Register Transfer Level Simulation Acceleration via Hardware/Software Process Migration

Aric David Blumer

Dissertation submitted to the faculty
of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Cameron Patterson, Chair

Peter Athanas
Robert Broadwater
Mark Jones
Henning Mortveit

October 15, 2007
Blacksburg, Virginia

Keywords: Reconfigurable Computing, Run-time Reconfiguration,
Process Migration, RTL Simulation, Formal Modeling, Canonical RTL,
Executive Locality of Reference

Register Transfer Level Simulation Acceleration via Hardware/Software Process Migration

Aric David Blumer

Abstract

The run-time reconfiguration of Field Programmable Gate Arrays (FPGAs) opens new avenues to hardware reuse. Through the use of process migration between hardware and software, an FPGA provides a parallel execution cache. Busy processes can be migrated into hardware-based, parallel processors, and idle processes can be migrated out increasing the utilization of the hardware. The application of hardware/software process migration to the acceleration of Register Transfer Level (RTL) circuit simulation is developed and analyzed. RTL code can exhibit a form of locality of reference such that executing processes tend to be executed again. This property is termed *executive temporal locality*, and it can be exploited by migration systems to accelerate RTL simulation.

In this dissertation, process migration is first formally modeled using Finite State Machines (FSMs). Upon FSMs are built *programs*, *processes*, *migration realms*, and the migration of process state within a realm. From this model, a taxonomy of migration realms is developed. Second, process migration is applied to the RTL simulation of digital circuits. The canonical form of an RTL process is defined, and transformations of HDL code are justified and demonstrated. These transformations allow a simulator to identify basic active units within the simulation and combine them to balance the load across a set of processors. Through the use of input monitors, executive locality of reference is identified and demonstrated on a set of six RTL designs. Finally, the implementation of a migration system is described which utilizes Virtual Machines (VMs) and Real Machines (RMs) in existing FPGAs. Empirical and algorithmic models are developed from the data collected from the implementation to evaluate the effect of optimizations and migration algorithms.

Dedication

To my wife Jamie
and to my three children,
Daniel, Miranda, and Lydia,
for supporting me throughout this work
and for understanding when I needed
absolute quiet.

Acknowledgements

I am sincerely grateful to
Mrs. Marion Bradley Via
and the College of Engineering
for the fellowships which funded this research.

Special thanks also goes to
Dr. Cameron Patterson, my advisor.
When I felt like I had wasted a couple years
pursuing an unfruitful topic,
he encouraged me to continue.

I extend my gratitude as well
to the remaining committee members
for serving on my committee and
for advice on the dissertation topic,
especially to Dr. Henning Mortveit
for his guidance on the formal model.

Table of Contents

List of Figures

List of Tables

List of Code

Chapter 1

Introduction

The capacity of integrated circuits has increased significantly in the last decades, following a trend known as "Moore's Law." Gordon Moore predicted in 1965 that transistor counts would double approximately every two years, and indeed they have as shown in Figure 1.1 [1]. The code bases describing these circuits are likewise becoming larger, and the design engineer is stressed by the opposing requirements of time to market and bug-free silicon. Some industry observers estimate that every 18 months the complexity of digital designs increases by at least a factor of ten [2], with verification time taking at least 70% of the design cycle [3]. Compounding the problem is the non-recurring expense of producing the circuits—65 nm designs are approaching $40 million [4]—which makes the goal of first-time working silicon paramount. Critical to satisfying both time to market and design quality is the speed at which circuits can be simulated, but simulation speed is "impractical" or at best "inadequate" [5, 6].

In an effort to address this issue, engineers are looking for creative ways to improve verification times for designs. In one instance, the boot time of a cell phone took thirty days to simulate using traditional, software-based simulation methods. By using an instruction set simulator (ISS) the time was reduced to ten days, and a reduced-detail model coded in C lowered the boot time to one day [7]. As in the software industry, the efforts to mitigate the effect of growing hardware complexity have included greater levels of abstraction, allowing faster modeling speeds (as demonstrated by the cell phone application) and smaller code bases.

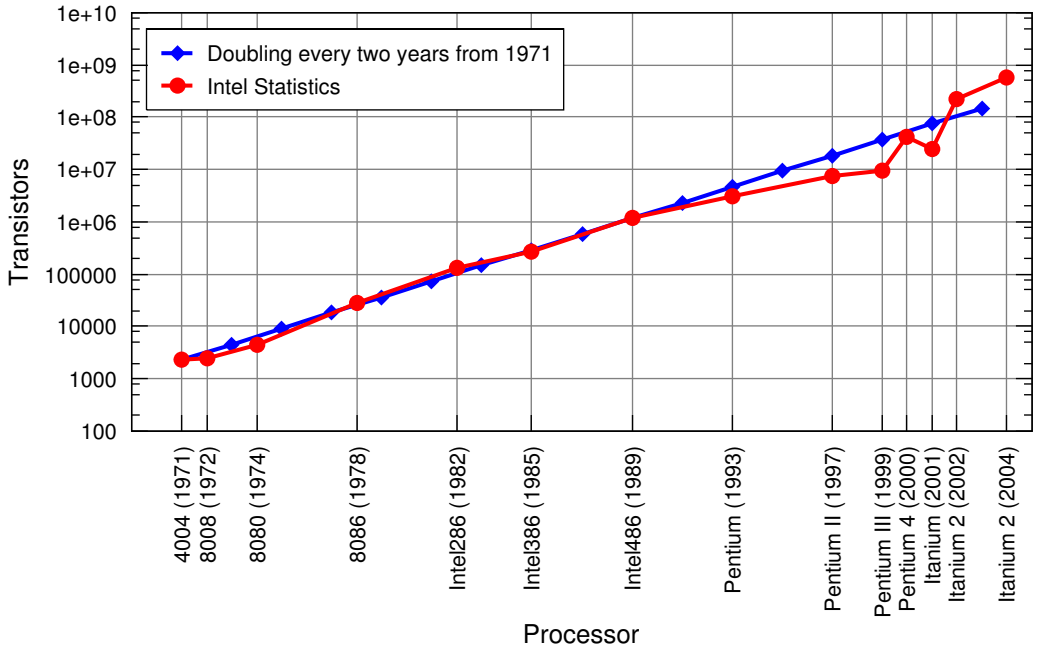


Figure 1.1. Intel Processors Demonstrating Moore's Law [1]

Circuits were, in times past, described by entering components into schematics with CAD tools, but the complexity of the circuits became great enough that a higher level of description, known as Register Transfer Level (RTL), has become the norm. The continuing increase in complexity has spurred the development of abstraction levels above RTL, such as Transaction Level Modeling (TLM) [5]. By definition, higher abstraction levels contain fewer details than lower abstraction levels for the sake of easier management, quicker operation, or easier conceptualization. In the realm of digital design, however, the details are critical because of constraints on timing and area. Software tools that supply the missing details of TLMs are not yet as efficient as human-coded RTL [8, 9]. Therefore, models with reduced detail are not yet suitable for design descriptions targeted for fabrication, and RTL descriptions still form a necessary part of current design flows.

Another option to improve verification times is hardware acceleration. For the cell phone booting example above, a hardware-based prototype took three seconds [7]. Prototyping in hardware, however, drastically reduces the design's debugging visibility, and it incurs significant overhead to compile the design to the prototype hardware. The author of [7], who is promoting a hardware prototyping product called TotalRecall™, conveniently leaves

the compile times out of his performance statistics. Hardware acceleration provides a middle ground between software simulation and hardware prototyping, but has traditionally suffered from high cost as well as long compile times. A hardware acceleration solution for RTL simulation having short compile times and lower cost is needed.

The fundamental component of hardware-based acceleration systems is the Field Programmable Gate Array (FPGA). FPGAs may be configured many times with different functionality, making them a good fit for hardware-based accelerators, which utilize arrays of them to implement the functionality under test. But FPGAs are designed to be statically configurable. That is, the power-up or reset configuration of the FPGA can change, but the configuration is static while the device is running. As a consequence, existing accelerators statically map the design to the FPGA array. If the design under test is large, a large array of FPGAs is required to accelerate its simulation, and a long compile time is required to map it to the array.

It is possible, however, to change the configuration of an FPGA at run time, and this is aptly termed *run-time reconfiguration* (RTR). RTR is a means of increasing the run-time flexibility of computing systems—not only simulation accelerators—and one application is the migration of processes between hardware and software. A run-time reconfigurable logic array behaves as a sandbox in which one can construct and tear down structures as needed. By instantiating and removing processors in the array during run time, users can establish a dynamically adaptable parallel execution system. It can be tuned by migrating processes into and out of processors in the FPGA and establishing communication routes between them on demand. The processors themselves can be reconfigured to support a minimum set of operations required by a process, thus reducing area usage in the FPGA. Area can also be reduced by instantiating a shared functional unit that processes migrate to and from when needed. Furthermore, time-consuming or often-executed processes can be Just-In-Time (JIT) compiled directly to hardware for further speed and area efficiency.

Because of the run-time reuse that RTR provides, hardware accelerators may be able to improve simulation performance with less hardware (and therefore less cost). Process migration in particular is a good complement to simulation acceleration for two reasons. First, RTL simulation consists of a set of parallel processes that can be decomposed into small

migration units. Second, since simulation processes often describe synthesizable hardware, they can also be compiled directly to the hardware at run time.

Considering the run-time reuse of FPGAs, the process migration that RTR enables, and the parallel processes that compose RTL simulation, a hardware-accelerated simulator takes shape. The system can migrate busy processes to an FPGA and idle processes from it to increase utilization. Because the processes in the FPGA execute in parallel, a simulation speedup can be achieved. Such a system behaves like a cache, where entries are aged and replaced to improve performance. Data caches improve performance by exploiting *locality of reference* to accelerate data access. Temporal locality occurs when an accessed data item is accessed again [10, p. 403]. It is exploited by keeping often accessed data in the cache. Spatial locality of reference occurs when data items near an accessed item are also accessed. It is exploited by placing data items adjacent to an accessed item into the cache. Data caches exploit these characteristics to hide storage device latencies.

Executive locality of reference refers to similar properties exhibited by a set of executing processes. *Executive temporal locality* refers to the property that an executing process will be executed again in the near future. It can be exploited by keeping the most active processes in hardware. Communicating processes may also exhibit *executive spatial locality*, which is the property that processes having a dependency on an executing process (e.g., receiving messages from it) will also be executed. Executive spatial locality can be exploited by keeping collaborating processes in the FPGA connected with direct communication channels. In the same way that a data cache replaces unreferenced items with referenced ones, a migration system should replace idle processes with busy ones. The FPGA, therefore, becomes an execution cache.

1.1 Overview

The thesis of this dissertation is as follows: When RTL simulations exhibit executive locality of reference, a process migration system can exploit it to gain a simulation speedup. The thesis is supported by the following chapters:

Chapter 2, "Background"

This chapter discusses technology and previous work related to the research presented herein. The functionality of reconfigurable hardware is described followed by a description of simulation methods including acceleration. An overview of existing simulation accelerators is presented. A migration system's architecture is described, and simulation speedup is discussed in light of Amdahl's Law with a special treatment of the impact of overheads.

Chapter 3, "Formal Modeling of Process Migration"

In an effort to understand the issues and difficulties of process migration, a formal model is developed. The definitions and theorems developed in this chapter establish a framework for reasoning about migration systems to ensure that an implementation guarantees correct and complete execution. Such guarantees are of special need in a migration system that can change during run time. The model defines a migration realm in terms of Finite State Machines (FSMs) and defines processes in an abstract form that can be translated into forms that are executable by concrete FSMs. In the context of a migration-capable simulator, the processes need to be in a portable format that can be translated to executable forms such as native-compiled code and hardware logic gates. The definitions also infer a migration realm taxonomy, and the theorems guide us in the development of algorithms to maximize the use of resources and to efficiently guarantee correct completion.

Chapter 4, "Processes and Temporal Locality"

In this chapter RTL processes are analyzed through profiling to detect executive locality of reference. Six designs from the OpenCores project were analyzed representing a range of code base sizes and functionality: an Ethernet MAC, an AC97 Audio Controller, a Serial Peripheral Interface (SPI), an Advanced Technology Attachment (ATA) Disk Controller, a Peripheral Component Interconnect (PCI) Bridge, and a memory controller. Transformations to RTL code, in light of the Verilog and VHDL standards, are developed that allow a migration system to exploit executive locality to its fullest. Processes are reduced to a *canonical form*, which may be combined as needed for optimization and load balancing within a simulator, while still maintaining correct behavior. These transformations are then applied to three of the OpenCores designs to demonstrate proper simulation behavior. Executive temporal locality

is then demonstrated through the use of *process guides* and *input monitors*. Input monitoring is the more efficient of the two methods, but process guides could be used for larger SoC simulations where entire Intellectual Property (IP) cores are used at certain times during the simulation. The input monitors produce a set of metrics that show the load of the processes on the system, and in virtually all cases a relatively few processes dominate the simulation time. These processes are the candidates for use in a simulator using process migration.

Chapter 5, "Implementation and Results"

In this chapter an implementation of a migration system is presented to demonstrate, analyze, and model a migration-capable RTL simulator. While Xilinx support for run-time reconfiguration is improving, it is still not trivial to accomplish. The partial reconfiguration flows that Xilinx provides allow the designer to produce a partial bitstream used to update the configuration of a portion of the FPGA, but a migration application requires the extraction and modification of per-processor state. Thus, independent of the partial reconfiguration flow, a method of extracting the location of per-processor state variables was developed. The resulting data are used to read and write configuration frames at run time, optimized by the use of frame caching. A system of 35 processors was created to measure the migration characteristics and parallel performance in a Virtex-II Pro FPGA. The implementation and the measurements obtained from it are used to empirically and algorithmically model a migration-capable RTL simulator to see the effect of various performance parameters on the system.

1.2 Contributions

The unique contributions of this work are the following:

- Definition and demonstration of executive locality of reference and exploiting it with process caching

  Similar terms of "spatial" and "temporal" can be found in the literature in reference to *task mapping* [11]. These terms, however, are applied to methods of task scheduling, rather than to locality of reference exhibited by the tasks themselves.

- Formal model of process migration and related taxonomy

The idea of process migration is not new, and its most familiar context is the migration of operating system processes within a cluster of workstations. Surprisingly, no efforts have been uncovered that address this technology in a formal way. The model contained in this dissertation is general enough to apply to operating-system-level migration as well as hardware/software migration, and it was presented at the Field Programmable Logic and Applications (FPL) conference in August 2007 [12].

- Addressing accelerated simulation at the RTL behavioral level

  Existing hardware accelerators in essence synthesize RTL code to gates and map it to FPGAs. The system presented here maintains the RTL level for execution as processes.

- The canonical form of RTL processes

  Several existing models of simulation are discussed, but these models approach the problem from a synthesis-semantic perspective rather than from an event-driven perspective. That is, they introduce components such as "latches" into their models which are only inferred from the semantics of RTL code, and they are not conceptually part of event-driven simulation. The canonical form of RTL maintains the semantics of event-driven processes, albeit in their most basic form.

