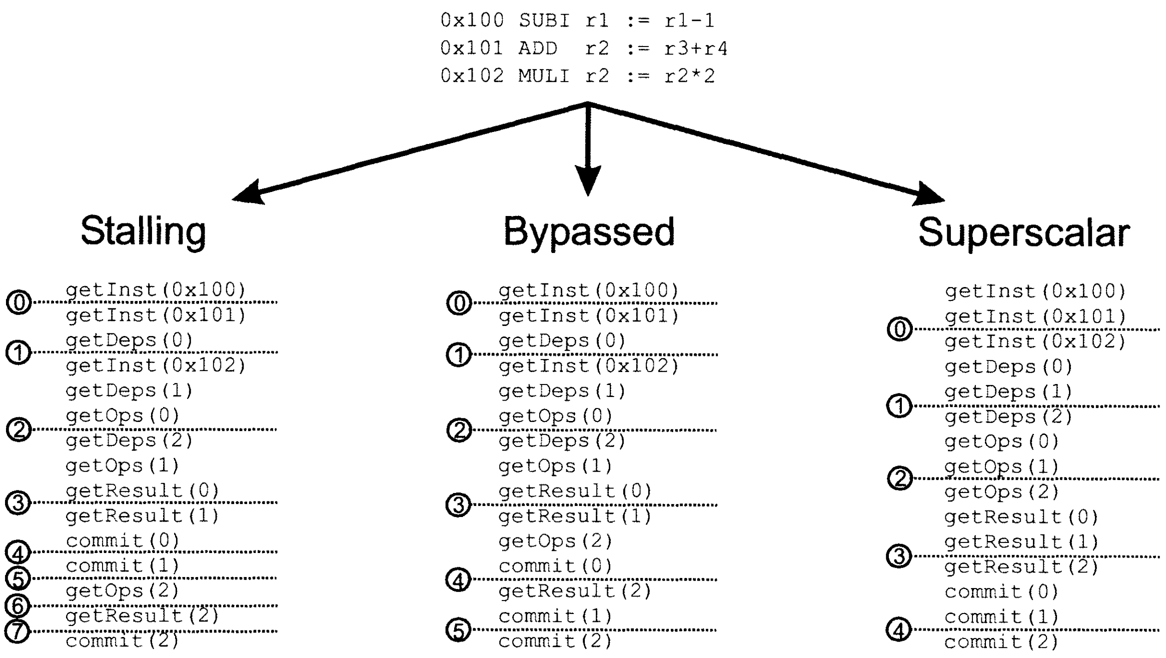


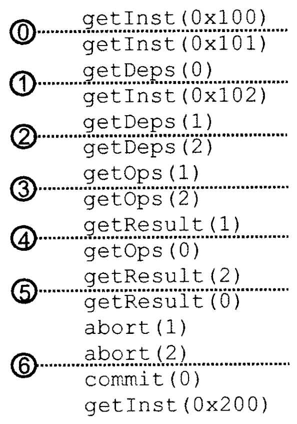
Figure 5-7: Three different timing models operating on the same instruction stream.

removes read-after-write hazards. Figure 5-7C shows a 2-way superscalar model that performs multiple operations on the functional partition before advancing the model clock cycle. Note that the functional partition is oblivious to the fact that it is now being used to model a superscalar processor.

For any given in-flight instruction, the operations are typically invoked in the order they are given in Figure 5-5 (operations which do not apply may by skipped). This corresponds to instructions flowing through pipeline stages in a real processor: the instruction is fetched (`getInstruction`) before it is decoded (`getDependencies`), which takes place before register read (`getOperands`), and so on. The order in which the timing model invokes these operations on separate in-flight instructions determines the state of the machine. We can conceive of a timing model which fetches ten instructions before decoding one, for example.

As the functional partition executes each operation, it changes the architectural state of the simulator, and thus the result of subsequent operations. For example, executing `getResult()` on an instruction which writes register R5 will mean that a subsequent `getOperands()` call will see that value of R5. If an instruction is executed in some way which is not consistent with program order, the `abort()` operation

```
0x100 BEZ  r1 0x200
0x101 SUB  r2 := r2 - r3
0x102 ADDI r4 := r4 + 1
```



```
getInst(0x100)
getInst(0x101)
getDeps(0)
getInst(0x102)
getDeps(1)
getDeps(2)
getOps(1)
getOps(2)
getResult(1)
getOps(0)
getResult(2)
getResult(0)
abort(1)
abort(2)
commit(0)
getInst(0x200)
```

Figure 5-8: Timing model demonstrating out-of-order issue and speculative execution.

undoes its effects and allows it to be retried. All operations are speculative and may be aborted until the `commit()` operation is called, at which point they become permanent. The distinction between local writes and global ones allows for accurate tracking of thread communication.

Using these constructs a timing partition can accurately model advanced processor features such as out-of-order issue, or speculative execution, as shown in Figure 5-8. In this example the branch is stalled on a dependence, and the timing model predicts branch not-taken and issues past it. Upon branch resolution the abort mechanism is used to rollback speculative operations. Data speculation can be supported by having the timing model provide results of the operations themselves (telling the functional partition that the result of `getOperands` should be zero, for instance), though this is not yet implemented.

It should be noted that the lack of ordering restrictions is loose compared to the requirements of most modern processors. Using these operations one could construct timing models which commit instructions out of program order, or fetch instructions non-sequentially. We specifically chose this level of granularity because it allows timing partitions the flexibility to model all types of speculation. In most cases it makes sense for the functional partition to check that committed instructions follow program order, raising a simulator exception if dependencies are violated.

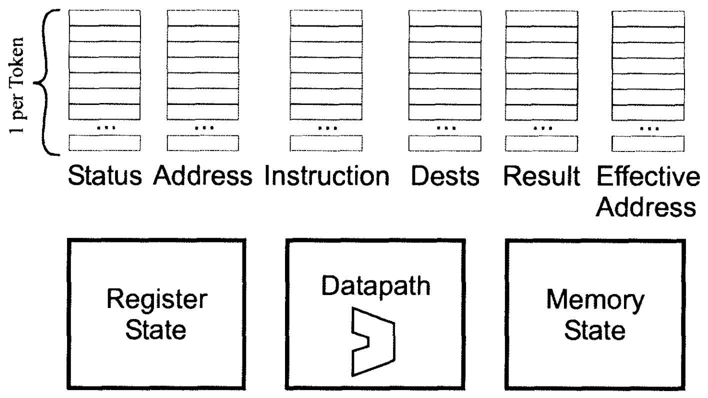


Figure 5-9: HAsim functional partition FPGA architecture.

5.5 Functional Partition Implementation

The FPGA implementation of HAsim's functional partition concentrates on making good use of port-limited BlockRAMs while maintaining a high-degree of parallelism. As shown in Figure 5-9, we use BlockRAMs to track the register state and memory state of the machine, as well as information about each in-flight instruction. The register state includes a physical register file, maptable, freelist, and snapshot/rollback mechanism to handle aborts. The memory state includes an on-FPGA cache and a store buffer which determines the youngest store to a particular address.

In-flight instructions are tracked using *tokens* -- pointers which allow the timing partition to refer to specific instructions without passing large amounts of data back and forth. The number of bits used to represent the token determines the number of instructions which may be in-flight simultaneously. This size is set by a static compile-time parameter, allowing it to be increased if a particular value proves insufficient (though doing so will increase the size of all the tables). It is often necessary to compare two in-flight instructions to see which is older (for example, see the store buffer below). For efficiency we wish to do this comparison using the tokens of the instructions. In order for this comparison to function properly we must add the restriction that in-flight instructions are retired (or aborted) in token order. This represents a restriction over the general semantics of our functional partition, but is consistent with the semantics of the architectures we are modeling.

Time-multiplexing of virtual instances is supported by supplementing every token with a virtual instance ID. Functional partition register state is duplicated on a per-instance basis. Age comparisons between tokens from different virtual instances are meaningless, but this is unnecessary since this is supported through the `doSpeculativeStores` and `commitStores` operation, which makes a memory update visible to other virtual instances.

5.5.1 Functional Partition Operations

The functional partition operations described previously in Section 5.4 are implemented as pipelines which read and write the token tables. For example, the getInstruction operation writes the address and instruction tables, which are later read by the `getResult` operation. An extra operation, `getToken`, was added to allocate a new in-flight instruction. Furthermore, the `getOperands` and `getResult` operations were merged for efficiency — none of the models we explored here utilized these separately, and by merging them we were able to eliminate intermediate state. The details of the operations implementations is as follows:

- Figure 5-10: The `getToken` operation creates a new in-flight instruction and returns the associated token. The `getInstruction` operation fetches the given address from memory, records it, and returns it.
- Figure 5-11: The `getDependencies` operation allocates a physical register file for the destination, and looks up which physical registers will contain the operands. The writers of those registers are returned.
- Figure 5-12: The `getOperands` and `getResult` operations are merged into one pipeline for efficiency. This pipeline reads the physical register file, as well as looking up the instruction itself to retrieve opcodes and immediate operands. The instruction address is retrieved for relative branches. Memory operations pre-calculate their effective addresses at this step.
- Figure 5-13: Loads read the value from memory, write it to the register file, and return it. Stores read the register file and make the value appear to local loads.

The commit operation frees the previous physical register which was mapped to a particular architectural register and deallocates the token. Not pictured: the `commitStores` operation commits the store in the memory state.

To implement rollbacks the register state was implemented as a physical register file with a maptable and a freelist, as would be found in many out-of-order processors. Rollbacks of stores are supported via a store buffer in the Memory State. Aborts have been grouped into a larger `rewindToToken` operation. Rewinds are implemented by sequentially walking back the maptable and undoing the execution of individual tokens. Therefore rewinds can take many FPGA cycles, but we expect them to occur comparatively rarely — at least, if the architect is modeling a target architecture that attempts to minimize speculative mispredictions.

5.5.2 ISA-Independent Datapath

When the timing model calls `getResults` the instruction sources are read from the register file and passed to the datapath. The datapath itself is defined in such a way as to be ISA-independent. In HAsim, support for a new ISA can be added by defining the following:

- Information on decoding the instruction: number of sources and destinations, and barrier information.
- A hardware datapath for executing common instructions.
- Optional software for emulating rare-but-expensive instructions such as system calls.

When an emulated instruction is encountered, the functional partition uses a migration system to transmit the updated register values of the current context to software. The software emulates the instruction, transmits register and memory updates (possibly invalidating the Memory State's caches), and returns control to the FPGA.

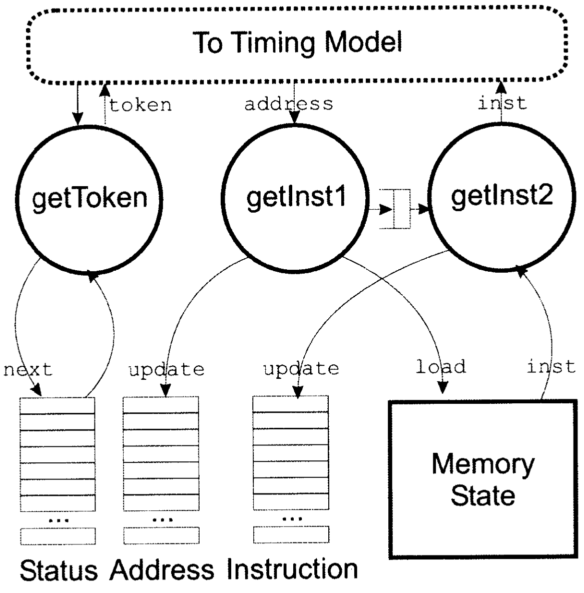


Figure 5-10: Implementation of `getToken` and `getInstruction` operations.

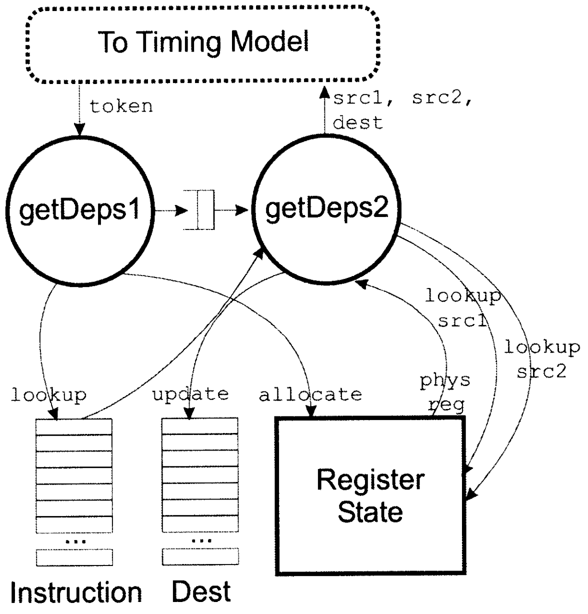


Figure 5-11: Implementation of `getDependencies` operation.

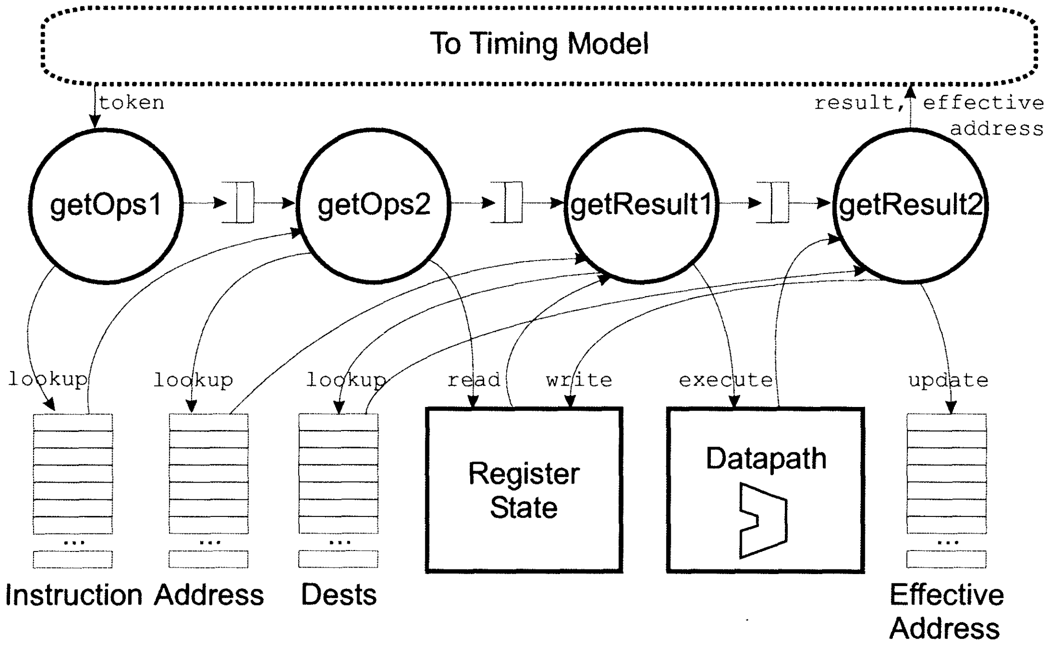


Figure 5-12: Implementation of `getOperands` and `getResult` operations.

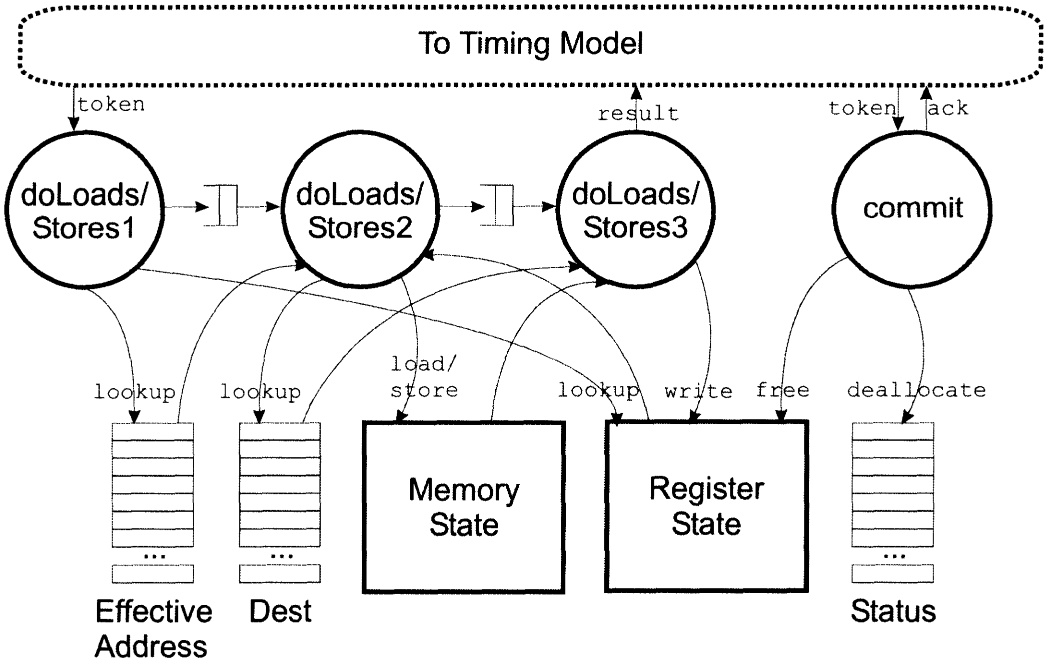


Figure 5-13: Implementation of `doLoads/SpeculativeStores` and `commit` operation.

Currently HAsim supports two ISAs: Alpha ISA and a subset of MIPS. Alpha system call emulation is supported by invoking the M5 simulator [5]. Alpha floating-point instructions are supported using floating-point accelerator blocks provided by Xilinx.

5.5.3 Register State

The functional partition Register State is responsible for tracking the architecturally visible registers. Because HAsim supports speculative and out-of-order timing models, the Register State must additionally track un-committed updates. As shown in Figure 5-14, this is accomplished by means of a freelist and physical register file. Architectural registers for each context are mapped onto physical registers. When the `getDependencies` operation is called it uses the maptable to lookup the latest mapping of source registers. These are the registers that will be read during `getOperands`. Additionally, it updates the maptable with new mappings for the destination registers. Additionally, the physical register which previously corresponded to that architectural register is recorded. When `commit` is called, that previously corresponding register may be returned to freelist. Alternatively, if `abort` is called then the new mapping is replaced with the previous one, and the new register returned to the freelist.

5.5.4 Memory State

The functional partition Memory State is responsible for retrieving instructions, translating virtual addresses, and maintaining a consistent view of program loads and stores. It accomplishes this by three components, shown in Figure 5-15. The first is a simple cache, used to accelerate simulation. The second is a store buffer, which uses the token age comparison method given above to ensure that any load will see the most recent store written to a particular address. The third is a functional Translation Buffer (TLB), used to cache virtual-to-physical address translations for timing model memory accesses. When an access misses in the TLB, the request is sent to

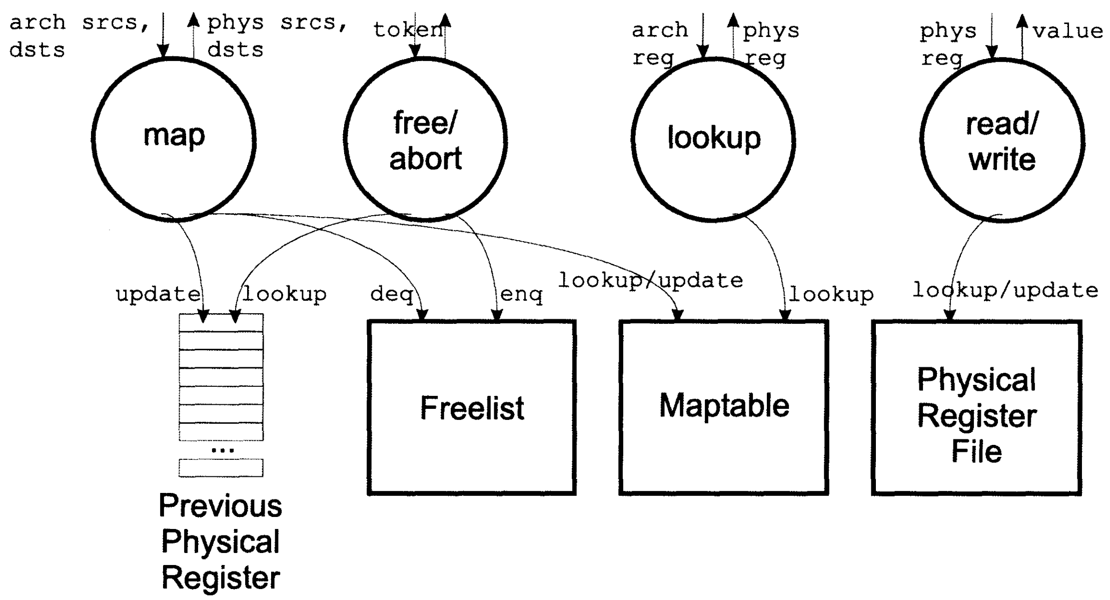


Figure 5-14: Functional Partition Register State

software, where the M5 [5] page translator is invoked. Note that this is separate from the timing model of the translation buffer, and does not affect the results of simulation. The timing model determines the timing of TLB-misses in a model-specific manner.

5.6 Timing Model Implementation

HAsim represents a framework for creating timing models, rather than any particular model. In this section we give examples of timing models that we have implemented in HAsim. These models do not represent any particular architecture, but rather stand as examples of how the functional partition is invoked to model a processor. In all of these examples, the modules are connected using A-Ports.

Core Models

Figure 5-16 shows the most basic processor timing model possible: an unpipelined "magic" processor that executes every instruction in a single clock cycle. The model is implemented by executing each functional partition operation before incrementing

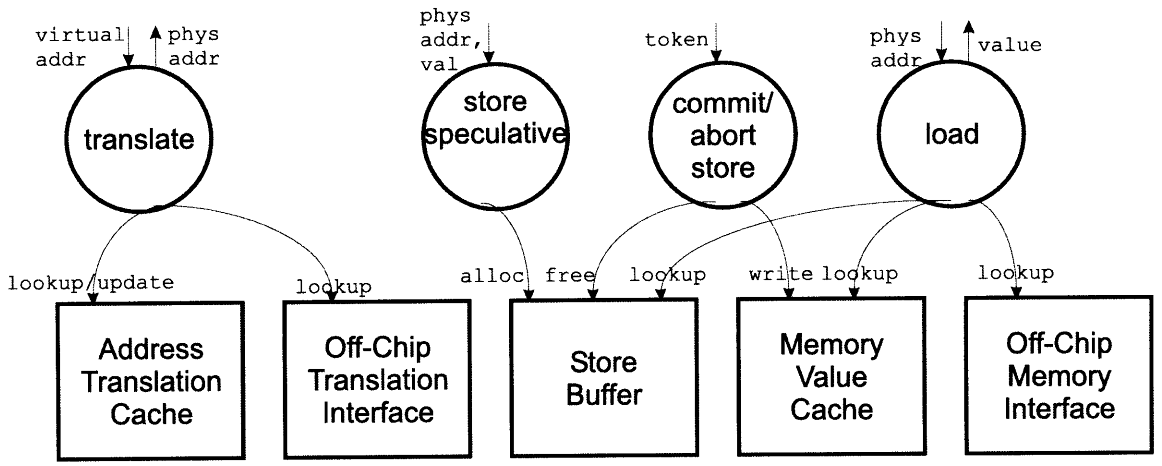


Figure 5-15: Functional Partition Memory State

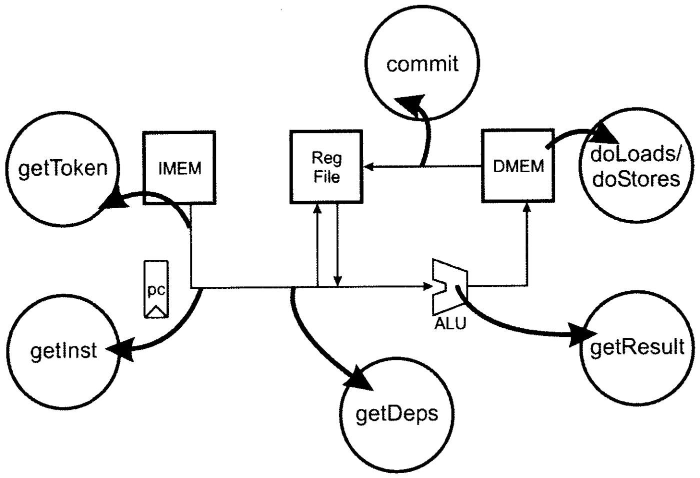


Figure 5-16: Unpipelined processor target.

model time. The ALU, IMEM, and DMEM, are not present, as they are simulated entirely via functional partition operations. The register file is also not present – as the model never has more than one outstanding instruction, the functional partition's register state is sufficient.

Figure 5-17 shows a more complex processor: a 5-stage pipeline similar to those commonly used in pedagogical examples. As with the unpipelined model, the IMEM, DMEM, ALU, and Register File are not present. The branch predictor structure is implemented entirely in the timing model, as it controls which address the timing model will pass to `getInstrution`. On mispredicts a `rewind()` is issued to represent a pipeline flush, and simulated no-ops are passed through the pipeline.

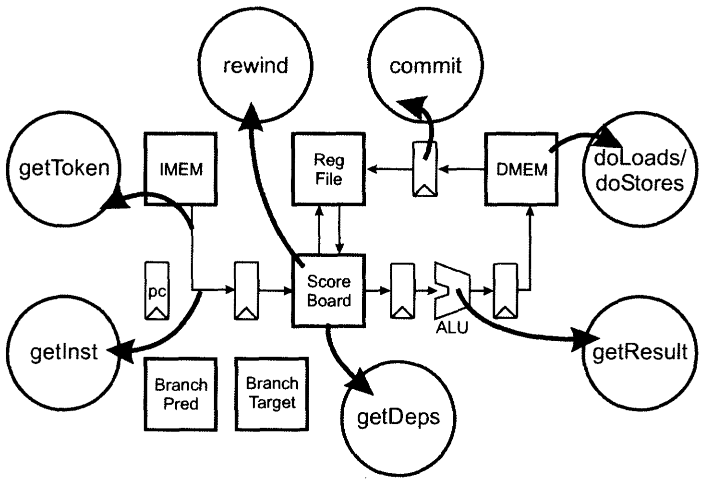


Figure 5-17: 5-stage processor target.

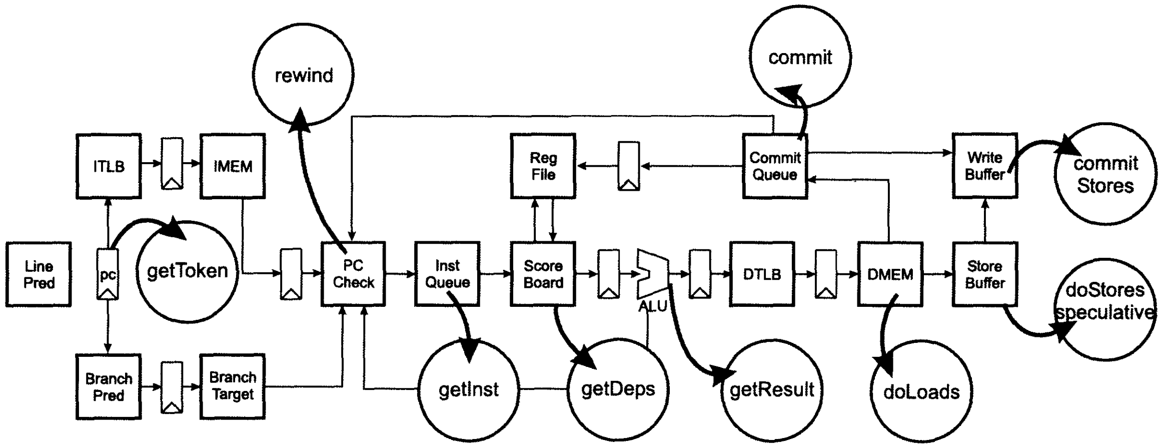


Figure 5-18: Detailed inorder processor target.

Figure 5-18 shows a more detailed inorder processor target. This expands the 5-stage pipeline into a more realistic inorder pipeline, including address translation, line prediction, faults, and cache retries. The PC Check stage resolves all possible simultaneous redirects using an epoch scheme. Redirects that only involve the front end (ICache retries, ITLB faults) do not trigger a `rewind()` in the functional partition. Speculative stores are placed in a 4-entry store buffer, then moved to a separate 4-entry write buffer when committed.