- The successful implementation of a migration system using existing FPGAs and tools

  This implementation is based upon commercially available Virtex-II Pro devices utilizing the standard Xilinx design flow with some extensions to extract bitstream location information. This work was also presented at FPL 2007 as a short paper and poster [13].

- Empirical and algorithmic models to explore migration system performance

  These models are unique to the implementation described in Chapter 5, but the methods employed could be used for more general systems.

Chapter 2

Background

The previous chapter introduces the motivation and the basic concepts of a migration system that utilizes executive locality of reference to increase the efficiency of an RTL accelerator. The background and related work of this technology is presented in this chapter. In Section 2.1, an overview of Field Programmable Gate Array (FPGA) technology is given with a discussion of the types of reconfiguration an FPGA may support. Section 2.3 gives an overview of simulation technology, types of simulation, Hardware Description Language (HDL) abstraction levels, and acceleration methods. Section 2.4 presents the architectural overview of the migratory simulator, and Section 2.5 closes the chapter with an analysis of Amdahl's law and its application to simulation acceleration.

2.1 Reconfigurable Hardware

Reconfigurable hardware has grown in prominence in the last decade due primarily to two factors. First, the capacity of integrated circuits has increased significantly, as mentioned in Chapter 1. This growth has occurred not only in special-purpose integrated circuits but also in FPGAs, so FPGAs can hold significant designs despite the extra reconfiguration and routing logic present in them. Second, the cost of producing integrated circuits has increased significantly as the feature size has shrunk. This growth in cost has helped the adoption of

FPGAs because they are less expensive for lower volumes since the non-recurring development expense is amortized over a large market [14]. These factors have worked in concert to make FPGAs more attractive for designs than they were a decade ago.

FPGAs obtain their name from two of their characteristics. As a general rule, they are manufactured with no particular logic function. Rather, they provide an array of logic structures that can be configured to perform whatever function the designer wishes (within timing and area constraints). That is, they are *gate arrays*, *programmed* in the *field* after manufacture. There are several technologies used to configure an FPGA's function such as fuses, antifuses, and static random access memory (SRAM) [15, p. 479]. All these technologies utilize configuration bits to program switches for connecting logic, to configure clocking devices such as phase-locked loops (PLLs) and delay-locked loops (DLLs), and to program the basic logic functions of the device.

In fuse-based FPGAs, the configuration bits are programmed into the device by leaving fuses as-is or by "blowing" them (opening the circuit) with an applied voltage. Antifuse devices work in a similar fashion by permanently closing circuits rather than opening them. Devices configured in these ways are not reconfigurable; that is, they can be programmed once because the circuit is physically altered. In SRAM-based FPGAs, programming does not alter the physical structure of the circuit, so they can be reconfigured by changing their configuration SRAM's contents. A digital designer implements his design for the FPGA, producing a configuration file called a "bitstream." This bitstream is often stored in a non-volatile memory, which the FPGA reads into the configuration SRAM when powered.

A simple example that demonstrates how a logic function may be configured is the look-up table (LUT). Figure 2.1 shows a LUT having two inputs and one output. The LUT is essentially a memory, and the inputs form an address into the memory's contents to select the output. Depending on the contents of the memory, a LUT can function as an *AND* gate or an *OR* gate as shown by the two configurations. This LUT can, in fact, provide any logic function having two inputs and one output.

Normally, the configuration bits of an SRAM-based FPGA are programmed at power-up and then left unchanged while power is applied. In some applications, there may be reason to reconfigure it after power-up, but the device is stopped and completely reprogrammed during

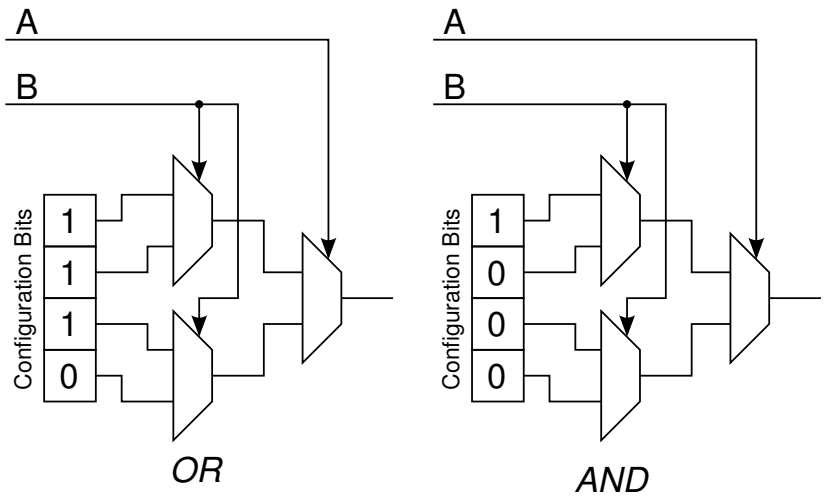


Figure 2.1. Basic LUT Architecture Showing *OR* and *AND*

reconfiguration—all state is lost. It is possible, however, to modify the configuration bits after the original power-on configuration while preserving state in another portion of the design [16, 17, 18]. This process is referred to as "active reconfiguration" or "dynamic reconfiguration."

There are two orthogonal aspects of run-time reconfiguration to be addressed. First is the *extent* of the reconfiguration, and second is the *mode* of reconfiguration. Concerning the extent of reconfiguration, it is categorized into *full* reconfiguration or *partial* reconfiguration. The normal operation of an FPGA begins with a full reconfiguration in which all the resources of the FPGA are configured with a bitstream. For the vast majority of designs, this is the only configuration the FPGA undergoes, but it is possible to subsequently reconfigure a select subset of the resources, a partial reconfiguration.

Concerning the second aspect of run-time reconfiguration, there are two modes, *active* and *non-active*. For default FPGA initialization, the device is configured, and the entire FPGA's state is reset. This is non-active (or static) reconfiguration. In contrast to this normal flow, active reconfiguration does not reset the FPGA's logic. In this mode, resources that are not being changed continue to run as though no reconfiguration were occurring. Because of this behavior, care must be taken to ensure that logic that is not being reconfigured does not depend on logic that is being reconfigured, and vice versa. This can be done, for example, with a global reconfiguration signal that gates inputs and outputs to the reconfigured logic. Combinations of

Table 2.1. Combinations of Reconfiguration

		Mode	
		Non-Active	***Active***
Extent	***Full***	Normal FPGA Configuration	Possible, but not likely
	Partial	Useful for quick modification of the FPGA functionality, but the entire FPGA is reset.	Used for dynamic modification of part of the FPGA with no change of state to the logic that is unmodified.

these orthogonal reconfiguration aspects are used to define the type of reconfiguration as shown in Table 2.1.

One FPGA vendor is Xilinx, Inc. Xilinx Virtex-II FPGAs support run-time reconfiguration to a limited degree, and Xilinx supports active, partial reconfiguration in their tools [19]. The product of the partial reconfiguration flow is a *partial bitstream* which is loaded into the FPGA through a configuration port. The configuration ports are Serial, SelectMAP, ICAP, and JTAG boundary scan [20], and they are to some degree mutually exclusive. These ports accept a common format desribed in the FPGAs' user guides, and a reconfiguration occurs when a file of this format is loaded into the FPGA.

The flexibility of FPGAs can be harnessed to provide customization of general processing systems. With a specific application in mind, designers can couple general-purpose processors with programmable logic arrays as demonstrated by systems such as PRISM [21] and Garp [22]. The programmable logic array can either provide bus-level devices to the processor to accelerate a computation, or it can extend the instruction set of the processor. Xilinx provides such an architecture in the form of Virtex™-II Pro and Virtex-4 devices. These devices contain PowerPC™ 405 processors surrounded by programmable logic [20, 23]. Both families can utilize bus level accelerators, but the Virtex-4 FX devices allow the PowerPC instruction set to be extended [24]. Altera, another FPGA manufacturer, formerly offered the Excalibur™ family whose devices contain Advanced RISC Machine (ARM) cores [25].

Another interesting application of FPGAs to reconfigurable processing is the Dynamic Instruction Set Computer (DISC) [26]. This architecture loads logic into a reconfigurable device to satisfy a program's execution as needed. As a program runs, any unsupported

instruction causes the loading of the module implementing the instruction using partial reconfiguration. The most recently used modules are kept in the FPGA forming an instruction module cache. The work presented in Chapter 5 is similar to DISC in that it forms a cache of the most-used functionality, but unlike DISC it implements a cache of processes rather than a cache of instruction modules.

PipeRench is a novel way of approaching computation with reconfigurable hardware [27]. Reconfiguration times of FPGAs are large (a hurdle faced in the implementation described in Chapter 5), but there are two traditional means in computing to hide such latencies: caching and pipelining. As its name suggests, PipeRench utilizes the latter to hide reconfiguration times. The PipeRench architecture consists of several reconfigurable pipelines. As an operation flows through PipeRench, it is reconfigured as needed, and so an operation can complete every cycle. The architecture is not suited for general computation, but performs well with stream-based algorithms. Simulation of RTL code is not a streaming application, so this approach cannot be used in this application.

Another use of run-time reconfigurable devices is the on-demand instantiation of functionality within it. The idea is to reconfigure a portion of the device to replace one functionality with another, and the available functionality could be retrieved with a Dynamic Module Server [28]. When using an FPGA as a migration system, optimizations could be made in area by instantiating only the minimal functionality required. For example, if a process does not use the multiply operation, then a processor without a multiplier should be instantiated to save FPGA area. Such selections could be made using a Dynamic Module Server, which improves performance by supplying a database of pre-compiled processors.

2.2 Design Abstraction

Figure 2.2 shows the various levels of abstraction available to a designer. At the lowest level is circuit design in which transistors, resistors, and other electrical components are arranged into circuits to implement the desired behavior. Working at this level limits the design to a relatively simple function because the number of components required to implement logic increases significantly with complexity. Constructing a 32-bit adder, for example, using

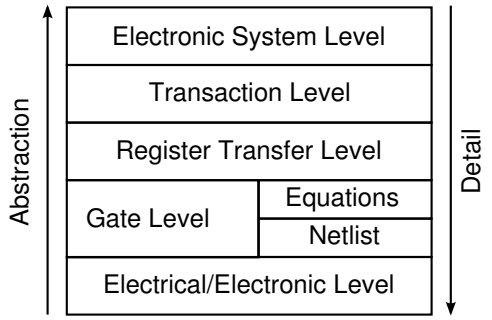
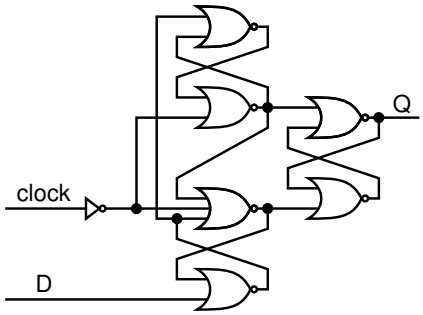


Figure 2.2. Levels of Abstraction for Electronic Design

transistors would be time consuming and difficult to debug. Above the circuit level is the logic gate level. At this level, the designer can work with boolean logic equations or with low level logic components to assemble a design. The components and logic functions may then be converted by tools to circuit descriptions. Implementing a 32-bit adder with boolean equations is significantly easier than doing so with transistors, but the complexity is still quite high because of the detail required to describe it. Further abstraction can be achieved by allowing the designer to express a 32-bit add with a simple plus sign ('+'), and *synthesis* tools then construct the boolean equations or target device components to implement the function. This is the level referred to as Register Transfer Level (RTL).

As mentioned in Chapter 1, the capacity of integrated circuits has doubled approximately every two years. Constructing a contemporary design at the electrical or gate levels is virtually impossible given the required detail. RTL has been, at least for the past decade, the abstraction level at which designs are described, but even the RTL level is proving to be too detailed for complex or System on Chip (SoC) designs. The next higher level of modeling in use today is the Transaction Level at which circuits' behavior is described using transactions, such as a bus or packet transfer. These models are not cycle accurate. The terms used to describe abstraction levels above (and to some degree including) the Transaction Level are still ambiguous, being in the formative stages [29], and so the term "Electronic System Level" (ESL) is used here.

As the need for greater levels of abstraction has been recognized, new languages have been developed to assist with the descriptions. In many cases, the desire to leverage exist-



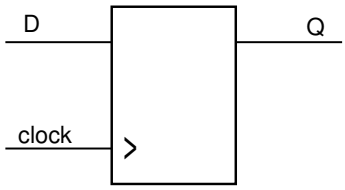

```
clock_not <= not clock;
net1      <= not(clock_not or net2 or net3);
net3      <= D nor net1;
q_bar     <= net1 nor q_gates;
Q         <= q_bar nor net2;
net2      <= clock_not nor net4;
net4      <= net3 nor net2;
```

Figure 2.3. Structural Representation of a D Flip-Flop (VHDL)

ing software code bases for use in hardware implementations has spurred the use of C-like languages for hardware design (often referred to as "C-to-gates"). SystemC utilizes a C++ class library to incorporate a simulation engine into compiled code, often allowing the reuse of existing C++ classes [30]. Handel-C, developed by Celoxica, provides extensions to ANSI C from which hardware can be generated [31].

In spite of the abstraction levels above RTL, the design flows of today still utilize RTL because the tools are not yet able to efficiently synthesize abstractions above RTL. This level represents the behavior of the circuit with high-level expressions describing the inputs and outputs of registers or flip-flops. RTL descriptions are often written in a hardware description language (HDL) such as Verilog [32] or VHDL [33], both IEEE standards, and synthesized into gate- or cell-level representations by tools such as Synplicity's Synplify® or the Synopsys® Design Compiler. A foundry can take the gate-level form and produce an integrated circuit, or the RTL can be used to create a configuration for an FPGA. Regardless of the target technology, however, RTL describes the parallel behavior of the circuit using "processes."

Figures 2.3 and 2.4 contrast the implementation of a D flip-flop in RTL and structural (or "gate-level") representations. A D flip-flop is a memory device that captures its `D` input at the rising edge of a `clock`, driving it at its `Q` output. There are also negative-edge triggered flip-flops that capture their input on the falling edge of the clock. If the input is different at the next rising edge of the clock, then the output will be updated to reflect this change. At the structural level, a logic circuit is described in the fundamental form of logic gates, and one implementation of a D flip-flop is shown in Figure 2.3. The RTL form shown in Figure 2.4 allows a designer to work at a higher level of abstraction resulting in much more concise and




```
d_flip_flop_rtl: process(clock)
begin
    if(clock'event and clock = '1') then
        Q <= D;
    end if;
end process;
```

Figure 2.4. RTL Representation of a D Flip-Flop (VHDL)

readable HDL code. In a synchronous digital circuit, the inputs to flip-flops are composed of "logic cones" which are sets of combinational logic that implement boolean or arithmetic functions. In a structural representation, the logic cones are described with primitive logic operations whereas at the RTL level, they are represented with high-level expressions such as addition and subtraction and with conditional constructs such as `case` statements and `if` statements. This RTL can then be compiled and simulated to verify correctness.