Figure 5-19 shows an out-of-order, 4-way superscalar target. The branch predictors are reused from the 5-stage model. The superscalar behavior is simulated by making multiple calls to the functional partition before advancing model time. The

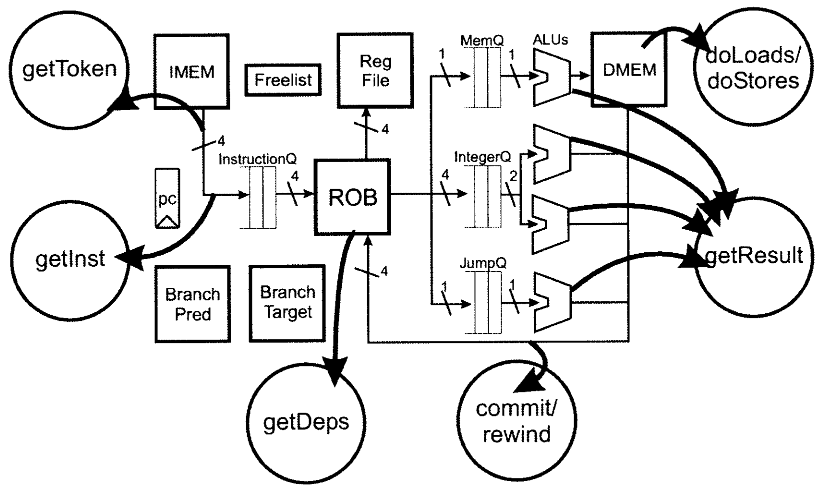


Figure 5-19: Out-of-order, 4-way superscalar target.

simulated ROB is substantially simpler than a real ROB, as it does not need to implement the dependency tracking logic. Instead it uses the result of the `getDeps()` operation and then uses a sequential search to determine which instructions should be issued next. The ALUs are not present, as they are represented by multiple calls to the `getResult()` operation.

Cache Models

HAsim models caches using the interface shown in Figure 5-20. In this scheme, requests are received from the processor, and are immediately answered with a response of `Hit`, `Miss`, or `Retry`. If a miss occurs then this response is accompanied from a `miss token` allocated from the Miss Address File (MAF). When a fill returns from memory it is associated with its miss token and returned on the separate fill port. This scheme allows responses to return from memory out-of-order (for example, if modeling a cache hierarchy). The processor model may make a local decision how to handle Miss and Retry events.

Note HAsim cache models are timing models only—the actual data associated with a load is returned by the functional partition. Therefore these models are only distinguished by the schemes they use to determine if an access hit or missed. Using this interface we have constructed four radically different cache models:

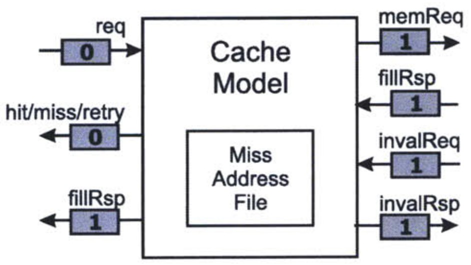


Figure 5-20: Interface for cache models.

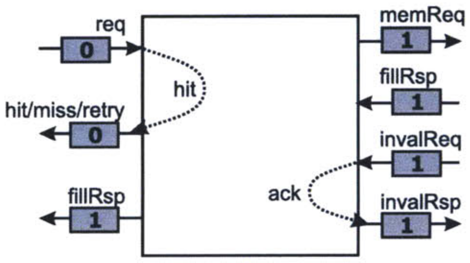


Figure 5-21: Null cache model that always hits.

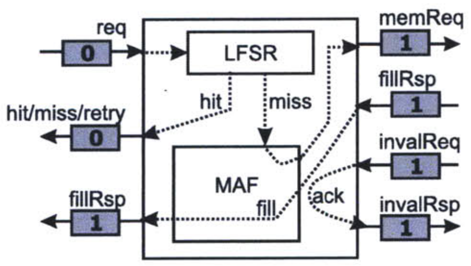


Figure 5-22: Pseudo-random cache model using Linear Feedback Shift Register.

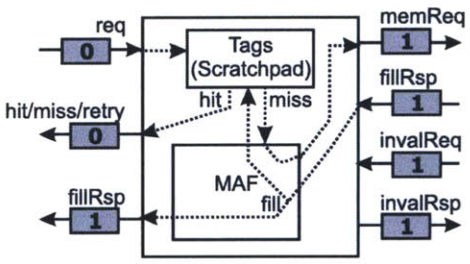


Figure 5-23: Direct-mapped cache model using LEAP Scratchpad to store tags.

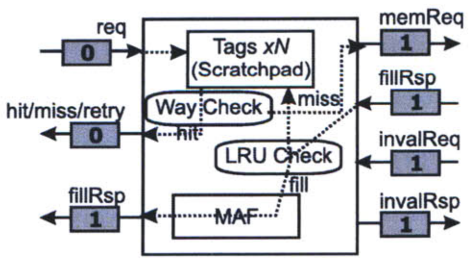


Figure 5-24: Set associative cache model using one-bit pseudo-LRU.

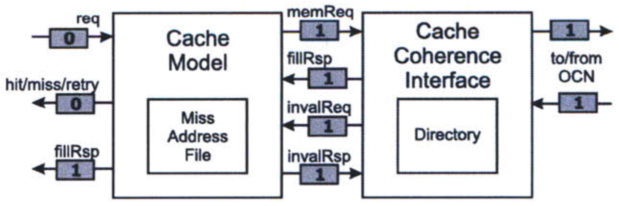


Figure 5-25: Cache model connected to a coherence interface.

- Figure 5-21: a null cache, or "magic memory." In this scheme all accesses are returned as hits.

- Figure 5-22: a pseudo-random cache that uses a Linear Feedback Shift Register to generate pseudo-random numbers. Hits or misses are determined by comparing this number to a threshold parameter.

- Figure 5-23: a direct-mapped cache that stores the tags in a LEAP scratchpad (Section 2.5.1). Size of the tag store is set as a static parameter.

- Figure 5-24: a set-associative cache parameterized both on tag size and number of ways. In this scheme the scratchpad stores a vector of tags, corresponding to the ways in the cache. Logic is used to search every way to determine if accesses hit or miss. When a fill occurs, a one-bit pseudo-LRU scheme is used to determine which entry will be flushed.

The cache models are designed in such a way as to be agnostic to the presence of a cache coherence protocol. As shown in Figure 5-25, the models may be hierarchically combined with a cache-coherence interface, which then interfaces with the on-chip network to generate invalidation requests.

Network Models

HAsim network models have a parameterized number of virtual channels, but it is assumed that a minimum of two virtual channels is required to model deadlock-free cache-coherence protocols. By default, the high-priority virtual channel is used for responses and the low-priority channel for requests. Using the permutation technique described in Chapter 4 we have constructed three different network models:

- A bus network model that does not require permutations, but simply sequentially reads and writes its ports.

- A ring network model using the permutations from Figure 4-6.

- A grid network model using the permutations from Figure 4-11.

All of these networks connect to a memory controller which represents off-chip memory accesses. Currently HAsim uses a simple latency-delay model to represent the time consumed by off-chip accesses. Addition of a more detailed model, such as a row/column DRAM-controller, is future work.

5.7 Discussion

Implementation of HAsim is a mostly matter of optimizing hardware structures for the specific constraints found on an FPGA. A key advantage of the HAsim approach is that this implementation work can be done by FPGA experts, leaving only the creation of the timing model itself for the architect.

It is our hope that HAsim is able to unlock the performance available in FPGAs, while not burdening developers with an undue amount of development effort. In the next chapter, we evaluate both HAsim's performance, and its impact on development effort.

Notes

[7]The functional partition is sometimes referred to as the "functional model" but this is an over-simplification. This term usually refers to a simple Instruction-Set Simulator (ISS). A general functional partition includes a large degree of functionality beyond this. More appropriate would be to call a functional partition paired with a timing model of a single-cycle "magic" processor a functional model.

Chapter 6

Assessment and Discussion

We assess HAsim by examining the impact our partitioning technique has on development effort. We assess single-core simulation performance, and the efficiency of A-Ports as a distributed simulation scheme. We assess the impact time multiplexing has on simulator scaling by comparing HAsim versus a traditional direct implementation. We present a small case study which serves both as a complete example of a HAsim multicore model, and as an argument for the importance of high-detail modeling. We examine simulator how simulator performance scales as more virtual cores are added to the time-multiplexed pipeline. Finally, we discuss options for future work, and conclude.

6.1 Impact of Partitioning on Development Effort

As we argued in Chapter 2, development effort is a key concern for FPGA-accelerated simulators. Here we attempt to gauge the impact HAsim's contributions have made on designer productivity. As we noted in Section 5.3, the timing-directed simulation scheme has a double impact on productivity. First, the designer does not have to re-implement the functionality of the instruction set for every new simulator. Second, the timing model actually does not need to implement all structures, as some of their functionality is handled by the functional partition, resulting in a further reduction in developer effort.

To assess this, we examined three HAsim configurations: the unpipelined "magic" core (Figure 5-16), the 5-stage pipeline (Figure 5-17), and the out-of-order core (Figure 5-19). Figure 6-1A shows a summary of the various structures in the target systems, and the degree to which they were present in the performance models. Almost all large processor structures are implemented within the functional partition. The timing model is responsible for controlling simulation, and thus implements the program counter and branch prediction. The functional partition operations (Section 5.4) eliminate the need for structures such as the register file and ALU. The dependencies tracking significantly eases the implementation burden of the issue logic, with the timing model generally tracking details like the number of model cycles operations consume.

Lines of code required to implement each partition are shown in Figures 6-1B and 6-1C. The out-of-order model Decode stage interacts with the branch predictor, which simplifies the Fetch stage despite its being 4-way superscalar. Similarly the complex ROB simplifies the out-of-order writeback. Memory ops are folded into the execute module.

Another motivation for the timing-directed scheme was to encourage reuse. Figure 6-1D summarizes the code reuse which was possible between the five-stage and out-of-order timing models. First off, the entire functional partition was reused with no changes. Within the timing partition, the greatest possibility for reuse came in the branch predictor, which was reused verbatim. This matches our intuition, as the behavior of a branch predictor is mostly separate from the surrounding pipeline. Partial reuse was possible in the fetch, execute, and data memory modules, which were essentially taken from the 5-stage pipeline and extended to more general, superscalar versions. No such reuse was possible in the decode stage or issue stages, where the out-of-order machine's ROB was different enough from the 5-stage pipeline's simple scoreboard to necessitate full re-implementation.

All in all, these results demonstrate that HAsim's partitioning scheme can result in a significant reduction in development effort, as the same functional partition can be re-used across radically different timing models.

Design	IMEM	Program Counter	Branch Predictor	Scoreboard/ ROB	Reg File	Maptable/ Freelist	ALU	DMEM	Store Buffer	Snapshots/ Rollback
Functional Part.	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Unpipelined	No	Yes	N/A	N/A	No	N/A	No	No	N/A	N/A
5-Stage	No	Yes	Yes	Partial	No	No	No	No	N/A	No
Out-of-Order	No	Yes	Yes	Partial	No	Partial	No	No	No	No

(a) Processor structure partitioning.

Datatypes	Token	Tables	Scoreboard	ALU	Register State	Memory State	Store Buffer
691	896		303	487	136	187	433

(b) Lines of Bluespec code to implement the functional partition.

Model	Fetch	Decode/Issue	Execute	Memory Ops	Writeback
Unpipelined	405				
5-Stage	156	190	148	138	93
Out-of-Order	79	953	157	N/A	30

(c) Lines of Bluespec code to implement the various timing models.

Functional Partition	Fetch	Branch Predictor	Decode	Issue	Execute	Memory Ops	Writeback
Full	Some	Full	None	None	Some	Some	Some

(d) Code reuse between the 5-stage pipeline and the out-of-order model.

Figure 6-1: Implementation of the partitioned model.

6.2 Single Core Simulator Characteristics

Let us consider the performance of HAsim modeling single-core processors by consideing three processor models. First, the unpipelined "magic" processor shown in Figure 5-16. Second, the more realistic in-order microprocessor pipeline shown in Figure 5-18. Third, the out-of-order superscalar processor shown in Figure 5-19. The cores were configured to run 64-bit Alpha ISA with floating point support. To maximize the differences between the processor pipelines themselves, the cores were paired with the one-cycle "magic" memory rather from Figure 5-21.

We synthesized all three configurations using Xilinx ISE 11.5, targeting a Nallatech ACP accelerator [40], which connects a Xilinx Virtex 5 LX330T FPGA to a host-computer via Intel's Front-Side Bus protocol. As a sanity check, we ran the targets on some sample SPEC 2000 integer and floating point benchmarks, as shown in Figure 6-2. 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 show that the out-of-order core executes between 1.3 and 2.5 times more instructions per clock, 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).

IPC represents the insight into the target design that most users of performance models care about. However, as simulator architects, we are also interested in comparative simulator performance. The average number of FPGA cycles each simulator requires to simulate a model cycle is given in Figure 6-3. For this metric the situation is reversed — the inorder simulator can simulate model clocks nearly 4 times faster than the out-of-order model, due to the multiplexing of the ALU which the out-of-order superscalar processor does during every model cycle. Similarly, the in-order model is faster than the unpipelined model due to being a better match for the parallelism in the model. The unpipelined model must fetch, execute, and com-

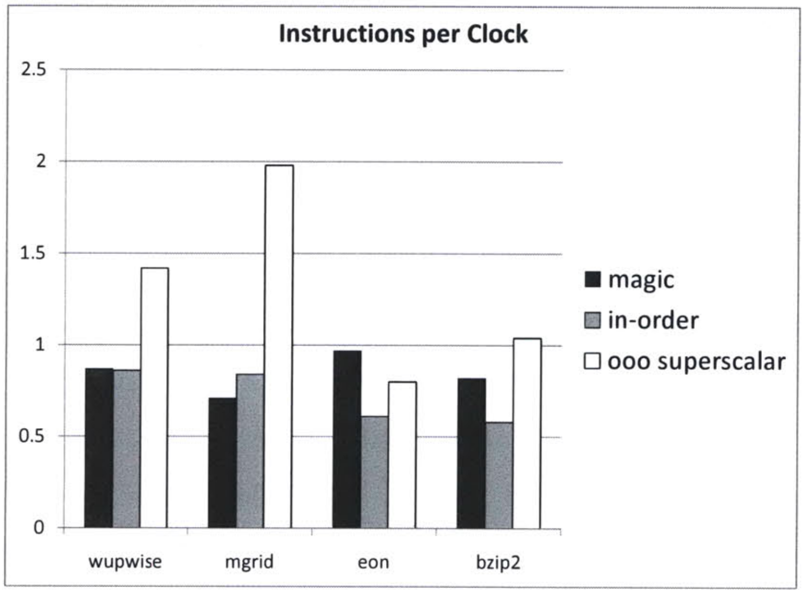


Figure 6-2: Assessing the target processors as a sanity check.