2.3 Digital Simulation

RTL simulation is an important development tool in digital design. In a multi-member team, the design can be subdivided with each member implementing a portion. Later, the sub-designs can be integrated to form the whole, but RTL simulation occurs at all stages. Some synthesis will likely be done to ensure the design fits into the target device and to ensure that the code structure supports the desired timings, but most of the early implementation phase is focused on RTL coding. During this early phase, it is desirable to have a quick turnaround for simulations along with good visibility into the design for debugging. Later, during the integration phase, the interfaces between the sub-designs will require debugging, which also requires a quick turnaround. After the design has matured, more esoteric bug fixes will require regression testing of the design to ensure that other errors are not introduced with the fixes. In this case, sheer simulation speed is beneficial.

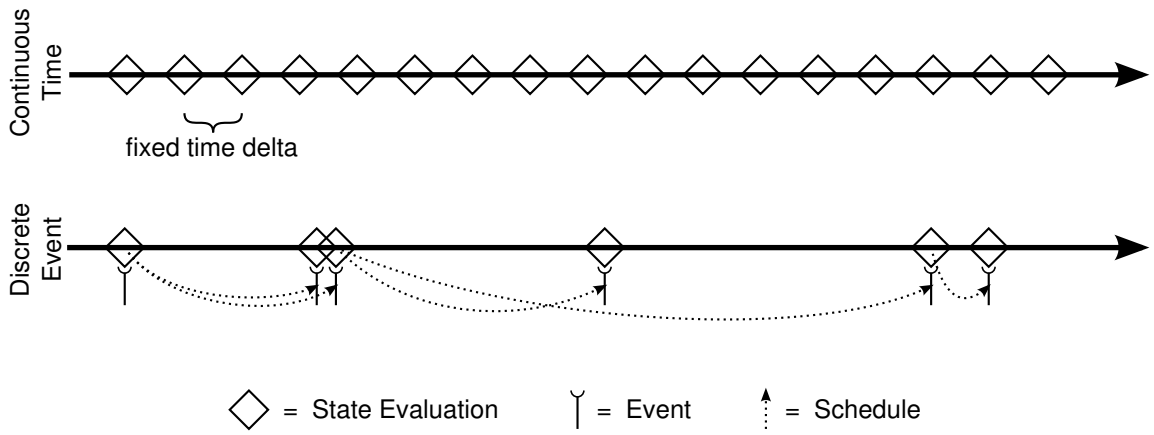


Figure 2.5. Two Simulation Types

2.3.1 Simulation Types

There are primarily two methods used to simulate systems: continuous time and discrete event [34], each depicted in Figure 2.5. In continuous-time simulations, the simulation time advances continuously at a fixed delta, and state within the simulator is a function of the time variable. Discrete-event simulations, on the other hand, skip simulation time when there are no changes in the state. Simulation of digital logic is performed with discrete-event simulation, where simulation *events* are placed, or *scheduled*, into the *event queue* which is sorted by execution time. Events include changes in the value of a signal or delays. In the case of Verilog, the IEEE defines five event queues into which different types of events are scheduled [32, p. 159]. The VHDL Language Reference Manual (LRM), the IEEE standard for VHDL, does not define execution in terms of a queue. In fact, the word "queue" does not appear within the 309 pages of the LRM [33]. It does, however, describe the execution in terms of sequences ordered by time, implying time-ordered queues like Verilog. The simulator's kernel services the event queue at the current simulation time, and it runs the processes which are activated by the events occurring at that time. If there are no events for the current simulation time, then the time is advanced to the next scheduled event, skipping the time between.

A more coarse-grained type of discrete-event simulation is cycle-based simulation. In the general discrete-event case, the transition of any signal at any time can trigger an event which evaluates state. For cycle-based simulations, the only event considered is the clock edge,

so it is only useful to simulate synchronous designs. A synchronous design is one composed of flip-flops sharing a clock with combinational logic between them. A D flip-flop captures its input at the rising edge of the clock, so its combinational input only needs to be evaluated at the clock edges. Any changes in logic between clock edges are irrelevant and do not need to be evaluated, so a simulation cycle simplifies from stratified queue management to an *alter-clock-retrieve* pattern [35]. This approach can improve simulator performance compared to a strict event-driven approach by reducing scheduling overheads and eliminating unnecessary process evaluations. A cycle simulator in use at IBM achieved a fifty-fold improvement over event-driven methods [36]. The SpeedSim simulator developed by SpeedSim, Inc. (later acquired by Quickturn Design Systems, Inc., which was in turn acquired by Cadence Design Systems) was claimed to improve run times by 10 to 100 times [37, 38]. Synopsys's RoadRunner cycle-based simulator was claimed to improve performance by 2 to 5 times [39], and their Scirocco™ and VCS® systems by 10 to 50 times [40]. These cycle-based simulators were absorbed into the acquirer's existing simulation systems, and cycle-based simulation techniques are applied "under the hood" [41, 42, 43]. The term "cycle-based" is no longer the marketing buzzword it once was, but cycle-based techniques are actively used in RTL simulators.

In discrete-event simulations, a "process" is a set of code that is executed to evaluate expressions and change state. It has a set of inputs and a set of outputs. When a signal value is changed in the simulation, the processes that are made sensitive to the signal must be run to evaluate their new output state. This reevaluation may trigger the immediate change of signals, causing more processes to be run, or it may schedule the change to occur later. When a signal is the input to multiple processes, it is possible to execute those processes in parallel. As shown later in Chapter 4, processes which can take advantage of cycle-based methods are easily identified.

2.3.2 Simulation Acceleration

There have been three primary methods employed to accelerate discrete-event simulations [44]: parallel execution in software, mixed-abstraction/multiresolution simulation, and hardware acceleration. These methods, of course, can be combined within a simulation system.

Parallel execution of processes introduces significant issues in keeping dependencies up to date across an asynchronous cluster while maintaining a respectable speedup. A good overview of the difficulties of parallel simulation, which primarily rest upon time management, can be found in [45]. Mixed-abstraction/multiresolution simulation is simply the use of varying modeling techniques so that simulation processes that are not crucial to the functionality under test are executed in the shortest time by using higher-level descriptions of them. An example is the combination of an instruction set simulator for an SoC core and a gate-level representation for a USB host device attached to the core. Simulating the entire SoC design at the gate level would be significantly slower.

Hardware acceleration of simulations uses dedicated hardware, and FPGAs have served that purpose well because of their reconfigurability. These accelerators are utilized by first mapping the RTL code to an array of FPGAs, thus offloading the work from a single processor to gain a speedup. This mapping is a time-consuming process requiring a non-negligible delay before the simulation can commence. Furthermore, resources are allocated at compile time with no possibility of adjustment during run time. This static mapping is acceptable if there is sufficient hardware to contain the entire design, but hardware accelerators are costly, ranging from hundreds of thousands to millions of dollars [44].

Commercial Offerings

Much progress has been made in the area of hardware acceleration with several commercial offerings in existence. Cadence Design Systems offers the Palladium® emulators [46] and the Xtreme® Server [47], both being parts of verification systems which can provide hardware acceleration of designs as well as emulation. Palladium was acquired with Quickturn Design Systems, Inc., and the Xtreme Server was acquired with Verisity, Inc. While information about their technology is sketchy, they do make it clear in their marketing "datasheets" that the HDL code is partitioned into software execution and hardware execution at compile time. State can be swapped from hardware to software for debugging, but there is apparently no run-time allocation of hardware resources occurring.

One publication by Quickturn [48], while not specifically mentioning Palladium, describes a system to accelerate event-driven simulations by providing low-overhead communi-

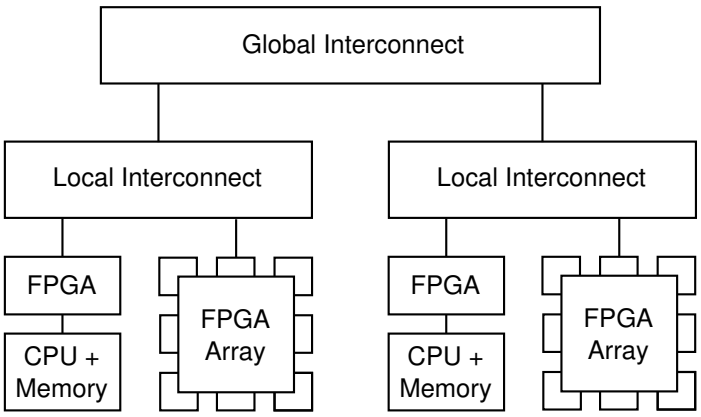


Figure 2.6. A Simulation Architecture Published by Quickturn

cation between the behavioral portions and the synthesized portions of the design. The system targets synthesizable HDL code to FPGAs, and then partitions the result across an array of FPGAs in an architecture depicted in Figure 2.6. The behavioral code is compiled for a local microprocessor run alongside an FPGA array. Additionally, the system synthesizes logic into the FPGAs to detect trigger conditions for the simulator's event loop, thus off-loading event detection from the software. The mapping to the hardware is still static.

EVE Corporation produces simulation accelerators in their ZeBu ("zero bugs") product line [49]. EVE differentiates itself somewhat from its competitors by offering smaller emulation systems such as the ZeBu-UF that plugs into a PCI slot of a personal computer, but their flow is similar, requiring a static synthesis of the design before simulation.

Berkeley Emulation Engine

These aforementioned systems are proprietary, and the information about their implementations is sketchy at best. In academia, the Berkeley Emulation Engine (BEE) provides an open platform that could be used for accelerated hardware verification [50]. The engine is composed of an array of 20 Virtex 2000's on each Main Processing Board, providing a theoretical 8 million gate emulation capacity. These FPGAs are connected in a two layer mesh structure (shown in Figure 2.7). Designs are developed as a high level model in Simulink and compiled into a VHDL netlist and mapped to the FPGA hardware. While the design flow of the BEE system is geared toward communication applications, it is a general system that could be

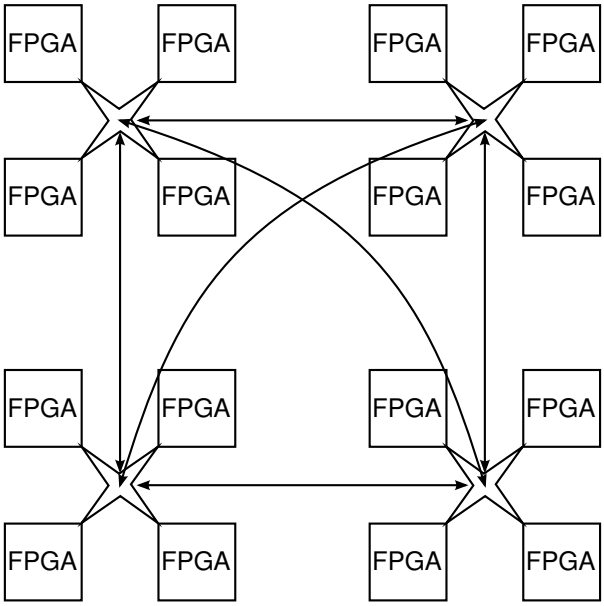


Figure 2.7. The Berkeley Emulation Engine Architecture

used for simulation acceleration. The second generation BEE2 is built upon the lessons learned in BEE, and provides improvements on it such as embedded PowerPC processors of Virtex-II Pro FPGAs and high-bandwidth DDR memory [9]. Upon this BEE2 platform, the Research Accelerator for Multiple Processors (RAMP) is envisioned to be built [51]. RAMP seeks to establish an open, community-supported platform for multiprocessor systems research, so that software and hardware can be rapidly prototyped. While it can be used for simulation acceleration, its designers intend it to be used for research at higher levels of abstraction. In their own words, ". . . RAMP targets architecture and software research instead of simulation of a single RTL design" [51, p. 20]. Furthermore, "The proposed RAMP 1 and RAMP 2 hardware would be comparable in capacity to the most capable commercial [acceleration] platforms. However, a fundamental difference is RAMP's emphasis on emulating high-level architectural behaviors rather than cycle-accurate RTL netlists" [51, p. 20].

SimPLE

Another approach to hardware acceleration is to assemble an array of simple processing elements in an FPGA, and to schedule very long instruction words (VLIWs) to execute the simulation as in the SimPLE system [52]. Each processing element (PE) is a two-input look-up

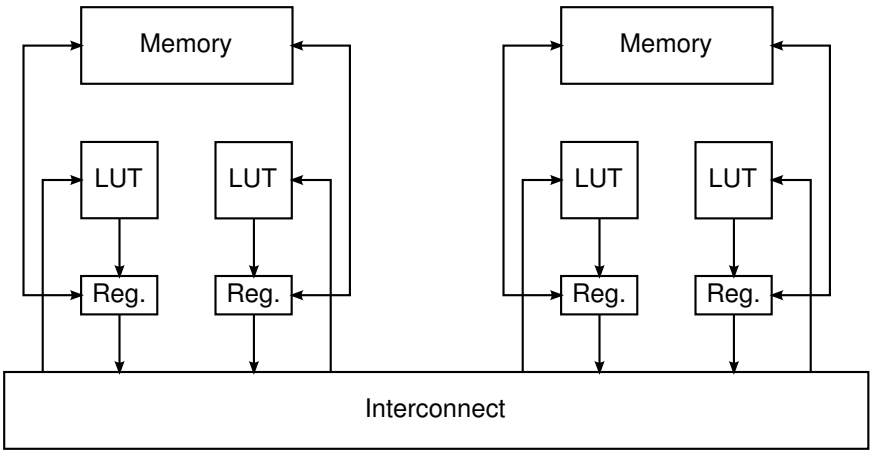


Figure 2.8. The SimPLE Processing Element Architecture

table with a two-level memory system. The first level memory is a register file dedicated to each PE with a second-level spill-over memory shared among a group of PEs, as depicted in Figure 2.8. The PEs are compiled into an FPGA which is controlled by a personal computer host through a PCI bus. The host schedules very long instruction words (VLIWs) to the FPGA, and each instruction word contains opcodes and addresses for all the PEs. Each opcode specifies the function of a two-input look-up table, and so consists of four bits. The register file consists of four bits, one of which is selected by the instruction for placement on the interconnection system. So on every cycle of the PE array, a single logic gate of the simulation is evaluated on each PE. The design flow is to synthesize the HDL design into a Verilog netlist, translate the netlist into VLIW instructions using their SimPLE compiler, and schedule them into the FPGA to execute the simulation. An EDA startup company, Liga Systems, Inc., has been founded to take advantage of this technology [53].