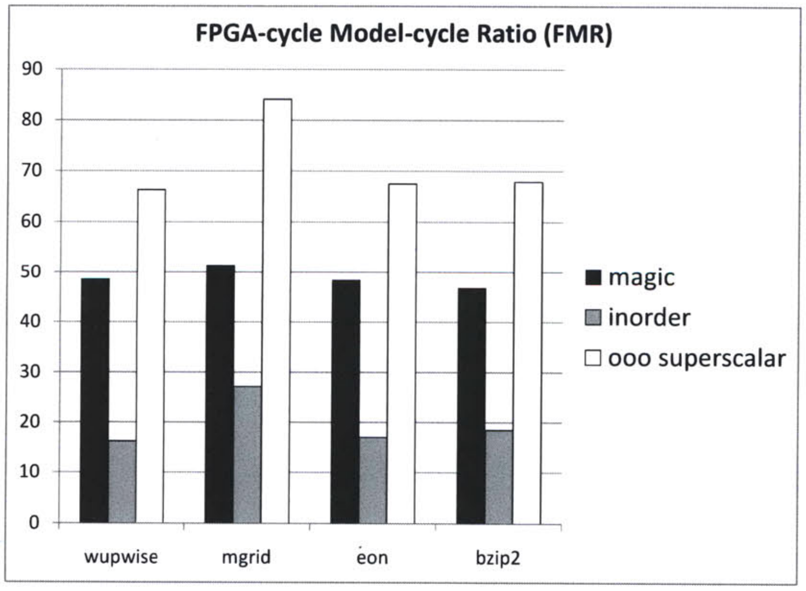


Figure 6-3: Simulator performance of the three models.

	Unpipelined	Inorder	Out-of-Order
Lookup-Tables	65,167 (24%)	89,937 (43%)	108.677 (52%)
Registers	51,364 (31%)	74,634 (35%)	88,898 (42%)
Block RAMs	69 (21%)	69 (21%)	70 (21%)
Clock Speed	75.0 MHz	75.0 MHz	75.0 MHz
Average FMR	48.8	19.7	71.5
Simulation Rate	1.54 MHz	3.81 MHz	1.05 Mhz
Average IPC	0.84	0.73	1.31
Average Simulator IPS	1.29 MIPS	2.75 MIPS	1.38 MIPS

Figure 6-4: Single-core simulator synthesis results for Virtex 5 330T.

mit an entire instruction before fetching the next instruction. The inorder model, in contrast, can fetch and execute different instructions in parallel.

Next let us consider the physical properties of each simulator, shown in Figure 6-4. When we combine clock speed with FMR we obtain the in-order model's simulation rate is almost four times faster than the out-of-order model. However, part of this difference is due to the increased presence of pipeline bubbles, which are fast to simulate. Combining clock speed with IPC we can consider simulated Instructions per Second. Here the situation is more balanced, ranging from 1.29-2.75 MIPS. This metric correctly compensates for the difference in target performance — remaining differences are due to the overhead of simulating out-of-order, superscalar execution.

Although these assessments were done on relatively simple cores without a memory hierarchy, our experience is that adding detail to these models does not 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 certainly require FPGA resources, FPGA cycles per model cycle generally remain relatively unchanged.

6.2.1 A-Ports and Simulator Performance

In order to assess the impact of the A-Ports scheme, we devised the following experiment: First, we took the models of the 5-stage pipeline and the out-of-order processor. We removed the A-Ports from the system and replaced them with simple registers. Coordination between the modules was acheived using the barrier synchronization

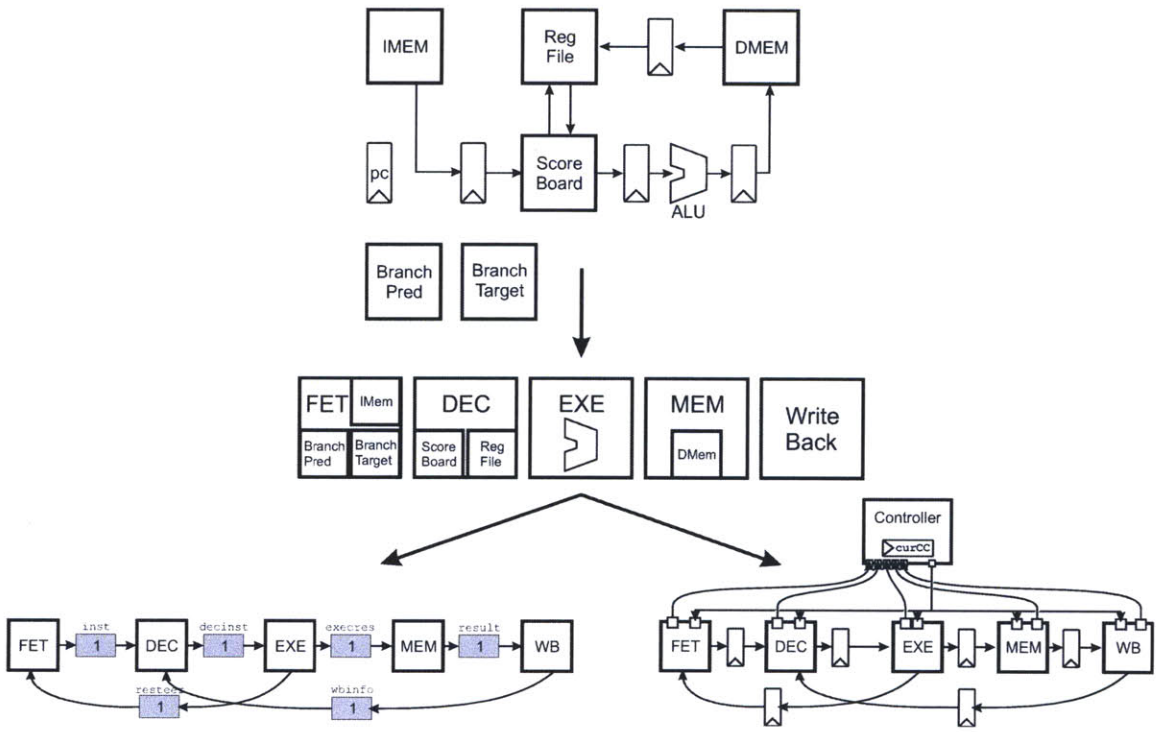


Figure 6-5: Implementing the 5-stage pipeline using A-Ports and barrier synchronization.

approach described in Section 4.1. Finally we compared the simulation rates of the two approaches.

The results of this experiment are shown in Figures 6-6 and 6-7. 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 3.5, the A-Ports buffer size limits the amount adjacent modules can slip in model time. Figure 6-8 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.

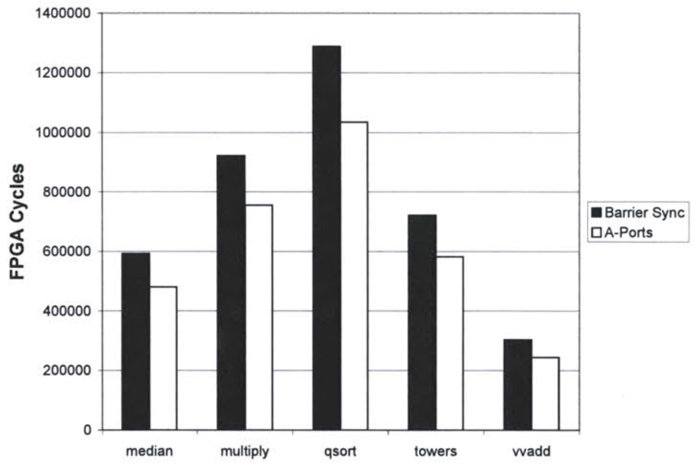


Figure 6-6: Assessing the in-order simulators.

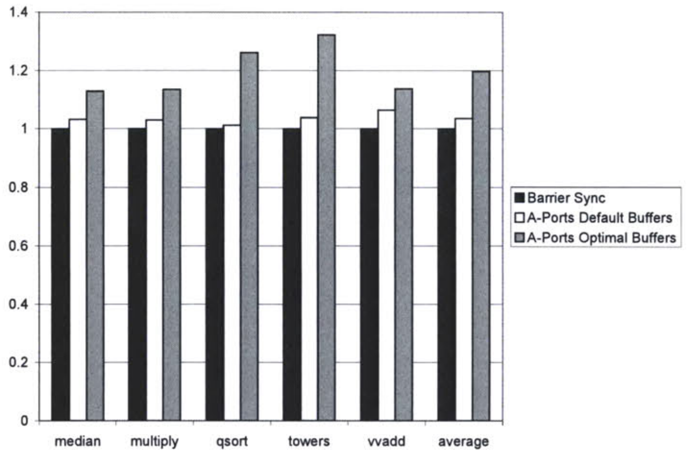


Figure 6-7: Assessing the out-of-order simulators.

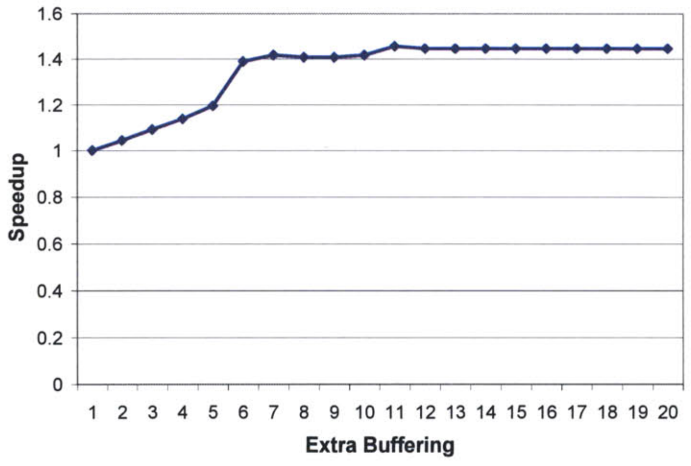


Figure 6-8: Out-of-order simulator performance improvement as buffering increases.

6.3 Multicore Simulator Characteristics

6.3.1 Muliplexing Versus Direct Implementation

In this section we compare HAsim's time-multiplexed approach to Heracles [30], a traditional direct implementation of a shared-memory multicore processor on an FPGA. Heracles aims to enable research into routing schemes by allowing realistic on-chip-network routers to be paired with caches and cores, and arranged into arbitrary topologies. Heracles emphasizes parameterization in an effort to fit in many different existing FPGA platforms. A comparison of a typical Heracles implementation and a typical HAsim model is shown in Figure 6-9.

We synthesized both configurations using Xilinx ISE 11.5, targeting a Nallatech ACP accelerator [40], which connects a Xilinx Virtex 5 LX330T FPGA to a host-computer via Intel's Front-Side Bus protocol. The resulting FPGA characteristics are shown in Figure 6-10. Heracles is specifically made for efficiency, but the FPGA synthesis tools still have a problem scaling a complete system with core, cache, and router. This is because duplicating Heracles' caches exceeds the FPGA's Block RAM capacity. The synthesis tool was able to complete even in the presence overmapping for the 2x2 and 3x3 configurations, but ran out of memory for the 4x4 case. We estimate that cache sizes would have to be reduced by a factor of 16 in order to successfully fit onto this FPGA.

In contrast, despite HAsim's significantly increased level of detail, we are easily able to fit a 4x4 multicore with L1 and L2 caches onto the same FPGA. This is due to four factors discussed earlier: First, separating the model clock from the FPGA clock allows efficient use of FPGA resources (Section 1.2). Second, use of off-chip memory allows large memory structures like caches to be modeled using few on-FPGA Block RAM (Section 1.2). Third, using a partitioned simulator allows HAsim to reduce the detail necessary in the timing model (Section 1.2): it is well-known that timing models of caches need to store tags and status bits, but not the actual data. Finally, and most significantly, the HAsim 4x4 model is actually a single physical core, single cache, and single router that has been time-multiplexed 16 ways (Section 4.2). The

	Heracles	HAsim
Core		
ISA	32-Bit MIPS	64-Bit Alpha
Multiply/Divide	Software	Hardware
Floating Point	Software	Hardware
Pipeline Stages	7	9
Bypassing	Full	Full
Branch policy	Stall	Predict/Rollback
Outstanding memory requests	1	16
Address Translation	None	Translation Buffers
Store Buffer	None	4-entry
Level 1 Instruction/Data Caches		
Associativity	Direct	Direct
Size	16KB	16 KB
Outstanding Misses	1	16
Level 2 Cache		
Size	None	256 KB
Associativity	None	4-way
Outstanding Misses	None	16
On-Chip Network		
Topology	Grid	Grid
Routing Policy	X-Y DO Wormhole	X-Y DO Wormhole
Virtual Channels	2	2
Buffers per channel	4	4

Figure 6-9: Component features of Heracles and HAsim.

	Heracles			HAsim
	2x2	3x3	4x4	4x4 (16-way multiplexed)
Registers	44,512 (21%)	65,602 (31%)	DNF	120,213 (57%)
Lookup Tables	33,555 (16%)	59,394 (28%)	DNF	165,454 (79%)
Block RAM	328 (101%)	738 (227%)	DNF	88 (27%)
Critical Path (ns)	7.3	14.0	DNF	19.9
Clock Rate (MHz)	139	71	DNF	50
FMR	1	1	DNF	16
Overall Rate	139 MHz	71 MHz	DNF	7.25 MHz

Figure 6-10: Scaling a direct implementation versus the multiplexing approach. Heracles performance is estimated assuming the design were able to fit on the FPGA. Note that 2x2 is a special case as no router requires a 5-way crossbar.

	Core Model		L1/L2 $	OCN	Func.	Infra-
	Detailed	Magic	Model	Model	Model	structure
Registers	24,052 (11%)	4,157 (2%)	28,425 (12%)	6,060 (2%)	34,586 (14%)	29,119 (14%)
Lookup Tables	35,728 (15%)	12,037 (6%)	47,739 (22%)	8,572 (4%)	50,895 (20%)	34,838 (16%)
Block RAM	0 (0%)	0 (0%)	6 (2%)	0 (0%)	50 (15%)	13 (4%)
Critical Path	7.4 ns	8.5 ns	7.72 ns	8.6 ns	8.8 ns	9.5 ns

Figure 6-11: Complete 16-way multiplexed HAsim simulator characteristics.

grid network is implemented using the side-buffer style permutations described in Section 4.2.2.

HAsim is an example of a space-time tradeoff. These techniques allow us to fit much more detail onto a single FPGA, paying for scaling by reducing simulation rate. Since at most one virtual instance can complete the physical pipeline per FPGA cycle, it takes a minimum of 16 FPGA cycles to simulate one model cycle. As the FPGA is clocked at 50 MHz, this gives HAsim a peak performance of $50/16 = 3.125$ MHz, multiple orders of magnitude faster to software-only industry models that are comparable levels of detail [13, 20].

6.4 Case Study: Effect of Core Detail on OCN Simulation

It is not uncommon for architects who wish to study an OCN topology to reduce the level of detail in the core pipeline for the sake of efficient simulation. In such a situation the architect is hoping that the ability to run an increased variety of benchmarks will offset the increased margin of error of each run. It our hope that FPGA-accelerated simulators will present an alternative to reducing fidelity. This idea is particularly appealing if the FPGA means that the extra detail has minimal impact on simulation rate.

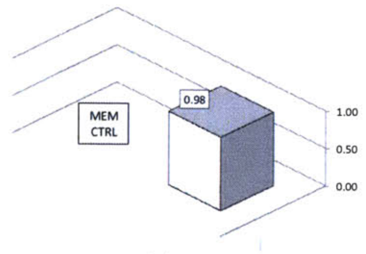
In order to evaluate the impact core fidelity can have on both simulation results and simulation rate, we modeled 2 multicore systems that differed only in the core pipelines. The first is a 1-IPC "magic" core running Alpha ISA that stalls on cache misses, similar to an architectural model. The magic core will never have more than one instruction in flight, and thus never produce more than one simultaneous cache miss. The second is the 9-stage pipeline described in Figure 6-9. This core does not reflect any particular existing architecture, but rather is representative of the general result of adding a higher-level of detail to the simulator.

Each core was then connected to the cache hierarchy described in Figure 6-9 and arranged into 4 different grid configurations: 1x1, 2x2, 3x3, and 4x4. In each case one of the nodes was occupied by the memory controller, so the 4x4 configuration consisted of 15 core/cache pairs and 1 memory controller.

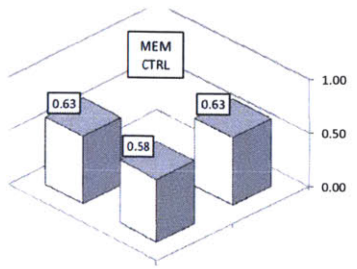
It is well-known that adding more cores to a shared-memory multicore can degrade the average IPC of each individual core, as contention on the OCN increases. This phenomenon represents a typical concern that an architect would like to characterize for a proposed OCN topology. We used HAsim to characterize the reported IPC of the individual cores running a variety of integer benchmarks, ranging from microkernels like Towers of Hanoi and vector-vector multiplication, to SPEC 2000 benchmarks gzip, mcf, and bzip2.

The results are given in Figures 6-12-6-14. They demonstrate that the reported IPC of a particular core varies 0.16-0.48 between the two models. The most variation was shown by core (1,0) in the 4x4 model—the core directly south of the memory controller. This is because in the detailed model the cores south and east of this core generate more OCN traffic, due to simultaneous outstanding misses. The dimension-order routing scheme overwhelms core (1,0)'s ability to serve its local traffic. In the undetailed model the reduced contention allows (1,0) to sufficiently warm up its caches to run without network accesses. An architect studying the detailed model might conclude to move the memory controller, or institute a different routing policy—insights that might be missed when using the magic core.

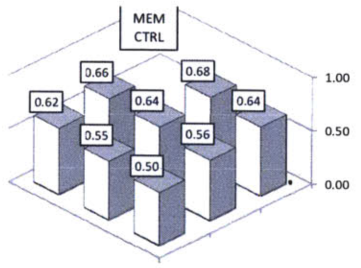
All in all, these results indicate that high-detail simulation will remain a useful tool in the computer architect's toolbox.



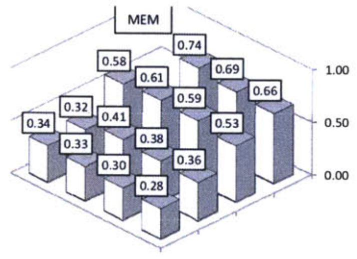
(a) 1x1



(b) 2x2

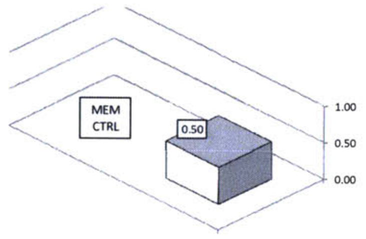


(c) 3x3

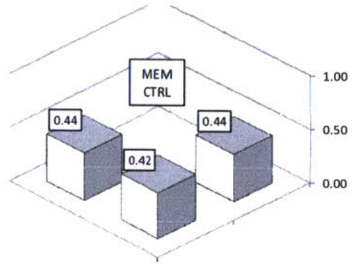


(d) 4x4

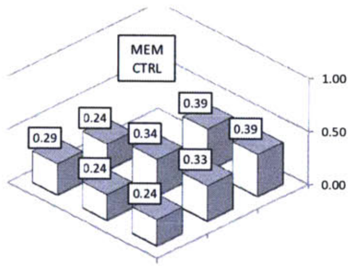
Figure 6-12: Per-Core IPC: Magic Core Grids



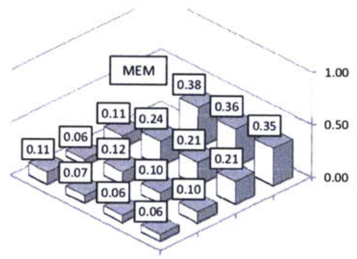
(a) 1x1



(b) 2x2

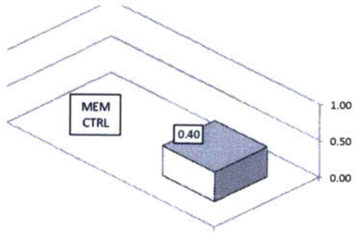


(c) 3x3

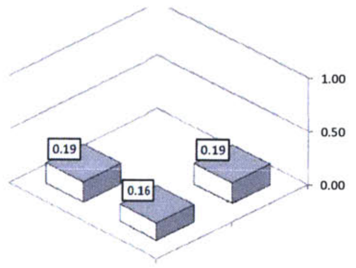


(d) 4x4

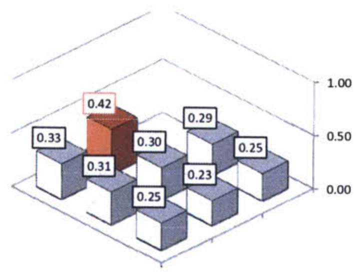
Figure 6-13: Per-Core IPC: Detailed Core Grids



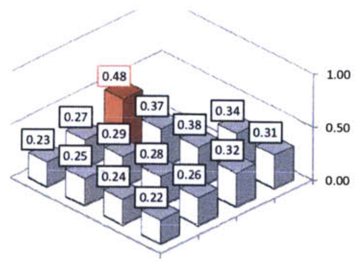
(a) 1x1



(b) 2x2



(c) 3x3



(d) 4x4

Figure 6-14: Absolute Difference in Reported IPC

6.4.1 Scaling of Simulation Rate

Now let us examine how HAsim's simulation rate scales as we add cores to the system. As explained in Section 4.1, HAsim simulates multicore processors by time-multiplexing a single physical pipeline between many virtual instances. In such a situation we can only finish the simulation for one virtual instance per model cycle. This means that simulating N processors has a best-case overall FMR of N, with a best-case *per-core* FMR of 1.

As given previously in Figure 6-2, the single-core in-order model takes an average of 19.7 FPGA cycles to simulate a model cycle. At first glance this seems to indicate that simulating N cores will reduce the FMR to $N * 19.7$. (FMR would scale linearly with the number of cores.) However, as noted in Section 4.1.2, HAsim's fine-grained multiplexing at the port granularity means that the modules themselves are implemented in a pipelined fashion. This pipelining can lower the impact of time-multiplexing. In the best-case scenario the FMR of 19.7 would mean that we could simulate 19 virtual cores without impacting FMR at all, as we could finish the simulation of a core per FPGA cycle.

Unfortunately the situation is not so simple. Adding more virtual cores to the system impacts the per-core FMR of individual cores. This is because:

- Virtual cores increase cache pressure on the LEAP scratchpad memories used to model the caches (Section 5.6). This can reduce the FMR of the cores (though note that it has no impact on the simulation results themselves).
- The round-robin multiplexing scheme described in Section 4.1 means that when a particular virtual instance stalls for an off-chip access, the amount of work the rest of the system can perform is limited. For example, if we are simulating a 4-core system and Core 0 has an off-chip access then we can only simulate Core 1, 2, and 3 before we are back to 0 and cannot proceed.

Thus in the worst-case simulation rate could actually scale *worse* than linearly with the number of cores. To test this phenomenon we used the time-multiplexed inorder core scaling between 1x1 and 4x4, as described in the previous section.

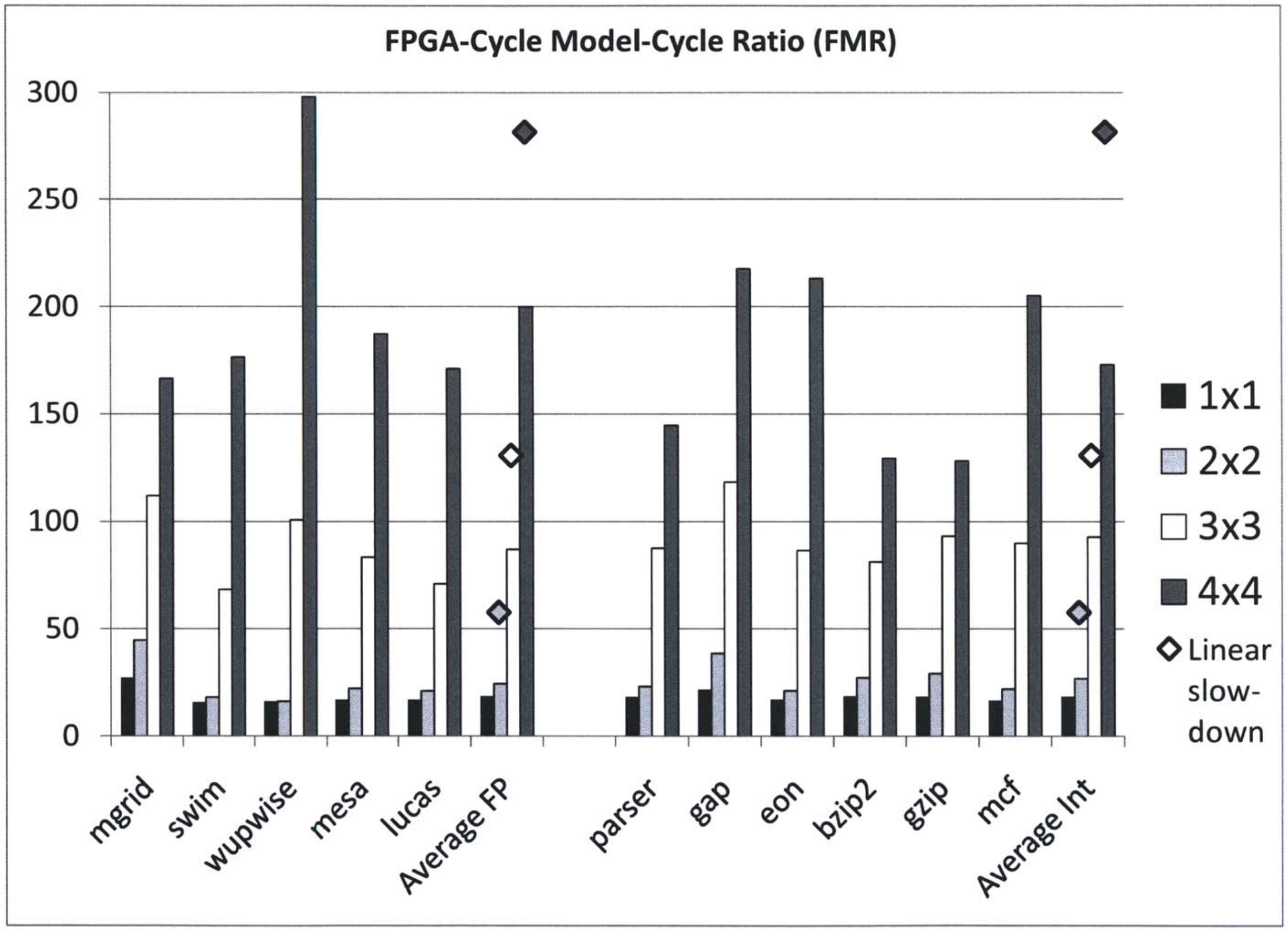


Figure 6-15: Impact on FMR of scaling inorder core to multicore. The diamonds represent linear slowdown compared to the FMR of a single core.

	FMR			FMR		
	Min	Max	Average	Min	Max	Average
Overall	16	218	80	160 KHz	3.2 MHz	625 KHz
Per-Core	5	27	11	1.84 MHz	9.5 MHz	4.54 MHz

Figure 6-16: Comparing overall simulation rate to per-core rates.

The results of this scaling are shown in Figure 6-15. There are several interesting features of this graph that are worth exploring. First, note that when we scale from 1x1 to 2x2, the performance impact is quite minimal. In fact, in the case of the `wupwise` benchmark HAsim actually acheives the best-case scenario of not reducing FMR at all. This is because `wupwise` has a small working set that exerts very little cache pressure. On average the additional cache pressure slows the 2x2 simulation by 46% over the baseline. On average, this is significantly better than linear a slowdown of 300%, which indicated by the diamonds on the graph. The fine-grained pipelining offsets the increased cache pressure, but not completely.

As we scale to 8 and 16 cores the increased cache pressure begins to have a greater impact. Although on aggregate we are still scaling better than linear slowdown, the difference is clearly reduced. One interesting case is `wupwise`, which goes from having the best 2x2 simulator performance to having the worst at 4x4. It seems that once this benchmark's working set no longer fits in the functional cache the impact is quite extreme.

A breakdown of per-core FMR and simulation rate is given in Figure 6-16. It demonstrates that although the fastest simulator runs at 3.2 MHz, the average is 625 KHz. However, this rate is because we are simulating so many cores. The per-core simulation rate averages 4.54 MHz, peaking at 9.5 MHz in the best case.

As simulation rates are almost entirely limited by off-chip accesses, current research is focused on improving hit rates in the host memory hierarchy, either by an improved cache algorithms, or using a hardware platform with larger on-board DRAMs, or providing faster access to host memory. An alternative approach would be to loosen the round-robin multiplexing in order to keep the FPGA busy longer when off-chip accesses occur. Currently, no scheme is known that results in better performance at an acceptable hardware cost.