The system behaves very much like the FPGA upon which it is constructed: The crossbar switch is analogous to an FPGA's interconnection system, the PE is analogous to the look-up tables of a configurable logic block (CLB), the registers are analogous to the flip-flops within each CLB, and the VLIW is analogous to the configuration bitstream of the device. There is no reconfiguration of the FPGA's logic functions; rather the reconfiguration occurs at the layer implemented above it. As such, this system can be implemented in a fixed-logic system such as an ASIC. The SimPLE architecture is nothing more than a reconfigurable

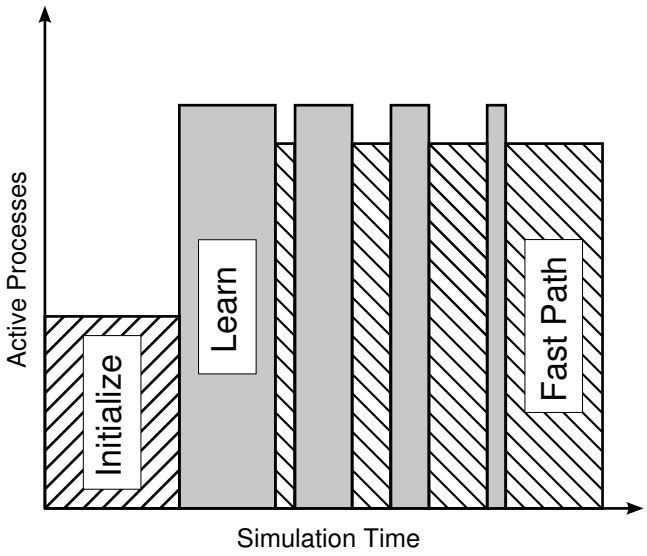


Figure 2.9. Example Simulation Phases

logic array with faster reconfigurability than traditional FPGAs, but that alone makes the system attractive for other run-time reconfigurable applications. As with the other systems, the drawback is the required synthesis to the gate level before a simulation can commence.

These systems all suffer from a common shortcoming, albeit to varying degrees. The design must be compiled to a gate-level netlist or to FPGA structures before it can be simulated. In some cases, the mapping is to a higher abstraction level, thus shortening the synthesis time, but some prior mapping is required. Furthermore, all the systems except the SimPLE compiler map to the hardware resources statically. That is, once the mapping to hardware is established, the logic remains there throughout the life of the simulation. If the hardware resources are significant, then this is an acceptable solution, but if a limited amount of hardware needs to be used efficiently, this can be problematic.

Take, for example, a simulation of a networking ASIC. The simulation goes through three phases as shown in Figure 2.9: initialization, learning, and "fast path." The learning and fast-path phases overlap to some degree as the network routes of the test traffic are learned. Those routes that have already been learned execute the fast path logic, and those that have not are still executing the learning logic. Eventually, the network routes are all learned, and the simulation is exclusively fast path. In a scenario such as this, a static mapping to limited

hardware resources can be inefficient. For example, if the compiler mapped the initialization processes to the hardware, then the latter two-thirds of the simulation would not benefit from the hardware acceleration. On the other hand, if a portion of the processes for each phase is mapped to the hardware, then all phases of the simulation would benefit, but at any given time approximately two-thirds of the hardware resources would be idle. Thus, a static mapping is not the most efficient use of the hardware.

Cost is also an issue. The commercial hardware accelerators are expensive, ranging from hundreds of thousands of dollars to millions depending on the configuration [44]. Hence, a lower-cost implementation that can be expanded as needs require and resources allow is desirable. The goal is to simulate at the RTL level without a *prior* mapping to hardware resources and without a *static* mapping to hardware resources. These are the primary guiding principles, but a secondary motivation is to develop a system that has a lower cost and finer grained incremental capacity than existing accelerators.

2.4 Architectural Overview

Since the desired goal is to simulate at the RTL level without a *prior* or *static* mapping to hardware resources, the prior mapping to hardware is first be addressed. This step can be avoided if the playing field is leveled, so to speak, between software and hardware. This can be done with a common instruction set (CIS) that can be executed in a software array of Virtual Machines (VMs) or in a hardware array of parallel Real Machines (RMs). Since a compiled process can be run in either location, a prior mapping is not necessary.

Software is historically more flexible than hardware; in traditional computing systems, a single change in a software program requires a recompilation of the source. A single, incremental change in hardware, however, typically requires the mapping, placement, and routing to be repeated for FPGAs. More drastic measures are required for an ASIC, such as metal-mask changes or a complete re-spin. The use of run-time reconfigurable devices aids in this regard, so that minor modifications of the logic of a device can be achieved more quickly. This run-time reconfiguration creates a hardware flexibility similar to the flexibility of software and avoids a static mapping to the hardware resources.

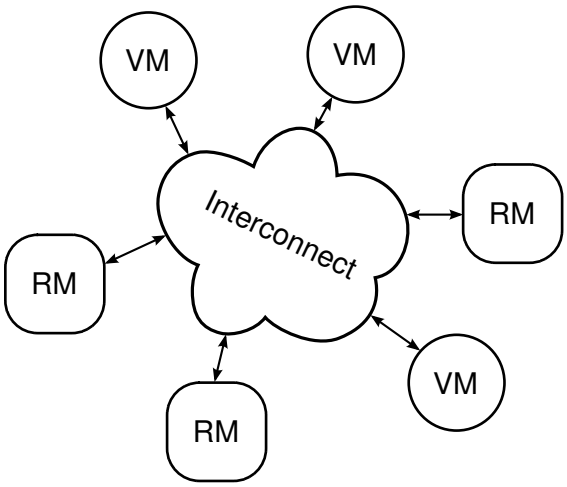


Figure 2.10. Migration System Logical View

Ideally, every process of an RTL simulation would be run on its own processor in parallel, but since a simulation can be composed of thousands of processes, the size of a single reconfigurable device may be prohibitive. The limited parallel hardware resources must be used in efficient ways in order to maximize the simulation speedup. Using a common instruction set in VMs and RMs also allows the migration of state between them to be straightforward.

Figure 2.10 shows the logical architecture of the system. A simulation begins with all of the processes executing in VMs, and processes are selected to migrate into the RMs to gain a simulation speedup. If hardware resources are not sufficient to contain all the processes, then the remaining active processes can be just-in-time compiled to native processor code. While the simulation progresses, execution characteristics of the machines are monitored, and idle processes are migrated out of RMs and replaced with busy ones. There may also be processes that could benefit from just-in-time compilation directly into FPGA resources.

2.5 Simulation Speedup

Hardware-based simulation accelerators obtain their speedups using parallel execution. The fastest possible execution of an RTL description is when it is implemented in a target device such as an ASIC or an FPGA. The functionality is fully parallel. On the other side of the spectrum is typical, low-cost RTL execution on a single processor, a fully serial execution.

The goal of a simulation accelerator is a balance between fully serial and fully parallel that minimizes run time, compile time, and cost. The cost pressure tends to minimize the amount of hardware, while the run time pressure tends to maximize the efficiency of whatever hardware is available. The system, therefore, may be partially parallel, and the speedup that can be obtained is governed by Amdahl's Law.

Amdahl's Law states that the speedup is

$$S = \frac{s+p}{s+p/N}, \tag{2.1}$$

where s is the fraction of the uniprocessor run time that is run serially, p is the fraction of the uniprocessor run time that is run in parallel, and N is the number of parallel processors [54]. In the migratory simulation system, N is limited to the number of available RMs. If one assumes, for the sake of qualitative analysis, that all processes require an equal amount of time to run, then the time units can be normalized to a single process's run time. Let P be the number of processes in the application. The fraction of uniprocessor run time becomes

$$s = \frac{P-N}{P} = 1 - \frac{N}{P}, \tag{2.2}$$

when a single process is run on each RM. Using Equation 2.2, the parallelized fraction of the run time becomes

$$p = 1 - s = \frac{N}{P}. \tag{2.3}$$

Equation 2.1 simplifies as follows:

$$\begin{aligned} S &= \frac{\left(\frac{P-N}{P} + \frac{N}{P}\right)}{\left(\frac{P-N}{P} + \frac{N}{P} \cdot \frac{1}{N}\right)} \\ &= \frac{P-N+N}{(P-N)+1} \\ &= \frac{P}{(P-N)+1} \end{aligned} \tag{2.4}$$

Typical plots that demonstrate Amdahl's Law increase monotonically while asymptotically approaching a theoretical maximum speedup as N increases. However, if p is a function of N (as shown in Equation 2.3), then the plot takes a different form which is a cross-section of several plots of Equation 2.1. This behavior is shown in Figure 2.11, and characterizes the behavior of the implementation described in Chapter 5.

Considering Equation 2.4, one can see that as the number of processors N approaches the number of processes P, the speedup approaches P. A simulation, however, can easily consist of thousands of processes, and the overheads, which will always be serial, cannot be ignored. So, the speedup relies on the number of RMs and on the overheads. The overheads are a function of the number of processes, and it is modeled with different values for VM processes and RM processes.

Let V be the number of processes running in VMs and R be the number of processes running in RMs. Note that $V = P - N$, and $R = N$. Equation 2.4, then, more realistically becomes

$$\begin{aligned} S &= \frac{PO_v + P}{VO_v + V + RO_r + 1} \\ &= \frac{P(O_v + 1)}{V(O_v + 1) + R(O_r + \frac{1}{R})}, \end{aligned} \tag{2.5}$$

where O_v is the VM overhead and O_r is the RM overhead, both as fractions of a process time.

There are two conclusions one can can draw from equation Equation 2.5: First, as Amdahl's Law predicts, it requires a significant percentage of processes to be run in parallel in order to get even a speedup of 2. Second, a large RM overhead (O_r) can cause a speedup of less than one, but that overhead must be greater than 100% of the process execution time. To illustrate these concepts, Figure 2.12 shows three plots of Equation 2.5 with an assumed VM overhead of 10%. The vertical axis is the speedup. The horizontal axis is the RM overhead as a fraction of the process execution time. The three curves show differing mixes of VMs and RMs for 100 processes.

John Gustafson has reevaluated Amdahl's Law [54] in light of the his experiences at Sandia National Laboratories, and points out that it only applies to problems of fixed size. He contends that better speedups are achievable than Amdahl predicts because, "in practice, *the*

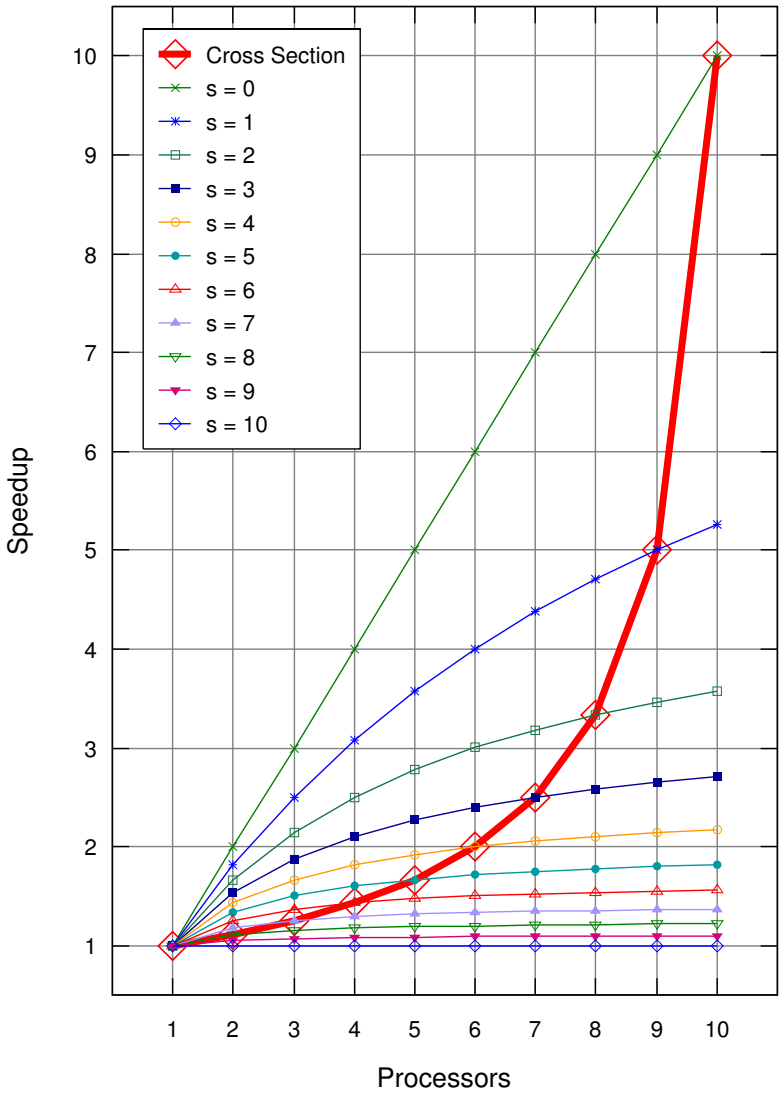


Figure 2.11. Cross Sectional Speedup: Speedup takes a cross section of typical Amdahl's Law plots when the amount that is parallelizable varies with the number of processors. The typical Amdahl's Law plots are $S = \frac{P}{s+(p/x)}$, where $P = s + p$. The cross section is given by $S = \frac{P}{P-N+1}$.

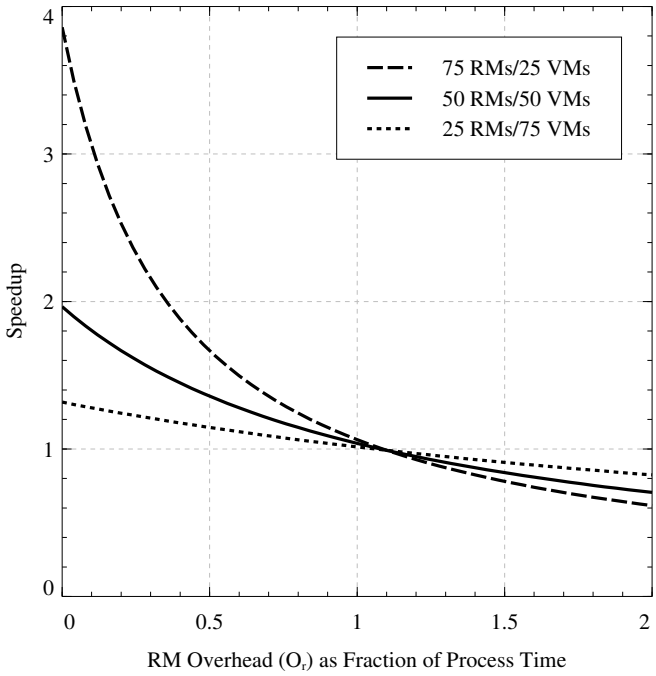


Figure 2.12. Hypothetical Speedups

problem size scales with the number of processors" (emphasis his). Within the framework of this simulation application, Gustafson is saying that the number of simulation processes must increase with the number of RMs. Such an increase is not possible, however, since the problem size is indeed fixed in a given simulation. Therefore, a parallel simulation system is limited by Amdahl's Law.

Chapter 3

Formal Modeling of Process Migration

The goal of this chapter is to present a formal model that describes in general the migration of processes among processors. It defines the concepts of a *program* and a *process*. A process is migrated within a *migration realm*, which is a collection of computing devices modeled in terms of Finite State Machines (FSMs). The instructions of a process are divided into sequences that are serially executed on one or more processors within the realm until the process completes. A process exists in an abstract form that can be translated as needed into a form that a concrete processor can execute. This model also forms the basis for a migration system taxonomy, and its theorems present a set of criteria which guarantees the correct completion of programs in a set of heterogeneous processors.

While the model is generally applicable to many forms of migration systems, its impetus is the migration of processes between software and FPGA hardware. Run-time reconfiguration of FPGAs provides the means of implementing a migration realm within an FPGA, and one application is the acceleration of digital circuit simulation. FSMs were chosen as the basis of the model primarily because digital circuits are often described in the form of FSMs, so the migration of FSM state is natural in this context. However, the model defines its components in general enough terms to be applicable to Turing Machines as well.

The next section presents related work and explains how it is different from the content of this chapter. Section 3.2 establishes the foundational definitions of strings and FSMs referred

to in subsequent sections. Section 3.3 describes a migration realm, a program, and a process. Section 3.4 elaborates on how to simplify migration realms by requiring certain similarities between FSMs. Section 3.5 describes an application to run-time reconfiguration of FPGAs in light of this model, followed by Section 3.6 which gives an overview of future work. Section 3.7 concludes.

3.1 Related Work

Brand and Zafiropolu, in the foundational work on communicating finite state machines, present a model of multiple machines communicating through a *protocol* [55]. The object of this model is to detect missing or "unexecutable" messages within the protocol. The modeling of digital simulation with FSMs has been done by Pétrot *et al.* [56], but that model focuses on the evaluation dependence between the machines and is in essence a communicating FSM problem rather than a state migration problem. Similarly, R. Milner *et al.* [57] have developed a calculus of mobile processes (π-calculus), which describes a set of communicating mobile processes that can change their communication link structure. This chapter describes the migration of the state of a single machine without consideration of communication or collaboration with other machines. Communication between collaborating FSMs is orthogonal to the migration of each machine's state.

Work has also been done with MobileUNITY regarding the movement of mobile programs through *ad-hoc* networks [58], but that work focuses on sharing and synchronization in mobile computing systems rather than the migration of processes themselves. More closely related to this dissertation work is the MAIL [59] language which formalizes itineraries of a process through a network, but it does not address translation of code and state in migration systems. It also focuses upon autonomous mobile agents rather than passive processes, the object of this work.

The migration of operating system processes has been studied extensively [60]. This type of migration is an application at a higher level than that addressed here, with a greater focus on implementation rather than modeling, but it does share some points with the content of this chapter with respect to heterogeneity. At the operating system level, heterogeneity is a

difference in the machines' operating system, instruction sets, and registers sets. The model presented here, however, addresses execution within instruction subsets as well as execution in completely different instruction sets, and there is no need at this level to consider operating system concepts such as file I/O and memory allocation.

3.2 Foundational Definitions

The execution units are modeled as finite state machines, also known as finite automata, and the concepts of programs, processes, the moving of state, and the translation of input are built upon them. Each program is modeled as a string of input symbols to an FSM for the purpose of establishing terminology and notation, and there must be a means to represent which symbols have been processed and which have not. To reach that goal, some foundational definitions are introduced first.

Definition 3.1. Let A be a finite set. A *string* with elements (symbols) from A is a finite sequence of elements from A juxtaposed. The length of a string σ, denoted $|\sigma|$, is the number of elements in the sequence. The empty string ε is the string of length zero. The set of all strings over A (including the empty string) is denoted A^*. □

The elements of a string are delimited using a positional, zero-based subscript when the context requires. The *concatenation* of two strings is the juxtaposition of each sequence of symbols from each string. Any string concatenated with the empty string is the original string: $\varepsilon\sigma = \sigma$. Thus the empty string acts as the identity under concatenation. One can express some simple properties of a string σ over set A:

$$\begin{gathered}\sigma = \sigma_0\sigma_1\sigma_2\ldots, \\ \text{for } 0 \leq i < |\sigma|, \sigma_i \in A, \\ \text{and } \sigma \in A^*.\end{gathered}$$

Definition 3.2. Let $\sigma = \sigma_0 \ldots \sigma_m$ be a string over a finite set A. A string $\sigma' = \sigma'_0 \ldots \sigma'_k$ is a *substring* of σ if there exists $n \geq 0$ such that $\sigma'_i = \sigma_{i+n}$ for $0 \leq i \leq k$. The substring is denoted $\sigma_{[i,i+n]}$. □

Example 3.1. Suppose that $\sigma = abcdefg$. The following conclusions can be made:

$$\begin{aligned} |\sigma| &= 7 \\ \sigma_{[1,3]} &= bcd \\ \left|\sigma_{[1,3]}\right| &= 3 \\ \sigma_{[0,|\sigma|-1]} &= \sigma \\ \sigma_{[6,6]} &= \sigma_6 = g \end{aligned}$$

A Finite State Machine (FSM) provides the basis upon which strings and substrings will be applied to establish the notion of a program and a process. An FSM is a machine that processes input symbols, changing its state for each symbol, and it can be classified as either an acceptor or a transducer, depending on what the output of the FSM is considered to be [61]. An acceptor has an output of `true` if the final state after all transitions is an element of the set of acceptor states, and `false` otherwise. For a transducer, there is an output function that maps the state of the FSM into an output alphabet. FSMs are often described with directed graphs whose vertices are the states, and whose arcs are the state transitions labeled with the symbol causing the transition.

Example 3.2. Computation can be represented with an FSM, but the number of states becomes quite large. Consider a machine with three registers of only four bits each. The state space is twelve bits with 4,096 states. To limit the examples to a reasonable number of states, simple acceptors are used in this chapter, but the principles apply for computational machines with much larger state spaces. Figure 3.1 shows a simple acceptor that processes a series of input symbols and determines if the input symbols spell "computer." Because of this property, acceptors may be used for language recognition. After all the input symbols are applied to the FSM, if the state is `true`, then the input was "computer;" otherwise, it was not. Its structure also differentiates between "computer" and "computers."

Migration deals with the transition of states between FSMs rather than the output of those FSMs, and the principles are applicable to both types. The initial states become a property of "programs" rather than the machines themselves. Therefore, for brevity, semiautomata are used, which are reduced FSMs that omit initial states and anything output related such as

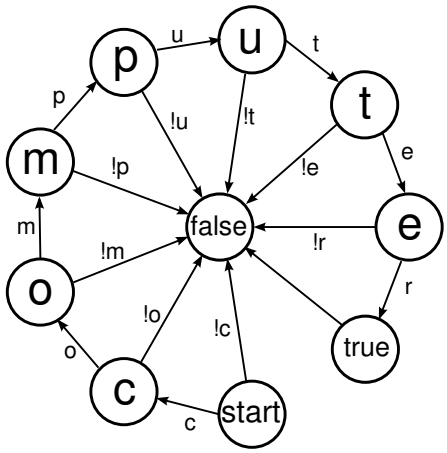


Figure 3.1. An Acceptor FSM Recognizing "computer"

acceptor states and output functions. For the remainder of this chapter, the term FSM is used as a synonym for a reduced FSM, unless stated otherwise.

Definition 3.3. A *reduced finite state machine* is a triple

$$\mathcal{A} = (S, A, \delta)$$

where S is a finite set, and $\delta\colon S \times A^* \to S$ is a function such that for all $s \in S$, $a \in A$, and $\sigma \in A^*$,

$$\begin{aligned} \delta(s, \varepsilon) &= s \quad \text{and} \\ \delta(s, \sigma a) &= \delta(\delta(s, \sigma), a)\ . \end{aligned}$$

The set S is the set of *states*, A is the *input alphabet*, and the elements of σ and symbol a are the *input symbols*. The function δ is the *transition function* of the FSM. □

For the remainder of this chapter, when an FSM $\mathcal{A}$ is referenced, S, A, and δ are assumed to be the elements of the FSM triple, though not explicitly mentioned. Similarly, when a specific FSM $\mathcal{A}_i$ is mentioned in a definition, then the context implies that S_i, A_i, and δ_i belong to that FSM, though not explicitly specified.

3.3 Migration Realms and Processes

The migration of a process requires a source processor and a destination processor, so any group of processors can provide a "realm" within which a process may migrate. Intuitively, if there is a set of processors that are identical in every respect, the migration of state is direct. If there is a need to migrate a process between two processors that use different instruction sets or have different numbers of registers, such differences are shown in this model using unequal alphabets, states, and transition functions.

Definition 3.4. A *migration realm* $\mathcal{R}$ is a set of FSMs

$$\mathcal{R} = \{\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots\}.$$

$|\mathcal{R}|$ is the number of FSMs in the realm. A *homogeneous migration realm* $\mathcal{R}$ is a migration realm such that for all $\mathcal{A} \in \mathcal{R}$

$$S = S'$$
$$A = A'$$
$$\delta = \delta',$$

for arbitrary S', A', and δ'. Otherwise, the migration realm is *heterogeneous*. □

The migration of a process has already been discussed in an informal manner, but in order to satisfy the rigor of the model, a program and a process must be formally defined before migration is defined.

Definition 3.5. A *program* $P = (\mathcal{A}, s_0, \sigma)$ is a triple consisting of a finite state machine $\mathcal{A} = (S, A, \delta)$, an initial state $s_0 \in S$, and a string of input symbols σ such that $A = \{\sigma_0, ..., \sigma_{|\sigma|-1}\}$. A *process* p of P is a quadruple $p = (\mathcal{A}, s, \sigma, c)$ where

$$s = \begin{cases} s_0, \text{ for } c = 0 \\ \delta(s_0, \sigma_{[0,c-1]}), \text{ for } 0 < c \leq |\sigma|. \end{cases}$$

s is the *current state*, and c is the *input counter*. The process is *complete* when $c = |\sigma|$. □

Within a migration realm, each machine is not required to be able to process an entire program, and so an abstract FSM is incorporated into the program to serve as a reference. A program's abstract FSM does not necessarily have a complete implementation in the realm, but the model uses it to establish relationships between it and the concrete machines in later definitions. This abstract machine is guaranteed by Definition 3.5 to be able to execute the entire program. In later definitions, the superscribed hat notation (e.g., $\hat{\mathcal{A}}$) is used to make the difference between the elements of the abstract machines and the concrete machines more readily identifiable.

The program itself is also abstract. It is in a form which has not yet begun execution, and it can be instantiated as multiple processes when it is run. Its definition applies intuitively to computer programming: Source code is compiled into a program for a target processor, and it consists of sections such as "text" and "data." The text is the instructions to be executed by the processor, and it is analogous to a string of symbols σ input to an FSM. The data section is the initial state of the program, analogous to s_0. An operating system can load the text and data into memory—multiple times, if desired—and begin executing it as a process. The target processor for which the source code is compiled is analogous to the abstract FSM.

A process is an instance of a program, and the input counter tells how far into the program the process has executed to arrive at the current state. Computer programs often contain loops where the program counter is set to a previous value. FSMs can model loops by unrolling them completely in the input string, and the program counter increases monotonically. A process is still abstract because it exists in terms of the abstract FSM of the program from which it was instantiated. That is, it is not necessarily in a form that can be executed on a concrete FSM, so the superscribed hat notation is employed to identify it as such when the context requires. The flow of a process from its abstract form to concrete forms is demonstrated in Figure 3.2. Through "binding," shown as $B(p)$, the process is translated into a form that a concrete FSM can process. When the FSM has processed its portion of the program, it is unbound ($U(p)$), or translated back into terms of the abstract FSM. So a *process* is abstract, and a *bound process* is concrete.

A binding function is the means of producing a bound process from a process, and there are two binding functions, one to bind the state and the other to bind the input string.

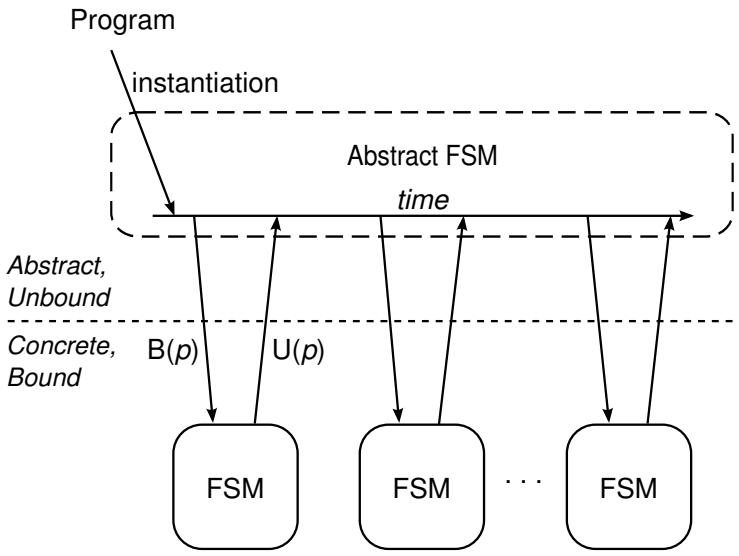


Figure 3.2. Processes Bound and Unbound Between Concrete and Abstract Machines

The binding of the state is straightforward. If there exists a binding function $B_s : \hat{S} \rightarrow S$ that injectively maps the current state of the process into the state space of the destination machine, then the state can be bound. Determining if the input string can be bound is a more difficult problem, since the target FSM is not required to process all of the remaining input.

Definition 3.6. Binding, Unbinding, and Migration

a) Let σ be a string. σ is a *bindable string* to $\mathcal{A}$ if there exists an injective function $B_A : \{\sigma\} \rightarrow A^*$ such that $B_A(\sigma) \neq \varepsilon$. The image $B_A(\sigma)$ is the *bound string*.

b) Let $\hat{p} = (\hat{\mathcal{A}}, \hat{s}, \hat{\sigma}, \hat{c})$ be a process of program $\hat{P} = (\hat{\mathcal{A}}, \hat{s_0}, \hat{\sigma})$. The six-tuple

$$p = (\hat{\mathcal{A}}, \hat{\sigma}, \hat{k}, \mathcal{A}, s, \sigma) \tag{3.1}$$

is a *bound process* of $\hat{p}$ to $\mathcal{A}$ if there exists a bindable string $\hat{\tau} = \hat{\sigma}_{[\hat{c},\hat{k}-1]}$ such that $\sigma = B_A(\hat{\tau})$, and there exists a function $B_S : \{\hat{s}\} \rightarrow S$ such that $s = B_S(\hat{s})$. $\hat{k}$ is the input counter marking the end of the bindable string. The relationship between $\hat{p}$ and p is denoted $p = B(\hat{p})$. If B_S and σ exist, then both $\hat{p}$ and $\hat{P}$ are *bindable* to $\mathcal{A}$. Process $\hat{p}'$ is the *unbound process* of p if $\hat{p}' = (\hat{\mathcal{A}}, \hat{s}', \hat{\sigma}, \hat{k})$ such that $\hat{s}' = U_S(\delta(s, \sigma)) = \hat{\delta}(\hat{s}, \hat{\tau})$. The relationship between $\hat{p}'$ and p is denoted $\hat{p}' = U(p)$.