6.5 Future Work

HAsim's approach to accelerating simulation demonstrates the potential FPGAs have for accelerating microprocessor simulation. Separating the FPGA cycle from the model cycle allows for much greater scaling of features that can be fit within a single FPGA. That being said, HAsim's scaling is still limited by finite FPGA capacity. As such, future extensions of the techniques presented here could be directed at scaling the target while limiting the impact on simulation rate.

6.5.1 Scaling to Thousands of Cores

If we wish to scale HAsim's models, it seems we are limited by the FPGA's finite capacity. However, there is no fundamental reason why the state of the target microprocessor must be stored on the FPGA. This is similar to a software simulator, which uses the operating system's virtual memory hierarchy to efficiently access the entire state of the target.

A similar approach could be used on an FPGA simply by storing the state of the target processors off the FPGA. The simulator would then simulate the next model cycle by loading the state of the target, performing local state updates, and writing the result back to the off-chip memory store. Using this technique scaling of the FPGA-accelerated model is no limited by the exact same factors as scaling a software model—adding extra state only affects the rate of simulation, but not whether or not the simulation fits on the FPGA.

It is possible that such an approach would undo some of the benefit that an FPGA has versus a conventional CPU. However we believe that the fine-grained parallelism within the FPGA will still lead to a benefit over pure software simulators. Using this technique we plan to scale the HAsim approach to simulating OCNs with thousands of cores.

6.5.2 Hardware Functional Partition

One of HAsim's major contributions is an architecture for implementing a generalized functional partition on an FPGA. This functional partition features a well-defined interface that can be used to simulate inorder, out-of-order, or superscalar processors. If more simulation speedup is required, then one possible approach would be to explore fabricating the functional partition as an ASIC. Under such a scenario the functional partition could benefit from increased speed and efficiency, possibly enough to offset the burden that the increased scaling described above would place on simulation rate.

In such a scenario we would want the timing model to retain the flexibility it has in the FPGA. Therefore, it would make sense to pair the "hard" functional partition with a fabric of reconfigurable logic that would host a HAsim-style timing model. It may also be possible to use the same fabric to augment the functional partition with new features—such as new instructions—that would not be otherwise possible if it were implemented as a full ASIC.

6.6 Conclusion

HAsim does not represent a simulation of any particular target processor running on any specific FPGA platform. Rather, HAsim represents a general framework for modeling target processors of greatly varying characteristics, and running those models on FPGA platforms with greatly varying underlying characteristics. HAsim places a large emphasis on easing development effort, as the process of developing for FPGAs remains significantly harder than developing software.

This thesis has made specific contributions to the HAsim framework, including:

- A hardware implementation of a generalized functional partitioning scheme. (Chapter 5.3)
- The A-Port framework for distributed control of simulation without a controller. (Chapter 3)

- A scheme for fine-grained time multiplexing of cycle-accurate processor models on a module-by-module basis. (Chapter 4)
- The Soft Connections abstraction to increase modularity in hardware description languages. Because this is a general technique not directly related to HAsim, it is presented in Appendix A.

Currently, the HAsim framework is being adopted within Intel to model experimental future-generation architectural features. Although the details of these modeling efforts have not been made public, the HAsim general framework, including the virtual platform, modeling library, and functional partition, have been made available under the General Public License (GPL). For more information please visit HAsim's homepage at `http://asim.csail.mit.edu`.

Appendix A

Soft Connections

Modularity is a key feature of hardware description languages (HDLs) for reducing development effort. Developing a model in HAsim entails frequent FPGA reconfigurations, as parts of the model are developed and refined. This is a direct contrast to ASIC production—the typical use case for an HDL–where designers work towards a single "golden model" for tapeout. In the FPGA environment ensuring that designers are able to swap alternative modules in a "plug-and-play" manner becomes even more important. Such swapping enables code reuse and design-space exploration, and thus enhances designer productivity.

In this section we describe a problem with modularity and HDLs that we encountered during HAsim's development. This problem was far-reaching enough that we developed a generalized solution named *soft connections*. Although HAsim is used as a motivating application, nothing in this chapter is restricted to the modeling of processors. The technique is general and could additionally be used for ASIC design instead of FPGA configuration. It is currently being applied to other AWB-based FPGA designs beyond processor simulation.

A.1 The HDL Modularity Problem

In structural HDLs it can be surprisingly difficult to swap one module for an alternative in isolation. This is because communication between modules can only follow the

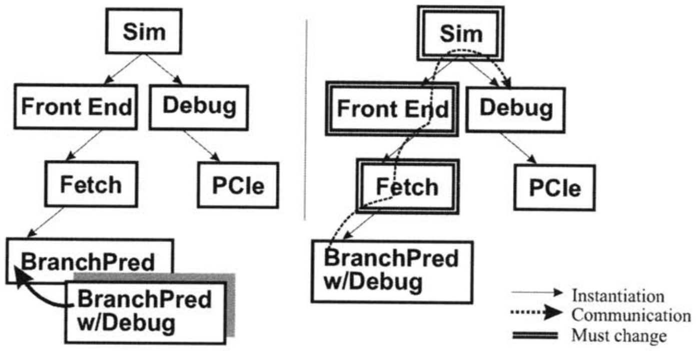


Figure A-1: Introducing cross-hierarchical communication.

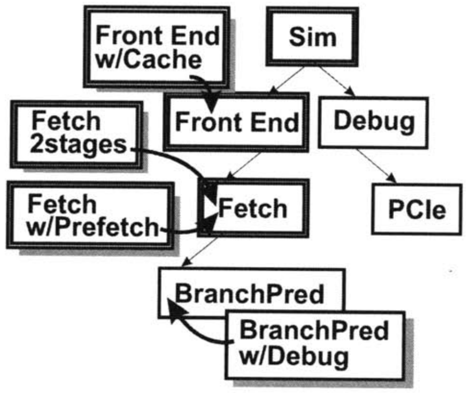


Figure A-2: Alternative modules can worsen the problem.

instantiation hierarchy. A module can only communicate to its parent and children. Thus, cross-hierarchical communication must always go through the least-common ancestor and every other intervening node. If a new module is inserted that requires communication with anything other than its direct parent, then we must change the parent module, the parent's parent, and so on.[8]

For example, consider the situation shown in Figure A-1. The designer knows that the Branch Predictor on the FPGA has a bug. He wants to swap it for a variant that outputs additional debugging information, that is sent to the host processor using PCIe. In order to do so he must add those wires to the Fetch, Front End, and top-level Simulator modules, then down to the Debug block.

The situation quickly deteriorates as we add more module alternatives to the system. In Figure A-2 we have three alternative Fetch units and two Front Ends that the designer is exploring. Each setup uses the branch predictor, and each manifests the bug in different ways. The designer must now produce alternative implementations of these, one of which does not pass PCIe wires up, and one which does. In the worst case the number of modules needed grows multiplicatively with the number of alternatives.

In this Chapter we attack this problem by "softening" the rigid communication hierarchy—thus we name our technique Soft Connections. This scheme restores modularity by allowing the designer to specify a *logical topology* of communication which is separated from its physical implementation. The endpoints are not connected by the user, but rather done automatically using static elaboration. Using Soft Connections restores modularity, allowing individual modules to be swapped in isolation, independent of the instantiation hierarchy.

A.2 Background: Static Elaboration

Our approach leverages the *static elaboration* phase of Bluespec compilation. During this phase:

- Polymorphic modules are instantiated with their specific types and parameters.
- Loops and recursive function calls are "unrolled" into combinational logic.
- Statically-known constants are propagated.

For example, the designer may describe a simple n-bit ripple-carry adder as follows:

```
function bit[n:0] addRC(bit[n:0] x, bit[n:0] y);
    bit[n:0] res = 0;
    bit c = 0;
    for (int k = n; k >= 0; k) begin
        res[k] = x[k] ^ y[k] ^ c;
        c = (x[k] & y[k]) | (x[k] & c) | (y[k] & c);
    end
    return res;
endfunction
```

The designer may then call this `addRC` function multiple times using different types. The HDL compiler will execute the function and its loop, using statically known values of n and k. If x and y are known statically then the function itself may result in no hardware, but rather a new static constant. However if x and y are dynamic inputs to the hardware block then the result is a netlist of AND- and XOR-gates. If for some reason n was dynamic, the result would be an error as the loop could not be turned into hardware.

Standard HDLs such as Verilog feature elaboration primarily through the use of *generate* blocks, which allow the user to create static control-flow structures such as loops and if-statements. Bluespec expands the power of the elaboration phase into a Turing-complete software interpreter. This allows the user to work with convenient high-level datatypes such as linked-lists or unbounded integers. These types may not have a hardware representation, but the designer can use them to influence the hardware that the compiler generates. For example, here is a Bluespec module that takes as inputs a list of integers. For each integer it instantiates a 32-bit FIFO of that depth (note that `<-` is the module instantiation operator in Bluespec):

```
module mkFIFOList#(List#(Integer) depths);
    let result_list = nil;
    while (depths != nil) begin
        Integer d = head(depths);
        FIFO#(bit[31:0]) q <- mkSizedFIFO(d);
        result_list = append(result_list, q);
        depths = pop(depths);
    end
    return result_list;
endmodule
```

This use of static elaboration could be thought of as "embedding a small software program in our hardware description source that the compiler runs to generate hardware." Soft Connections represent a novel use of static elaboration, and help to demonstrate how a more powerful notion of elaboration can benefit hardware designers.

A.3 Soft Connections and Logical Topologies

Point-To-Point Connections

Soft Connections are a library of communication primitives that the designer uses to describe a *logical* topology of communication. The basic Soft Connection is a point-to-point First-In-First-Out channel with a single logical sender and receiver.

Figure A-3 shows the branch predictor with debug information using a Send connection. This module sends debug information into the channel when it is trained with a misprediction, just as if it were enqueuing into a guarded FIFO. Elsewhere, the Debug module takes the debug information from the channel and transmits it to software using PCIe, as shown in Figure A-4. From the point of view of the module's usage, there is no difference between these soft connection endpoints and a familiar guarded FIFO.

Where Soft Connections differ is the instantiation of the channel itself. Instead of instantiating a single channel module and passing it to both of the users, the communicating modules instantiate the endpoints separately, naming the channel with a unique identifier. For example:

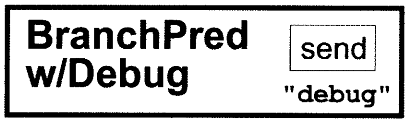
```
let link_to_debug <- mkConnectionSend("debug");
```

Elsewhere, the receiving module instantiates the dual endpoint:

```
let link_from_sender <- mkConnectionRecv("debug");
```

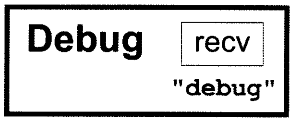
The modules do not need to instantiate one buffer and pass references to its interface through the hierarchy. Rather, the channel itself is instantiated during elaboration, as shown in Figure A-5. The physical buffering is instantiated by code in the Soft Connections library, rather than code written by the user.

As Soft Connections often represent communication between distant modules, we have chosen to implement them using a guarded buffer. Flow-control on the buffer is handled via Bluespec's standard guarded interface scheme [51], so that the producer's action may not be taken if the buffer is full, nor the consumer's if it is empty.



```
method Action train(BPredInfo inf);
   if (inf.branchTaken != table.lookup(inf.pc))
       link_to_debug.send(debugMsgMispredict(inf.pc));
   table.update(inf.pc, inf.branchTaken);
endmethod
```

Figure A-3: Using a Send connection.



```
rule debugToPCIe;
    let msg = link_from_sender.receive();
    pcie.transmit(pcieRequest(msg));
    link_from_sender.deq();
endrule
```

Figure A-4: Using a Receive connection.

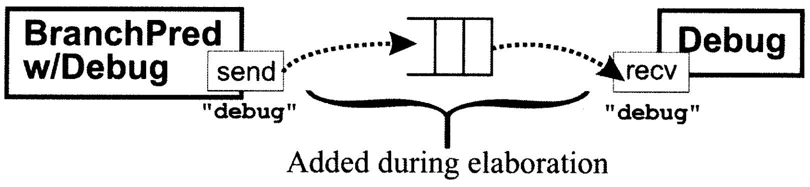


Figure A-5: Connecting a point-to-point channel.

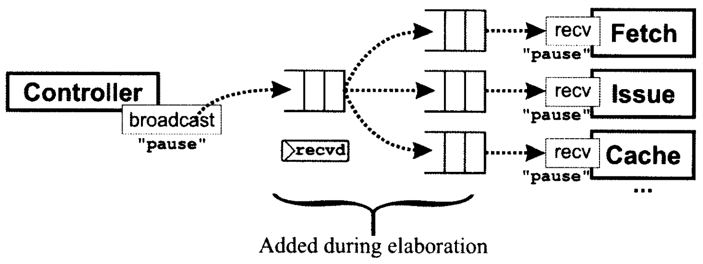


Figure A-6: A one-to-many connection.

If our connection algorithm finds no matching endpoint with the same name, the result is a compilation error. If an error is not desired either endpoint may be specified as optional:

```
let link_from_sender <- mkConnRecvOptional("debug");
```

An optional receiver with no corresponding sender will never receive data from the channel (as if it was connected to a sender who never sent a message). A sender that connects to a channel that that has no receiver may still enqueue messages — data the sender transmits will simply disappear, and the FIFO will never appear to be full. Both cases are similar to a wire unterminated on one side.

One-to-Many and Many-to-One

A one-to-many send is a broadcast channel that transmits the same data to all listeners (Figure A-6). When every receiver has gotten the data the main queue is dequeued. Note that the receivers connected to the channel are standard Point-to-Point receives, and may be agnostic to the fact that there are other receives listening to the same channel.

A many-to-one receive is a channel multiplexed by an arbitrator, that also tags the data with a bit field indicating which sender the data comes from (Figure A-7). These tags are assigned by our connection algorithm during compilation. Note that the senders themselves are standard point-to-point sends, and may be agnostic to the fact that there are other senders transmitting in this channel.

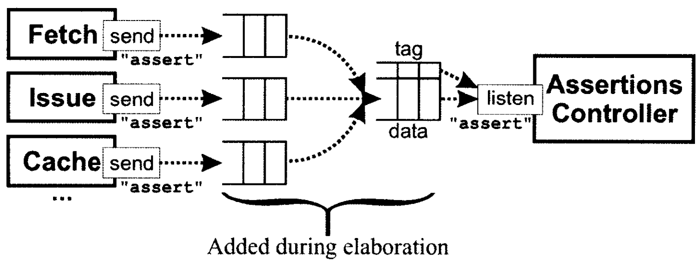


Figure A-7: A many-to-one connection.