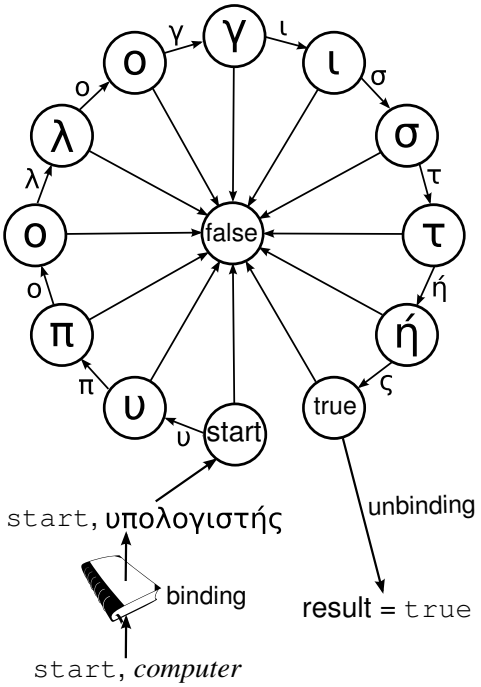


Figure 3.3. Binding to and Unbinding from a Greek Acceptor FSM

c) Function $M : \{p\} \rightarrow \{p\}$ is a *migration function* such that $p' = M(p) = B(U(p))$. When the context requires, M_{ij} is the migration function from a process bound to $\mathcal{A}_i$ to a process bound to $\mathcal{A}_j$. □

For the sake of brevity, further references to a process $\hat{p}$ or a program $\hat{P}$ infer the constituents $\hat{\mathcal{A}}$, $\hat{s}_0$, $\hat{s}$, $\hat{\sigma}$, and $\hat{c}$.

Definition 3.6 formalizes the migration mechanism. In order to migrate a process from one state machine to another, there must be a function to map state and a function to map input symbols into a form usable by the destination concrete machine.

Example 3.3. The program in Figure 3.1 would be defined as $\hat{P} = (\hat{\mathcal{A}}, \texttt{start}, \textit{computer})$, where $\hat{\mathcal{A}}$ contains the states and transition function shown with an input alphabet of the twenty-six English letters. Suppose that a migration realm consists of the single FSM shown in Figure 3.3 with an input alphabet of the twenty-four Greek letters (ignoring diacritical marks). The state binding function B_S maps from the `start` of the English FSM to the `start` of the Greek. The unbinding function U_S maps `true` from the Greek FSM to `true` of the English FSM, and it maps all other Greek FSM states to `false` of the English FSM. The input string

binding function B_A is an English-to-Greek dictionary. Thus, when the input string *computer* is translated to υπολογιστής, the concrete FSM in Figure 3.3 processes it, and the final unbound state is `true`. Through this sequence, the program is executed in a completely different FSM from which it was defined, but the result is the same.

The definition of an unbound process also addresses the issue of correctness. A process may be bindable to a concrete FSM, but that does not guarantee that the final state will be mappable back into the abstract FSM's states, nor does it guarantee, if it is mappable, that the state is correct. The requirement in unbinding that $U_S(\delta(s,\sigma)) = \hat{\delta}(\hat{s},\hat{\tau})$ ensures that the result is correct. That expression essentially states that if the final state of the bound process—when mapped back into the abstract state space—is the same as the state that the abstract FSM produces, then the final state is correct. If they are not equal, then the unbound process does not exist, and the process is *stranded*. A process can also be stranded in an unbound process form if there are no concrete FSMs to which it can be bound. It is a waste of resources in a real implementation to strand processes, so one must ensure that the process can be executed to completion within a realm.

Definition 3.7. Let $\hat{p}$ be a process of program $\hat{P}$. $\hat{P}$ is *realm executable* in $\mathcal{R}$ if there exists a finite sequence of bound processes $\mathbb{P} = (p_1, p_2, \ldots, p_n)$ such that

$$p_{i+1} = \begin{cases} B(\hat{p}), \text{ for } i = 0, \\ M(p_i), \text{ for } 0 < i < n, \end{cases}$$

and there exists $\hat{p}'$ such that $\hat{p}' = U(p_n)$. The sequence of ordered pairs of FSM and bindable string from $\mathbb{P}$ is the *execution path*. The sequence of FSMs from $\mathbb{P}$ is the *FSM path*. □

Theorem 3.1. *Given a homogeneous migration realm $\mathcal{R} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$, a program $\hat{P}$ that is realm executable in $\mathcal{R}$ is also realm executable in any non-empty subset of $\mathcal{R}$.*

Proof. Let $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ be an arbitrary FSM path of $\hat{P}$. Let $\mathcal{R}' = \{\mathcal{A}_i\}$. Because $\mathcal{R}$ is homogeneous, for all $\mathcal{A} \in \mathcal{R}, \mathcal{A} = \mathcal{A}_i$, so $\mathbb{A} = (\mathcal{A}_i, \ldots, \mathcal{A}_i)$. Therefore, $\hat{P}$ is realm executable in $\mathcal{R}'$. □

3.4 Reducing Heterogeneity

Definition 3.4 defines a heterogeneous migration realm as a realm in which at least one of the three elements of the FSM triple differs between automata. Definition 3.6 and Definition 3.7 formalize the requirements that must be satisfied in order for a program to be realm executable. These are general requirements for all realms, and the correctness requirement of Definition 3.6 [$U_S(\delta(s,\sigma)) = \hat{\delta}(\hat{s},\hat{\tau})$] implies that the program must be processed by the abstract machine's transition function in order to know for certain if the result is correct. This requirement defeats the purpose of having a realm.

A close examination of Definition 3.6 shows that the requirements depend on two sets of functions: the binding/unbinding functions (B_S, B_A, and U_S), and the state transition functions (δ and $\hat{\delta}$). One can significantly reduce the difficulty of satisfying the correctness requirement by applying constraints, because the binding functions can be simplified or eliminated.

Let us constrain the state binding and unbinding functions to be identity functions. This is tantamount to constraining the machines to use the same states and alphabets. Now, the correctness criterion from Definition 3.6 is summed up as follows: $U_S(\delta(B_S(\hat{s}),B_A(\hat{\tau}))) = \hat{\delta}(\hat{s},\hat{\tau})$. When using the identity binding functions, this criterion is simplified significantly: $\delta(\hat{s},\hat{\tau}) = \hat{\delta}(\hat{s},\hat{\tau})$. But this equality, by Definition 3.3, can only be true if $\hat{s}$ and $\hat{\tau}$ are elements of the domains of both δ and $\hat{\delta}$. That is, $\hat{s} \in (\hat{S} \cap S)$ and $\hat{\tau} \in (\hat{A} \cap A)^*$. Thus, correctness is guaranteed if δ and $\hat{\delta}$ are equivalent when limiting their domains to $(S \cap \hat{S}) \times (A \cap \hat{A})^*$. Limiting the domain of a function to a subset of its defined domain is called a restriction, and it is denoted, in this case, $\delta|_{(S\cap\hat{S})\times(A\cap\hat{A})^*}$. There are several relationships between S and $\hat{S}$ and between A and $\hat{A}$ that can satisfy this correctness criterion, and they are investigated in Definition 3.8.

Definition 3.8. A migration realm $\mathcal{R}$ is a *restricted realm* of program $\hat{P}$ if for all $\mathcal{A} \in \mathcal{R}$,

$$\delta|_{(S\cap\hat{S})\times(A\cap\hat{A})^*} = \hat{\delta}|_{(S\cap\hat{S})\times(A\cap\hat{A})^*},$$

and B_S, B_A, and U_S are the identity functions. A realm $\mathcal{R}$ of $\hat{P}$ such that for all $\mathcal{A} \in \mathcal{R}$, $S = \hat{S}$ is a *state-equivalent* (SE) realm. Consider the following three constraints for FSMs of a

state-equivalent restricted realm $\mathcal{R}$ of $\hat{P}$:

1. $A = \hat{A}$ (*identity machine*)
2. $(A \cap \hat{A}) \subsetneq \hat{A}$ (*SE subset machine*)
3. $A \supset \hat{A}$ (*SE superset machine*)

If for all $\mathcal{A} \in \mathcal{R}$, constraint 1 is true, then $\mathcal{R}$ is an *identity realm*. If for all $\mathcal{A} \in \mathcal{R}$, constraint 2 is true, then $\mathcal{R}$ is an *SE subset realm*. If for all $\mathcal{A} \in \mathcal{R}$, constraint 3 is true, then $\mathcal{R}$ is an *SE superset realm*. □

Take special note of the state-equivalent realms presented above. It is ubiquitous in computing systems that state is stored in binary digits. While there are differences in byte order and floating point representation, one can assume that mappings between them are isomorphisms. The actual migration system that is motivating the development of this model consists of a set of virtual machines run in software and real machines run in hardware, and they share identical instruction sets and memories. They form an SE realm. We therefore focus on SE realms for the remainder of the this model, but future work includes the study of restricted realms that are not state equivalent. The migration system implementation shall be extended to support the compilation of processes directly to hardware which will not necessarily be state equivalent.

Theorem 3.2. *For every bound process in an SE restricted realm, there exists an unbound process.*

Proof. Let $p = (\hat{\mathcal{A}}, \hat{\sigma}, \hat{k}, \mathcal{A}, s, \sigma)$ be a bound process in SE restricted realm $\mathcal{R}$. Since B_A is the identity function, σ is both the bindable string and the bound string. Let $\hat{p} = (\hat{\mathcal{A}}, \hat{s}, \hat{\sigma}, \hat{k})$ be a process. From Definition 3.6, $\hat{p}$ is an unbound process of p if $\hat{s} = \delta(s, \sigma) = \hat{\delta}(s, \sigma)$, since U_S is the identity function. Since $\delta|_{(S \cap \hat{S}) \times (A \cap \hat{A})^*} = \hat{\delta}|_{(S \cap \hat{S}) \times (A \cap \hat{A})^*}$ for a restricted realm, it must be shown that $s \in (S \cap \hat{S})$ and $\sigma \in (A \cap \hat{A})^*$.

Since σ is the bound string, then by Definition 3.5a $\sigma \in A^*$. Since σ is also the bindable string, $\sigma \in \hat{A}^*$. Thus, $\sigma \in (A^* \cap \hat{A}^*)$. Therefore $\sigma \in (A \cap \hat{A})^*$.

The definition of a bound process indicates that $s \in S$, and for a state equivalent realm, $S = \hat{S}$. Hence, $s \in \hat{S}$. Consequently, $\delta(s, \sigma) = \hat{\delta}(s, \sigma)$. Therefore, there exists an unbound process for every bound process in an SE restricted realm.

□

Theorem 3.3. *Given program $\hat{P}$, let $\mathcal{R} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ be an SE restricted realm of $\hat{P}$. $\hat{P}$ is realm executable in $\mathcal{R}$ if $\hat{A} \subseteq A_1 \cup A_2 \cup \ldots \cup A_n$.*

Proof. For this proof, a sequence of bound processes is constructed that meets the requirements of Definition 3.7. The sequence consists of one bound process for each input symbol.

I. It must be shown that for every input symbol there exists a machine to which it is bindable. Let $\mathbb{A} = A_1 \cup A_2 \cup \ldots \cup A_n$. The definition of a program shows that for all $\hat{\tau} \in \hat{\sigma}, \hat{\tau} \in \hat{A}$. Since $\hat{A} \subseteq \mathbb{A}$, for all $\hat{\tau} \in \hat{\sigma}$, $\hat{\tau} \in \mathbb{A}$. Therefore, for every symbol in $\hat{\sigma}$ there exists a machine to which it is a bindable string.

II. Because the realm is state-equivalent, $S = \hat{S}$. Because the realm is restricted, B_S and U_S exist and they are the identity functions.

III. Now let us show that for every adjacent pair of bound processes, one can migrate from the first to the second. Let

$$S_i = \begin{cases} \hat{s_0}, \text{ for } i = 0 \\ \hat{\delta}(\hat{s_0}, \hat{\sigma}_{[0,i-1]}), \text{ for } 0 < i \leq |\hat{\sigma}|. \end{cases}$$

Let $\mathcal{A}_i$ be a machine to which symbol $\hat{\sigma}_i$ is bindable. Let bound process $p_i = (\hat{\mathcal{A}}, \hat{\sigma}, i+1, \mathcal{A}_i, S_i, \hat{\sigma}_i)$. Let $\mathbb{P} = (p_0, p_1, \ldots, p_{|\sigma|-1})$ be a sequence of bound processes, one per input symbol. To satisfy Definition 3.7, it must be proven that $p_{i+1} = M(p_i)$, for $0 < i < |\sigma| - 1$. That is, it must be proven that there exists a process $\hat{p}$ such that $\hat{p}$ is the unbound process of p_i, and p_{i+1} is a bound process of $\hat{p}$. Theorem 3.2 indicates that $\hat{p}_i = (\hat{\mathcal{A}}, \hat{\delta}(S_i, \hat{\sigma}_i), \hat{\sigma}, i+1)$ exists as the unbound process of bound process p_i. Next is proven that p_{i+1} is a bound process of $\hat{p}_i$. As defined above for $\mathbb{P}$, $p_{i+1} = (\hat{\mathcal{A}}, \hat{\sigma}, (i+1)+1, \mathcal{A}_{i+1}, S_{i+1}, \hat{\sigma}_{i+1})$. From Definition 3.6, the bound form of $\hat{p}_i$ is $(\hat{\mathcal{A}}, \hat{\sigma}, (i+1)+1, \mathcal{A}_{i+1}, \hat{\delta}(S_i, \hat{\sigma}_i), \hat{\sigma}_{i+1})$. These are identical except for the state.

$$\begin{aligned} S_{i+1} &= \hat{\delta}(S_i, \hat{\sigma}_i) \\ &= \hat{\delta}(\hat{\delta}(\hat{s_0}, \hat{\sigma}_{[0,i-1]}), \hat{\sigma}_i) && \text{(Def. of } S_i\text{)} \\ &= \hat{\delta}(\hat{s_0}, \hat{\sigma}_{[0,i-1]}\hat{\sigma}_i) && \text{(Definition 3.3)} \end{aligned}$$

$$= \hat{\delta}(\hat{s}_0, \hat{\sigma}_{[0,(i+1)-1]}) \qquad \text{(Def. of concat.)}$$

$$= S_{i+1} \qquad \text{(Def. of } S_i\text{)}$$

Therefore, p_{i+1} is a bound process of $\hat{p}_i$.

IV. Finally, to satisfy Definition 3.7, it must be shown that $p_0 = B(\hat{p})$ and that there exists $\hat{p}' = U(p_{|\hat{\sigma}|-1})$. As the elements of $\mathbb{P}$ are defined, $p_0 = (\hat{\mathcal{A}}, \hat{\sigma}, 1, \mathcal{A}_0, s_0, \hat{\sigma}_0)$. Now from Definition 3.6, $B(\hat{p})$ with a single input string of $\hat{\sigma}_0$ is $(\hat{\mathcal{A}}, \hat{\sigma}, 1, \mathcal{A}, s_0, \hat{\sigma}_0)$, and this equals p_0. By Theorem 3.2, $\hat{p}'$ exists.

Therefore, the requirements of Definition 3.7 are met, and $\hat{P}$ is realm executable in $\mathcal{R}$. □

Theorem 3.3 serves an important role for migration systems. It states that for a state-equivalent restricted realm, if the realm as a whole supports the entire input alphabet of a process, then that process is executable to completion within the realm. For an implementation, an SE restricted realm's processors differ only in the extent to which they support the instruction set. Figure 3.4 shows graphically the concepts of SE restricted realm executability as well as alphabet relationships in SE subset and superset machines.

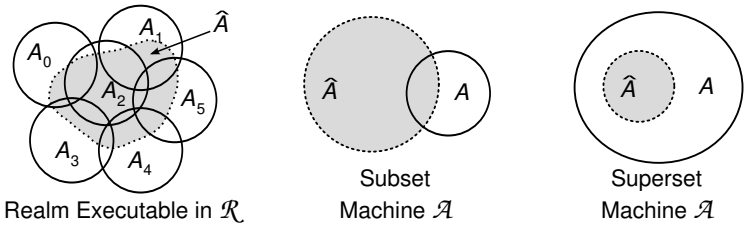


Figure 3.4. Executability: For a state-equivalent restricted realm, a program is realm executable when all the machines together can process the entire input string. The two right-most diagrams demonstrate the relationship of the alphabets in SE subset and superset machines.

In a subset realm, any given machine supports only a subset (including the empty set) of the program's input symbols. A homogeneous subset realm is an interesting variation, but provides no value. By definition, no single machine can execute the entire program, but since all machines are identical, the entire realm cannot execute the program to completion. Formally, it does not fulfill the requirements of Theorem 3.3.

Theorem 3.4. *A homogeneous subset realm is never realm executable.*

Proof. Theorem 3.3 presents the requirement for realm executability in a subset (restricted) realm, where $\hat{A} \subseteq A_1 \cup \ldots \cup A_n$. For a homogeneous realm, $A_1 = A_2 = \ldots = A_n$, and the requirement simplifies to $\hat{A} \subseteq A_1$, which can never be true because it is mutually exclusive with the subset realm requirement that $(A_1 \cap \hat{A}) \subsetneq \hat{A}$. □

Let us next consider a superset realm. This is a realm composed of machines that all provide a superset of the program's alphabet. This is the type of realm that is often called a homogeneous cluster. A superset realm can be either homogeneous or heterogeneous, and the properties are identical: From the perspective of the process, all machines in the realm might as well be homogeneous because they can all execute the entire program.

One final type of realm to consider is the identity realm. An identity realm is a homogeneous realm consisting of one or more FSMs that implement exactly the abstract FSM of a program. Theorem 3.1 proves that a program is realm executable in an identity realm consisting of a single machine. The fact that it supports the minimum required alphabet translates into the smallest processor in an implementation that can complete the program. Using the characteristics of the realms defined above, we form a taxonomy shown in Figure 3.5.

During the explanation of this model, the input symbols are viewed as being analogous to instructions for a microprocessor, but the level of abstraction is not relevant to the model. For instance, a PowerPC processor and an x86 processor are considered heterogeneous because they have different instruction sets, among other significant differences. However, if the programs are modeled at a higher level of abstraction, then those two machines can be considered homogeneous. Two practical examples of abstraction which can make two heterogeneous processors behave in a homogeneous fashion are Remote Procedure Calls and the Java™ platform (`http://www.javasoft.com/`).

3.5 Application to Hardware/Software Migration

In the RTL acceleration application, the desire is to improve the performance of digital logic simulation by running the processes describing the behavior of the circuit in parallel in

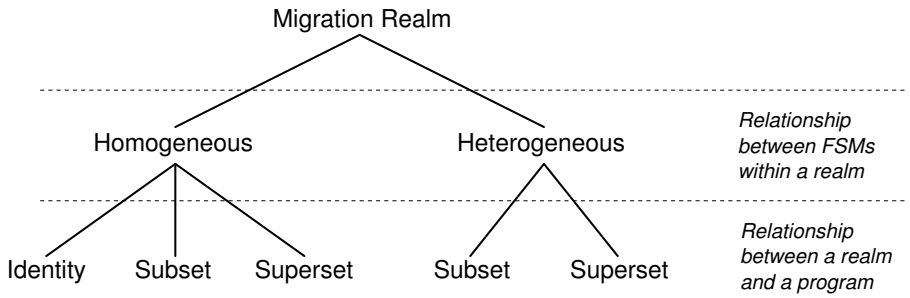


Figure 3.5. Migration Realm Taxonomy

an FPGA. Since the parallel hardware resources are limited, they must be used efficiently by migrating idle processes out and busy processes in. The performance of a process migration system depends upon the implementation, and there are three penalties that need to be considered: instantiation overhead, migration overhead, and area utilization. The instantiation overhead is the time required to reconfigure a portion of the FPGA at run time to create or remove a processor. The migration overhead is the time to bind and unbind a process to a processor in the FPGA. The area utilization is a measure of the use of the logic structures of the FPGA. Minimizing the area utilization maximizes the number of processors.

This model shows that a completely heterogeneous realm provides no value to the execution of a process because of the problem of knowing if the result is correct. But fully heterogeneous systems are only theoretical: Computer systems all share homogeneous concepts such as add and multiply. Through requiring certain homogeneous characteristics, some of the most expensive requirements for correctness are eliminated. In contrast to full heterogeneity is full homogeneity, typically in the form of a superset realm for which a compiler ensures that the program is a subset of the supported states and alphabet. A homogeneous superset realm, however, does not maximize the use of an FPGA's area. A more efficient use is to place a process within a processor that supports just the instructions the process utilizes—an identity realm. The instantiation overheads, however, must also be considered. If there already exists one or more processors in the FPGA that aggregately support the process's instruction set (an SE subset realm), then completion is guaranteed without incurring instantiation overhead.

Example 3.4. Continuing the example of the "computer" acceptor, Figure 3.6 shows that a subset realm can provide the means to complete the recognition of "computer" while also

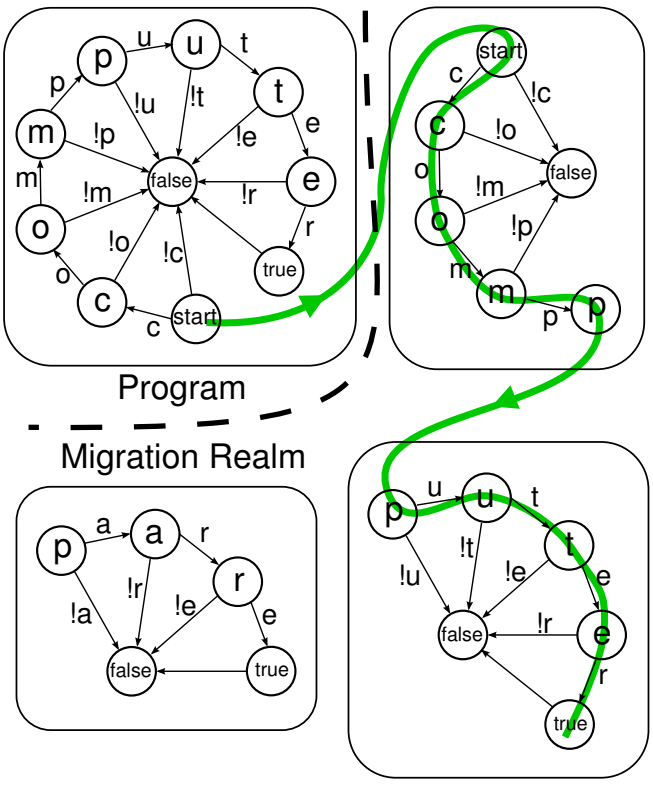


Figure 3.6. An Example of Subset Migration

providing recognition of "compare." With the sharing of the prefix acceptor "comp" via migration, less area is used within an FPGA compared to two complete FSMs for "computer" and "compare." In the migration realm, there are 11 arcs for a complete path, which would correspond directly to processing logic. Without a migration realm, two independent FSMs would require 15 arcs. The observant reader will notice that the migration realm shown in Figure 3.6 is not a state-equivalent realm. These FSMs could be augmented with a complete set of states to make them state equivalent, but since the example merely demonstrates the principle of a subset realm, the required space in the figure is not warranted.

The type of realm that is used is highly dependent on the implementation and the trade-offs between performance and area utilization, and with run-time reconfiguration of an FPGA, the realm can dynamically change, for example, from a superset realm for one process to a subset realm for another. Figure 3.7 shows a qualitative arrangement of the three useful types of migration realms. The most efficient in migration overhead and instantiation overhead is the superset realm because it is likely that no instantiation is required, and in

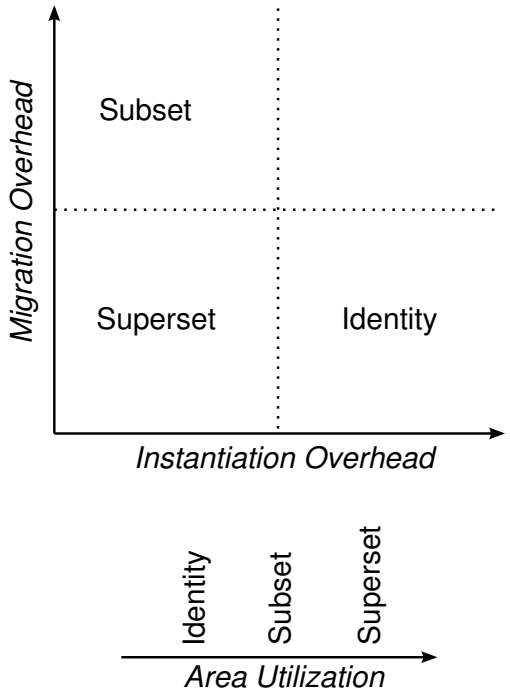


Figure 3.7. Realm Overhead and Area Utilization

order to complete the execution of a process, only one migration is required. Also shown, however, is that the superset realm requires the greatest area resources in the FPGA because it contains more than is necessary to execute the program to completion. The smallest area to support a single program is provided by the identity realm where by definition the system instantiates only the instructions required, although subset realms may utilize less area for multiple programs as demonstrated by Figure 3.6. But an identity realm potentially requires an instantiation of a processor for every migration, thus incurring two overheads. The subset realm may require less instantiation overhead because the assumption is that the processors that provide the execution will be reused by multiple processes, but the migration overhead increases because no single processor can execute the program to completion. Any optimal solution based on migration realms is likely to use a hybrid approach.

Nevertheless, all of the useful realms developed in this chapter share a common characteristic: They are state-equivalent restricted realms. Using Theorem 3.3, an algorithm can be proposed that allows a system to guarantee complete execution of a process within such a realm. Execution units in the FPGA will be configured, reconfigured, defragmented, and

removed during execution, but if a list of the instructions supported by the realm at any given time is maintained, the system can check a process against the list to determine realm executability. Another method of ensuring realm executability is to always maintain one machine that is a superset machine of all the processes. Then, regardless of what kind of subsets are in the remainder of the realm, there is always this superset machine to fall back on.

Theorem 3.5. *An SE restricted realm of $\hat{P}$ having one superset machine is realm executable.*

Proof. Let $\mathcal{A}'$ be a superset machine of $\hat{P}$ in the SE restricted realm $\mathcal{R} = \{\mathcal{A}_0, \ldots, \mathcal{A}_n\}$. Since $\hat{A} \subsetneq A'$, then it follows that $\hat{A} \subseteq (A' \cup A_0 \cup \ldots \cup A_n)$, satisfying Theorem 3.3. Therefore $\hat{P}$ is realm executable in $\mathcal{R}$. □

The latter method is the one employed in the migration system implementation presented in Chapter 5. Because of the limited FPGA resources, it is possible that all of the processes in the realm cannot fit into parallel execution units in the FPGA, and so the system maintains the remainder of the processes in software superset machines, thus guaranteeing realm executability regardless of the contents of the FPGA.

3.6 Future Work

The model is complete enough to draw some conclusions, but future work is required. The development of theorems for non-state-equivalent migration realms is of special interest due to migrations such as the one shown in Figure 3.6. The utilization of an FPGA can be reduced further by having a machine work with a subset of the states, but correct execution must be guaranteed.

Furthermore, a migration realm and a program can be represented as a directed graph with FSMs as nodes and arcs as migrations, as the example shown in Figure 3.8. In Definition 3.7, realm executability is expressed in terms of a finite sequence of bound processes, and the execution path can be used to construct the arcs and nodes of a graph. The graph starts at the program's abstract FSM, and each element of the execution path forms the arc and the endpoint for each hop of the path. The labels on the arcs denote the substrings of the

input string bindable to the FSM at the arc's endpoint. A path through the graph represents the executability of the process through the realm.

If there exist multiple paths through the graph, then an optimization algorithm could be used to determine the best path. The overhead incurred from a migration includes the time required to unbind and bind a process. If the overhead is small, then the least number of hops through the graph may not necessarily be optimal. With large migration overhead, however, the system should likely minimize the number of hops. Furthermore, because the FPGA is dynamically reconfigurable, the graph is dynamic with paths potentially being completely disrupted during the course of execution.

Another consideration arising from realms as directed graphs, is the execution of multiple processes within the realm. Figure 3.8 shows two processes, one with solid arcs and the other with dashed arcs. Places where the processes' paths share a common FSM (in the example, $\mathcal{A}_4$ and $\mathcal{A}_5$) denote a potential bottleneck, and these bottlenecks may require the use of non-optimal paths for individual processes to improve the execution time of an entire group of processes.

3.7 Conclusion

This chapter presents a model that describes programs, processes, and the migration of those processes among a group of finite state machines. It formally establishes the criteria that must be met in order for the execution of the process to be correct and complete. The definition of different types of realms results in a simple migration realm taxonomy, and the theorems guide the development of algorithms to guarantee complete and correct execution in efficient ways.