Clients and Servers

The uni-directional channels presented above represent the primitive Soft Connections on which our elaboration algorithm operates. We then use these as building blocks to create useful abstractions for bi-directional communication. The first abstraction is that of a request/response paradigm (Figure A-8). The client has a request channel (send) and a response channel (receive). The server listens for requests and makes the appropriate responses. This arrangement is often used to connect functional units to their users. Note that the send and receive channels are standard one-to-one connections.

This idea can be combined with one-to-many and many-to-one connections to make multi-user clients and servers. A server with a many-to-one request channel can receive requests from multiple clients, and uses many one-to-one connections which deliver responses, writing the appropriate output channel based on the tag added to the incoming request (Figure A-9). Note that the clients that connect to this server are standard one-to-one clients, and may be agnostic of the fact that other clients are using the same server.

The dual of this is a client that is connected to many servers. It broadcasts its requests to all of them, then receives the responses in serial. This is a one-to-many send for the request channel, and a many-to-one receive for the responses (Figure A-10). Note that the servers are standard one-to-one servers, and may be agnostic of the fact that there are other servers receiving the same request.

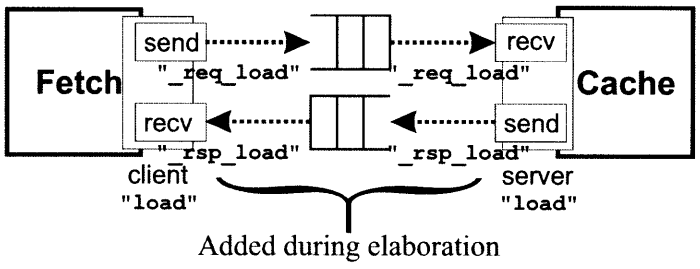


Figure A-8: Basic client/server abstraction.

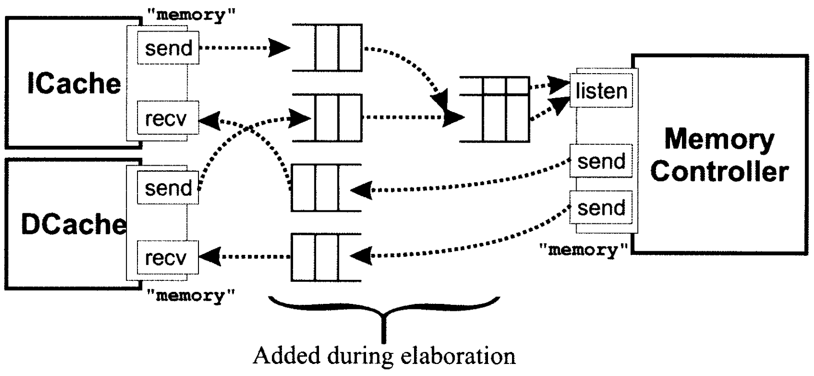


Figure A-9: A multi-user server.

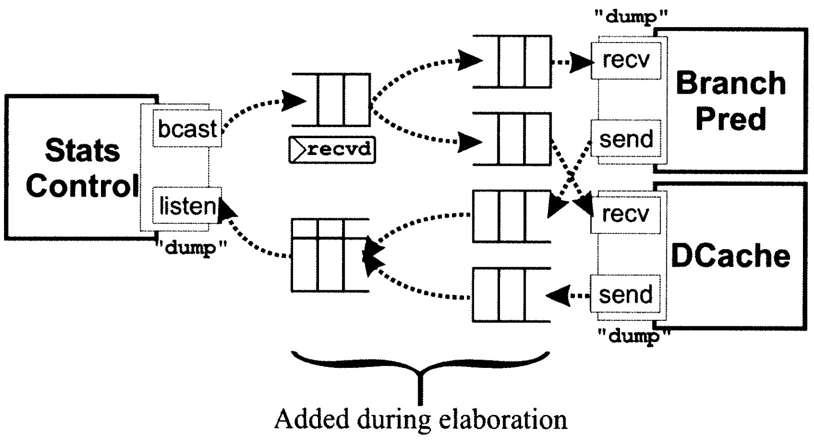


Figure A-10: A client connected to multiple servers.

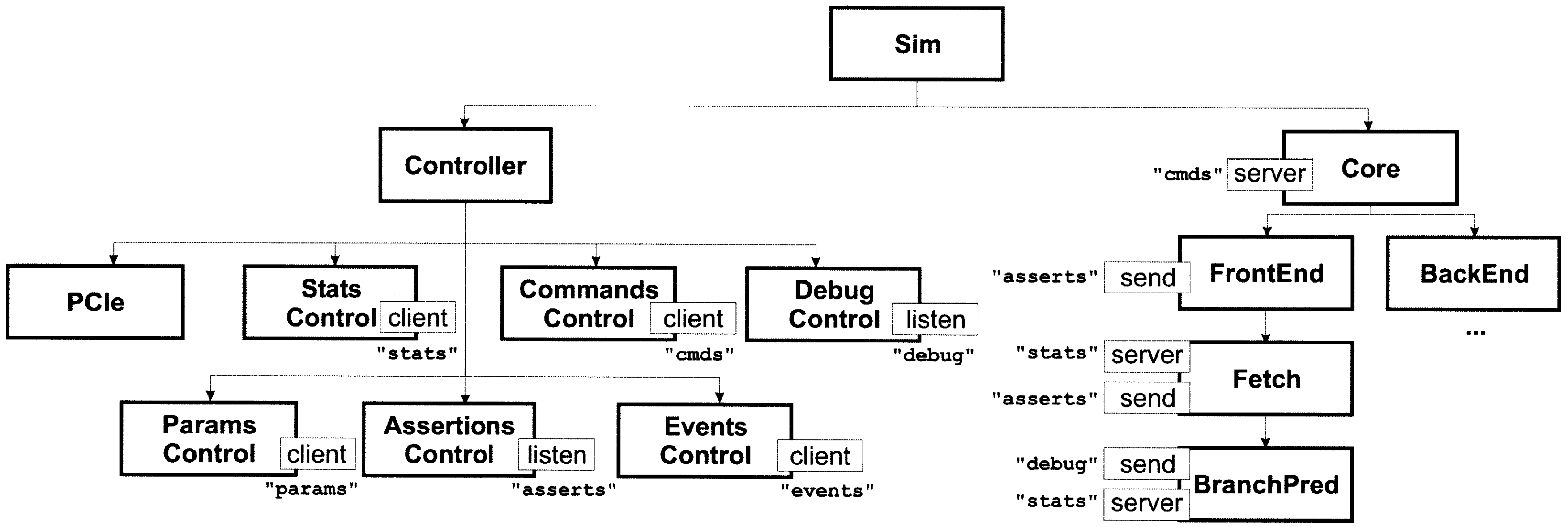


Figure A-11: Example: HAsim's simulation controller mediates the connection between host software and hardware modules.

A.3.1 HAsim's Simulation Controller

HAsim's simulation controller, presented in Figure A-11, represents an example of how Soft Connections can improve designer productivity. The controller is a module that sits on the FPGA and communicates with software on the CPU, mediating interaction with the virtual platform (Section 2.5). The controller instantiates six sub-controllers based on functionality:

- **Commands:** This receives commands from software such as "start" or "pause" and broadcasts them to listening modules. These modules respond when simulation is finished. Thus this module is a client of many distributed servers.
- **Params:** This receives dynamic parameters set on the command line when the user initiates the software. These parameters are broadcast to the appropriate listeners. Thus, for example, a timing model's cache can be disabled without re-synthesizing the design.
- **Events:** These represent a detailed trace of results from the simulator. The host software can enable or disable event-dumping dynamically.
- **Stats:** Periodically the host software can request a dump of statistics. This request is relayed to all listeners, who respond with their current values. The controller relays these to the host software.
- **Assertions:** When an assertion fails in a hardware module, it is sent to this controller, which relays the message to the host software, which prints out a message and ends the simulation gracefully.
- **Debug:** This module listens for debugging messages and relays them to the host software.

Using Soft Connections for the communication from these controllers to the simulator modules results in several benefits. First, the designer can fluidly swap modules without rewiring their connection to the controllers. This encourages users to create many variations of their module, without worrying that (for example) a direct-mapped write-through cache contains a smaller set of statistics than an associative write-back cache. Finally, it raises the level of abstraction for the user, who just records stats and assertion failures, without worrying about how this information is communicated to software.

A.4 Sharing A Physical Interconnect

Soft connections make life easier for the designer by making module communication implicit. The disadvantage of this is that the designer can lose intuition about the implementation cost of their communication network. For example, we have found that the assertions facility is useful for the FPGA in practice. Thus it becomes frequently used. A typical HAsim configuration has 42 dynamic assertions, most of them sanity checks relating to correct instruction execution. Implementing these as 42 FIFOs arbitrating directly with the controller would be expensive, and would place a large burden on the place-and-route tools due to the fan-in.

Assertion failures are (hopefully) a rare occurrence, so it makes sense to aggregate these using a multiplexed physical interconnect such as a tree. Such a scheme would increase the latency a message takes to reach the endpoint, but could result in more efficient hardware. Furthermore, other rarely-used connections such as statistics could also be mapped onto the same interconnect. Thus the user can separate a Soft Connection's *physical* representation (exclusive channel or shared interconnect) from its *logical* representation (point-to-point, one-to-many, etc).

The user creates a shared connection by first instantiating a network station. This station is then passed in to the constructor of the Soft Connection:

```
let fetch_station <- mkStationTree("fetch");
let link_to_assert <- mkConnSendShared(fetch_station, "asserts");
```

Whether a Soft Connection is implemented as an exclusive or shared interconnect is transparent to the modules which use the endpoints—they use the connection's

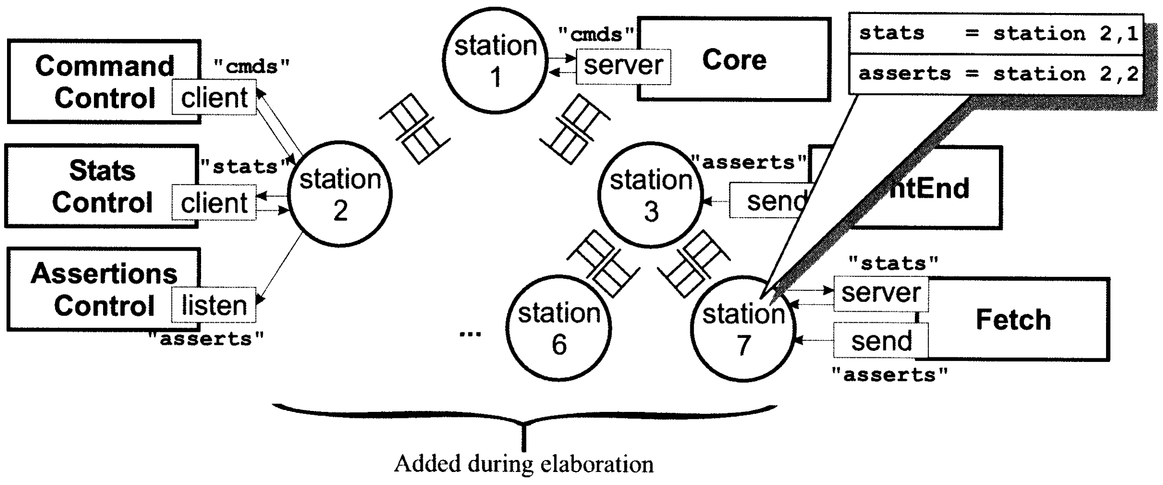


Figure A-12: Multiple logical Soft Connections implemented on a shared physical interconnect. Each station routes logical channels to the appropriate destination using a generated routing table.

operations (send, receive, broadcast, etc) as normal. The only difference from the user's point of view is that the latency of communication between sender and receiver has increased, as the messages are in fact being passed over an interconnect which is shared with other endpoints. Our elaboration algorithm connects the stations together into a physical network, and creates a routing table to dynamically guide messages to the appropriate destination. The addressing of messages is handled by the stations themselves. Figure A-12 shows an example mapping many Soft Connections onto the same shared interconnect.

Currently our algorithm connects the stations into a branching tree topology that follows the module instantiation hierarchy. (Layers in the hierarchy with no stations are optimized away.) This topology was chosen because it maximizes spatial locality by keeping the stations near their endpoints, and because it results in a single static route between two given endpoints, which minimizes station routing logic. In the future, support is planned for other physical network topologies such as rings, two-way rings, or grids.

Algorithm 1 Connecting Soft Connection endpoints directly.

```
(sends, recvs) = ... // Get collected info
for each s in sends do
    rs = matchByName(s, recvs)
    if rs = {} and not optional(s) then
        error("Unmatched Send " + s);
    else if rs = {r} and not manyToOne(r) then
        connect(s, r) // Instantiate buffering
    else
        connectBroadcast(s, rs) // As in Figure A-6
    recvs = recvs − rs
for each r in recvs do
    if manyToOne(r) then
        ss = matchByName(r, sends)
        if ss = {} and not optional(r) then
            error("Unmatched Receive " + r);
        else if ss = {s} then
            connect(s, r) // Revert to 1-to-1
        else
            connectListener(s, rs) // As in Figure A-7
        sends = sends − ss
    else
        error("Unmatched Receive " + r);
```

A.5 Connection Algorithm

When a module instantiates a Soft Connection endpoint it is implicitly transforming the interface it presents to the outside world. For a module with interface i, its new interface is a tuple of i plus linked lists that describe what Soft Connection endpoints the module has instantiated:

$$i' = (i, \{sends\}, \{recvs\})$$

The module's parent (and the parent's parent) see only the original interface i. This, along with collecting all the lists from all of the modules, is accomplished using a standard Bluespec library called `ModuleCollect`.

Algorithm 1 describes our process for connecting Soft Connection endpoints directly. Connections that are unmatched (and not optional) result in a compilation error via Bluespec's built-in `error` function, which halts elaboration. Note that although a user may declare a connection to be many-to-one or one-to-many, we may discover that during elaboration that it only has one sender and one receiver, in which case it reverts to the hardware for the cheaper one-to-one connection.

The algorithm for instantiating Soft Connections sharing a physical interconnect is most naturally described as a recursive module—it may call itself during elaboration, resulting in a tree-topology of stations connected to each other:

```
module mkStationTree#(STATION_INFO info) (STATION);
    List#(STATION) child_stations = nil;
    for (int x = 0; x < length(info.children); x++)
    begin
        let cur_child = info.children[x];
        // Recurse down the tree.
        let c <- mkStationTree(cur_child);
        child_stations = append(child_stations, c);
    end
    let table <- mkRoutingTable(child_stations,
                                info.recvs, info.sends);
    let s <- connectStation(table, child_stations,
                                info.recvs, info.sends);
    return s;
endmodule
```

The routing table is constructed mechanically using Algorithm 2. Note that one-to-many sends have the potential to be sent to multiple local receivers, and to multiple children. Additionally, they are always routed up to the parent (which drops the message if none of its other children are receivers). When we reach the root of the tree, endpoints that are unmatched result in an error, as in the basic point-to-point case.

A.6 Assessing Soft Connections

In this section we examine HAsim in order to give some insight into how Soft Connections can improve the process of engineering a complex real-world application. Figure

Algorithm 2 Constructing a station's routing table.

```
let ($childs$, $sends$, $recvs$) = ... // Parameters
// Routing decisions for traffic from local sends.
for each $s$ in $sends$ do
    if matchByName($s$, $childs$) = $\{cs\}$ then
        // A child (or its descendant) has the recv.
        sendRoute[$s$] := toChildren $cs$
    else if matchByName($s$, $recvs$) = $\{rs\}$ then
        // The endpoints are both local to this station.
        sendRoute[$s$] := toRecvs $rs$
    else // The endpoints are not in this subtree.
        sendRoute[$s$] := toParent
    if oneToMany($s$) then
        // oneToMany sends are always additionally routed to
        // our parent as they may have more receive points,
        // and to any children that have receive points.
        $cs$ = matchInStation($s$, $childs$)
        sendRoute[$s$] := {toParent, toChildren $cs$} + sendRoute[$s$]
// Routing decisions for traffic from children.
for each $c$ in $childs$ do
    // Find all sends this child could not match.
    for each $s$ in sendsRoutedToParent($c$) do
        if matchInStation($s$, $childs$) = $cs$ then
            // This station is the least-common ancestor.
            childRoute[$c$][$s$] := toChildren $cs$
            // Continues as above...
```

A-13 gives an overview of how the HAsim inorder model from Figure 5-18 uses Soft Connections. The connections are categorized according to the partitions shown in Figure 2-1. The overall prevalence of soft connections is a demonstration of their utility. The fact that so many soft connections cross partition boundaries supports the argument that adding a new traditional connection would result in changing a large number of intermediate interfaces.

In order to attempt to quantify the exact productivity a connection provides, we define a metric called *span*. For each connection c between two modules:

span(c) = the number of module instantiation boundaries between the send and receive endpoints.

Span measures the potential work the Soft Connection is saving the designer.

Category	Number
Intra-Timing	33
Intra-Functional	19
Intra-Infrastructure	20
Timing-Functional	24
Timing-Infrastructure	42
Function-Infrastructure	76
Unused Optional	3
total	217

Figure A-13: Number and use of Soft Connections in HAsim inorder model.

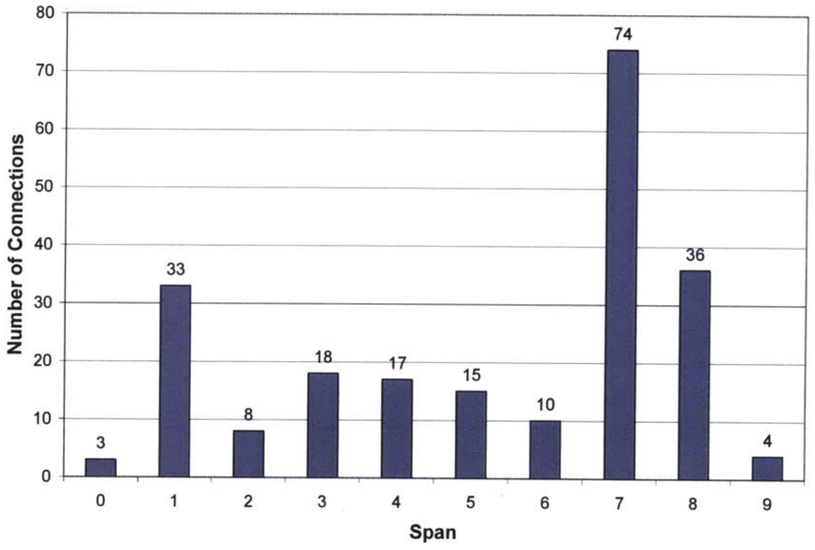


Figure A-14: Histogram of Soft Connection span in HAsim inorder model.

Namely, the number of modules that the designer would have to change if she was not using Soft Connections and swapped in a module with a different interface, or added a new connection between distant modules. We acknowledge the limitations of measuring the amount of work that our technique *potentially* can save, but believe that this metric gives valuable insight into the degree that communication between distant endpoints can exist in a hardware design.

Figure A-14 shows a histogram of the span of every connection in our simulator—i.e. our simulator contains 74 connections with a span of 7. Spans of 0 represent optional connections which are not being used. We found that the average Soft Connection in our simulator crosses 5.27 module instantiations, and that 50% of them cross 7 or more. This demonstrates that cross-hierarchical communication can

Benchmark	Model Cycles	FPGA Cycles		Change
		Baseline	Shared Tree	
test_164_gzip	7,612,202,736	120,866,746,639	120,848,407,550	-.0002%
test_176_gcc	4,412,926,919	97,284,304,169	96,331,305,044	-.001%
test_181_mcf	515,321,465	13,393,128,486	13,375,809,359	-.001%

Figure A-15: Running SPEC benchmarks on the shared interconnect version.

be prevalent in real-world situations.

Much of the cross-hierarchical communication—and all of the many-to-one/one-to-many connections—involve communicating data to or from the Simulation Controller (Section A.3.1). The cost of multiplexors between these signals can be high, and can result in a burden on the place and route tools. In order to explore this we implemented an alternative version of our simulator where all connections to the controller shared the same interconnect tree.

Overall 100/217 connections were mapped onto this tree, representing the statistics, assertions, commands, parameters, and events facilities. The tree had 14 stations arranged into a depth of 4, with the controller as the root node. All told, this tree spanned 20 module instantiations. We found this version consumes an additional 3076 slice LUTs (4% of total available) because of its extra buffering and routing tables. RAM utilization and clock speed are not affected, as the critical path is elsewhere.

Multiplexing these connections onto the same tree can increase the latency of communication. To measure the impact of this on dynamic performance we ran three SPEC benchmarks on each model. The results, shown in Figure A-15 demonstrate that over a run which spans billions of model cycles there was no measurable impact on performance—the differences in total FPGA cycles fall within expected run-to-run variation.

Notes

[8]We do not consider Verilog's Out-of-Module References (OOMR) to be a satisfactory solution, as they break modular abstraction. High-level HDLs such as SystemVerilog raise the level of abstraction so that the user moves around typed interfaces instead of wires, but the basic problem remains.

Bibliography

[1] Accellera. Standard co-emulation modeling interface (SC-EMI) reference manual, 2010.

[2] J. Babb, R. Tessier, M. Dahl, S.Z. Hanono, D.M. Hoki, and A. Agarwal. Logic emulation with virtual wires. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(6):609 –626, June 1997.

[3] Ken 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.

[4] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[5] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *52-60*, 26.4:52–60, 2006.

[6] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computing Systems*, 2(1):39–59, 1984.

[7] Bluespec Inc. `http://www.bluespec.com`, 2008.

[8] Randy Bryant. Simulation on a distributed system. In *First International Conference on Distributed Systems*, July 1979.

[9] 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.

[10] K. M. Chandy and J. Misra. Asynchronous parallel simulation via a sequence of parallel computations. *In Communications of the ACM*, pages 198–206, April 1981.

[11] C. Chang, J. Wawrzynek, and R.W. Brodersen. Bee2: a high-end reconfigurable computing system. *Design Test of Computers, IEEE*, 22(2):114 – 125, mar. 2005.

[12] Jianwei Chen, Murali Annavaram, and Michel Dubois. Slacksim: a platform for parallel simulations of cmps on cmps. *SIGMETRICS Performance Evaluation Review*, 37(2):77–78, 2009.

[13] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. H. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. Fpga-accelerated simulation technologies FAST: Fast, full-system, cycle-accurate simulators. In *MICRO*, 2007.

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

[15] E. Chung, E. Nurvitadhi, J. Hoe K. Mai, and B. Falsafi. Accelerating Architectural-level, Full-System Multiprocessor Simulations using FPGAs. In *FPGA '08: Proceedings Eleventh International Symposium on Field Programmable Gate Arrays*, 2008.

[16] F. Commoner, A.W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Science*, 5, 1971.

[17] John D. Davis, Chuck Thacker, and Chen Chang. BEE3: Revitalizing computer architecture research. Technical Report MSR-TR-2009-45, Microsoft Research, 2009.

[18] DRC Computer Corp. `http://www.drccomputer.com`, 2009.

[19] Kattamuri Ekanadham, Jessica Tseng, and Pratap Pattnaik. Ibm powerpc design in bluespec. Technical Report RC24706, IBM Research, 2008.

[20] 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*, pages 68–76, February 2002.

[21] J. Emer, C. Beckmann, and M. Pellauer. Awb: The asim architect's workbench. In *Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, June 2007.

[22] Kermin Fleming, Chun-Chieh Lin, Nirav Dave, Jamey Hicks, Gopal Raghavan, and Arvind. H.264 decoding: A case study in late design-cycle changes. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Anaheim, CA, 2008.

[23] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, jan. 2010.

[24] G. Gibeling, A. Schultz, and K. Asanovic. The RAMP Architecture & Description Language. Technical report, University of California, Berkeley, 2006.

[25] HiTech Global Design and Distribution, LLC. `http://www.hitechglobal.com`, 2009.

[26] Interra Systems. Bluespec testing results: Comparing RTL tool output to hand-designed RTL. `http://www.bluespec.com/images/pdfs/InterraReport042604.pdf`, April 2004.

[27] Tsuyoshi Isshiki, Dongju Li, Hiroaki Kunieda, Toshio Isomura, and Kazuo Satou. Trace-driven workload simulation method for multiprocessor system-on-chips. In

DAC '09: Proceedings of the 46th Annual Design Automation Conference, pages 232–237, New York, NY, USA, 2009. ACM.

[28] Daniel Jones and Nigel Topham. High speed cpu simulation using ltu dynamic binary translation. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 50–64, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] G. Kahn. The semantics of a simple language for parallel programming. *Information processing*, 1974. J.L Rosenfeld, editor.

[30] M. Kinsy. Heracles: Fully synthesizable parameterizable mips-based multicore system. Technical report, MIT, forthcoming (private communication).

[31] Christos Kozyrakis. Using fpgas for systems research: Successes, failures, and lessons. `http://ramp.eecs.berkeley.edu/Publications/ramp-retro.kozyrakis%20(Slides,%208-25-2010).pptx`, 2010.

[32] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre yves Droz. RAMP blue: a message-passing manycore system in FPGAs. In *In 2007 International Conference on Field Programmable Logic and Applications, FPL 2007*, pages 27–29, 2007.

[33] E. Larson, S. Chatterjee, , and T. Austin. MASE: A novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, November 2001.

[34] Benjamin C. Lee, Jamison Collins, Hong Wang, and David Brooks. Cpr: Composable performance regression for scalable multiprocessor models. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 270–281, Washington, DC, USA, 2008. IEEE Computer Society.

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

[36] Sungjin Lee, Kermin Fleming, Jihoon Park, Keonsoo Ha, Adrian M. Caulfield, Steven Swanson, Arvind, and Jihong Kim. BlueSSD: an open platform for cross-layer experiments for NAND flash-based SSDs. In *The 5th Workshop on Architectural Research Prototyping (WARP)*, Saint-Malo, France, 2010.

[37] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, Pengju Ren, Omer Khan, and Srinivas Devadas. Darsim: a parallel cycle-level noc simulator. In *Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, June 2010.

[38] C.J. Mauer, M.D. Hill, and D.A. Wood. Full-system timing-first simulation. *ACM SIGMETRICS Performance Evaluation Review*, 30.1:108–116, 2002.

[39] J.E. Miller, H. Kasture, G. Kurian, C.Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, January 2010.

[40] Nallatech, Inc. `http://www.nallatech.com`, 2009.

[41] OSCI. Systemc language reference manual version 2.1.

[42] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Processing*, 12:87–107, 1996.

[43] A. Parashar, M. Adler, M. Pellauer, and J. Emer. Hybrid cpu/fpga performance models. In *Workshop on Architectural Research Prototyping (WARP)*, June 2008.

[44] Angshuman Parashar, Michael Adler, Kermin E. Fleming, Michael Pellauer, and Joel Emer. Leap: A virtual platform architecture for fpgas. In *To Appear in Proceedings of The First Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2010)*, 2010.

[45] David Patterson. RAMP in retrospect. `http://ramp.eecs.berkeley.edu/Publications/RAMPretroPatterson%20(Slides,%208-25-2010).ppt`, 2010.

[46] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. A-Ports: An efficient abstraction for cycle-accurate performance models on FPGAs. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, February 2008.

[47] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. Quick performance models quickly: Closely-coupled timing-directed simulation on fpgas. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2008.

[48] 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 *The 12th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2006.

[49] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *SIGMETRICS Performance Evaluation Review*, 31(1):318–319, 2003.

[50] G. Pfister. The yorktown simulation engine. In *19th Conference on Design Automation (DAC)*, 1982.

[51] Daniel L. Rosenband and Arvind. Modular Scheduling of Guarded Atomic Actions. In *Proceedings of DAC'04*, San Diego, CA, 2004.

[52] Charles Selvidge, Anant Agarwal, Matt Dahl, and Jonathan Babb. Tiers: Topology independent pipelined routing and scheduling for virtualwire compilation. In *FPGA '95: Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 25–31, New York, NY, USA, 1995. ACM.

[53] L. Soule and A. Gupta. Parallel distributed-time logic simulation. *IEEE Design and Test*, pages 32–48, November 1989.

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

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

[56] Chuck Thacker. Beehive: A many-core computer for fpgas (v5), 2010.

[57] M. Vijayaraghavan and Arvind. Bounded Dataflow Networks and Latency-Insensitive Circuits. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, 2009.

[58] J. Wawrzynek, D. Patterson, M. Oskin, S. L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. Ramp: A research accelerator for multiple processors. *IEEE Micro*, Mar/Apr 2007.

[59] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. A practical fpga-based framework for novel cmp research. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 116–125, 2007.

[60] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. *SIGARCH Computer Architecture News*, 31(2):84–97, 2003.