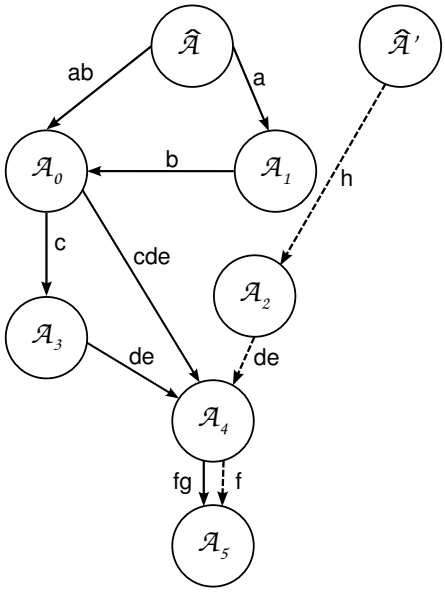


Figure 3.8. A Realm as a Digraph

Chapter 4

Processes and Temporal Locality

The efficiency of a system migrating busy processes to a limited set of parallel processors depends fundamentally on the exploitation of executive locality of reference described in Chapter 1 on page 4. Spatial executive locality is relatively straightforward to determine in a simulator because a simulator must already track the input/output dependencies among processes. These dependencies are known either at compile time or, at the latest, elaboration time. Executive temporal locality is a more difficult property to measure: An implementation must identify idle and active processes at run time. A process is a sequence of code that is made sensitive to one or more simulator events, and every time one of those events occurs, the process code is executed. If all the processes are always active during the simulation, then the parallel execution of those processes that fit in the RMs results in only a modest speedup dictated by Amdahl's Law (see Section 2.5 on page 24), and only a single, initial migration is necessary. If, however, the processes have phases of activity, migration achieves greater efficiency by keeping the parallel RMs fully utilized, and allowing software to handle the idle ones.

The contributions of this chapter fall into two sections: Section 4.1 analyzes processes and transformations that can be used on processes for efficiency, and Section 4.2 describes how process activity can be determined. The process analysis of the first section shows that coded processes can be decomposed into canonical units, and that those units can be combined

Listing 4.1. VHDL Clocked Process

```
1 main: process(clk, reset_l)
2 begin
3     if(clk'event and clk == '1') then
4         A <= B + C;
5     end if;
6     if(reset_l = '0') then
7         A <= (others => '0');
8     end if;
9 end process;
```

Listing 4.2. Verilog Equivalent of Listing 4.1

```
1 always @(posedge clk, negedge reset_l)
2 begin
3     A <= B + C;
4     if(reset_l == 0) begin
5         A <= 0;
6     end
7 end
```

in multiple ways depending on the needs of the system. These code transformations are then applied to existing HDL source to demonstrate equivalent behavior. The subsequent section examines these transformed code bases as well as untransformed code bases to demonstrate executive temporal locality of reference. These two sections support the thesis by showing that executive locality of reference does exist in RTL source code, and that a migratory simulator has great flexibility in manipulating processes to exploit it.

4.1 RTL Processes and Canonical Processes

The two most common HDLs are Verilog and VHDL. VHDL defines a "process" within its grammar, an example appearing in Listing 4.1. While Verilog does not explicitly define a "process" within its grammar, the structure of an `always` block (shown in Listing 4.2) corresponds to a VHDL `process`. Some ambiguity arises, however, because other syntactic structures within these languages also infer simulation processes. Furthermore, a single VHDL `process` may be executed as multiple simulation processes.

The *IEEE Standard for Verilog Hardware Description Language* defines a process as follows:

> A design consists of connected threads of execution or processes. Processes are objects that can be evaluated, that may have state, and that can respond to changes on their inputs to produce outputs. Processes include primitives, modules, initial and always procedural blocks, continuous assignments, asynchronous tasks, and procedural assignment statements [32, p. 158].

The *IEEE Standard VHDL Language Reference Manual* defines all non-process syntax elements in terms of processes. Note the two following instances of characteristic wording within the specification:

> A concurrent procedure call statement represents a process containing the corresponding sequential procedure call statement [33, p. 135].

> Elaboration of all concurrent signal assignment statements and concurrent assertion statements consists of the construction of the equivalent process statement followed by the elaboration of the equivalent process statement [33, p. 170].

In other words, the following two listings are identical simulation processes in VHDL, although only one is coded as a grammatical `process`:

```
D <= E + F;
```

```
process(E, F) begin
    D <= E + F;
end process;
```

In the context of this document, "process" refers to a sequence of code executed by the simulator in response to an event in a sensitivity list, and it does not necessarily correspond to a "process" defined by the HDL standards. Constructs such as concurrent assignments will be executed in a simulation process, but due to code transformations or optimizations the correspondence is not necessarily one-to-one. The standards define processes in such a way that a coder may instantiate a number of processes where one would suffice, as illustrated in Listing 4.3. In that listing, the two upper `always` blocks can be combined into the single

Listing 4.3. Equivalent Functionality, Varying Number of Processes

```
1  // Assignments in two processes
2  always @(posedge clk)
3      output0 <= input0 + input1;
4  always @(posedge clk)
5      output1 <= input0 - input1;
6
7  // Equivalent assignments in one process
8  always @(posedge clk) begin
9      output0 <= input0 + input1;
10     output1 <= input0 - input1;
11 end
```

lower block. For the upper blocks, the standards define two processes. For the lower block, they define one.

A compiler or simulator, however, may merge multiple processes into a single process even if it is written as two. The Verilog standard explicitly allows active processes to be executed in any order during an event processing [32, p. 160], and the VHDL standard does not specify the order of process execution within the simulation cycle [33, p. 177], tacitly allowing any order. That is to say, two processes that share identical sensitivity lists are triggered on the same event and can be run in any order. A simulator, therefore, is free to combine processes in any order when their sensitivity lists are equivalent. In typical fashion, vendors provide little information about their optimizations, but the ModelSim documentation briefly mentions cases where the process merging optimization may be turned off [62, p. 419][63, p. 348]. One can conclude that such optimizations are utilized in commercial simulators.

Both HDL specifications require the order of statement execution within a process to be preserved [32, p. 160][33, p. 177]. However, there are cases when the order need not be preserved, but changes to order must be thoroughly analyzed to ensure that the result is identical to the result obtained by the written order. While such execution-order transformations are a direct violation of the letter of the specifications [33, p. 117][32, p. 139], if it can be proved that the result is the same as the original-order result, then it stands to reason that the specifications are satisfied. Indeed, the Verilog standard allows such an interpretation: “Verilog HDL simulators are free to use different algorithms from those described in this clause, provided the *user-visible*

effect is consistent with the reference model" [emphasis added] [32, p. 158]. The remainder of this section is dedicated to demonstrate such consistency.

4.1.1 Related Work

Proving that reordered statements give the same results as the original order would require a formal model. Some formal modeling of simulation has been done by Hering [64], but this work was done at a structural level (e.g., it models latches) rather than at the RTL process level. For example, there is no concept within Hering's formal model of events and process execution triggering, which are necessary and essential parts of RTL simulation modeling. Ubar, *et al.* present a method of increasing cycle-based simulation speed through the use of decision diagrams [65]. The work is similar to the concepts presented here except that they focus on run-time optimizations using decision diagrams to reduce the number of expressions that are evaluated each cycle. The work presented here describes compile-time decomposition and optimization through transformations which allow the grouping of similar processes later at run time. Reducing the number of expression evaluations at run time is orthogonal to decomposing and merging processes.

There has also been some recent work in simulation modeling for clock suppression [66]. Clock suppression is the stopping of a clock for a time period of a cycle-based simulation, and then updating process outputs to the correct value when the clock is resumed. In the meantime, no execution time is spent on the process thus giving a speedup. For example, a counter with value 1 can have its clock suppressed for 9 cycles. After the nine cycles, the output can be updated to 10 and the clock turned back on, but the simulator must be able to determine whether the counter's output is used by other processes during this period of clock suppression. Such a technique also requires the simulator to recognize that the process implements a counter so the correct value can be driven at the end of the suppression. Similar functions must be known for every other process in the design which will be clock-suppressed. This approach differs from what is presented here in two respects. First, the actual behavior of an RTL simulator needs to be modeled, whereas the clock suppression model is based on a semantic interpretation of processes. That is, in RTL there is no concept of a clock—it is merely

a signal defined like every other signal in the simulation—but the clock suppression model is based on a clock. Second, the goal of clock suppression is direct simulation acceleration, and the goal of the work here is the identification and exploitation of locality of reference (which is used to accelerate by a different means).

A good model of event-driven RTL simulation was developed by Kupriyanov, *et al.* [67]. Unlike the other models, this one specifically addresses the issue of process sensitivity, and they develop the means to construct a minimal set of sequential elements that need to be updated each simulation cycle. They also develop a partitioning method "to minimize the number of evaluations of combinational nodes during a simulation cycle by the introduction of intermediate registers." This is essentially common subexpression elimination at run time [68].

The Kupriyanov model may be usable as a basis to prove HDL transformation equivalence, but it approaches the problem from a *minimal evaluation* perspective rather than a *minimal process* perspective. The difference is critical: A minimal evaluation model focuses on inputs and is beneficial for serial execution on a single processor; it minimizes the number of evaluations to increase performance. A minimal process model focuses on a process output to discover the expression (and therefore the minimum number of inputs) only required to calculate that output, and so expressions may be duplicated rather than eliminated. Eliminating common subexpressions across processes makes them less parallel because it introduces a common dependency. Furthermore, Kupriyanov's model decomposes all processes into sequential *elements* ("latches") and combinational *operations* (such as add, multiply, compare), whereas a minimal process model seeks to combine all input expressions into sequential processes. The Kupriyanov model also borders on being a logic-equations-level model rather than an RTL model because `if` and `case` statements are synthesized into combinational elements such as multiplexors. This model also does not address how processes may be combined as is required by the transformations below. It is believed that this decomposition of the RTL into sequential and combinational *elements* (rather than processes) makes the Kupriyanov model overly complex for analyzing minimal process algorithms, and the model is much like using decision diagrams to reduce logic during synthesis.

The purpose of the next sections is to demonstrate that RTL process transformations can be developed to provide a migratory simulator with basic units to work with without

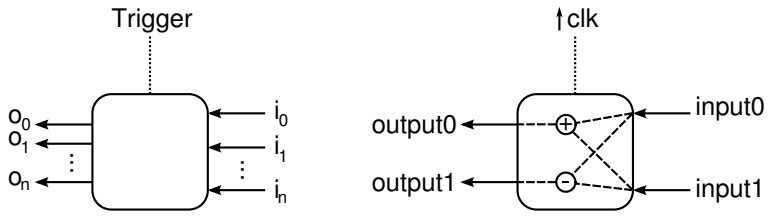


Figure 4.1. Illustrations of Simulation Processes

affecting the behavior of the design. To that end, three HDL designs are simulated after applying transformations to their code to verify correct execution. While a full development with proofs of these transformations would be beneficial, for the purpose of supporting the thesis let it suffice to use a semi-formal approach with definitions and examples to demonstrate equivalence by induction. However, future work may include a completely formal development of these transformations.

4.1.2 Definitions of RTL Processes

The previous code examples on pages 52 and 53 demonstrate that the same functionality can be written in multiple ways with a varying number of processes according to the HDL standards. One thing that is common, however, is that every process consists of a set of functions to be evaluated with the process's inputs when at least one of a set of sensitivity events occurs. The general form of a process is illustrated in Figure 4.1 on the left, with a concrete example from Listing 4.3 on the right. In Verilog, the events are part of the sensitivity list and consist of `posedge` which triggers on the rising edge of a signal, `negedge` which triggers on the falling edge of a signal, and a trigger on any change of the signal, referred to herein as the `any` event.[1] In VHDL, the sensitivity list is composed only of signal names, and the process is run on any change in at least one of those signals. However, falling and rising edges are detected within the process code using standard constructs such as the following [69, p. 7]:

```
if(clk'event and clk = '1')  --- Detect rising edge
```

[1] There is one more event within these languages, and that is a delay expiry. This event is not considered in this model since the focus is on synthesizable RTL modeling where timing is irrelevant. Hard-coded delays in RTL code are unnecessary, and using back-annotation gives a false sense of timing closure when static timing analysis should be used.

A compiler can detect these standard constructs [69, p. 7] and incorporate the rising and falling edge qualifiers into the event list. The following notation is used to represent events within a sensitivity list: ↑ is `posedge`, ↓ is `negedge`, and ↕ is `any`.

A *canonical RTL process* (CRP) is a function whose evaluation is triggered by a `true`-valued event in the sensitivity list. An HDL process can be decomposed into CRPs such that each CRP contains the expressions required to determine the value of a single `signal`, `wire`, or `reg` at the completion of the process. The sensitivity list is shown within the function enclosed in square brackets with each event delineated with the "|" symbol. Table 4.1 shows several examples of HDL processes and their canonical forms. For the sake of clarity, the canonical forms can also be shown with arbitrary pseudo-code as in the last example of the table, but defining the mode of expressing the function is outside the scope of this document.

Table 4.1. HDL and Canonical Forms

HDL	***Canonical Form***

`always @(posedge clk) begin`
`  A <= B + C;`
`end`
	`A = [↑clk] B + C`

`process(clk, reset_l) begin`
`  if(rising_edge(clk)) then`
`  A <= B + C;`
`  end if;`
`  if(reset_l = '0') then`
`  A <= (others => '0');`
`  end if;`
`end process ;`
	`A = [↑clk|↕reset_l]`
	`  reset_l ?  B + C : 0`

`always @(posedge clk) begin`
`  A <= 0;`
`  if(enable) A <= B + C;`
`end`
	`A = [↑clk] enable ?  B + C : 0`

(Continued on next page)

Table 4.1. HDL and Canonical Forms (continued)

HDL	*Canonical Form*

`A <= B + C;`
`process(clk) begin`
`  if(rising_edge(clk)) then`
`  if(enable) then`
`  E <= A + D;`
`  end if;`
`  end if;`
`end process ;`
	`A = [↕B|↕C] B + C`
	`E = [↑clk] enable ?  A + D : E`

`process(clk) begin`
`  if(rising_edge(clk)) then`
`  A <= '0';`
`  case(state) is`
`  when IDLE =>`
`  if(enable = '1') then`
`  state <= ENABLED;`
`  A <= '1';`
`  end if;`
`  when ENABLED =>`
`  if(enable = '0') then`
`  state <= IDLE;`
`  end if;`
`  end case;`
`  end if`
`end process ;`
	`A = [↑clk]`
	`  (state == IDLE && enable == 1) ?`
	`  1 :  0`
	`state = [↑clk]`
	`  if(state == IDLE && enable == 1)`
	`  state = ENABLED`
	`  else if(state == ENABLED &&`
	`!enable)`
	`  state = IDLE`
	`  else state = state`

A CRP can be classified into two types, *combinational* and *sequential*. A combinational CRP is one in which all inputs to the function are contained in the sensitivity list with an event of `any`. Otherwise, the CRP is sequential. These two types are illustrated in Figure 4.2.

Using this CRP model, the validity of re-ordering statements within a process is now considered. The ultimate goal is to demonstrate the validity of decomposing a process as coded so that portions of it can be executed in parallel. The problem in doing so is that the execution of sequential statements from a single process in parallel is a reordering in direct violation

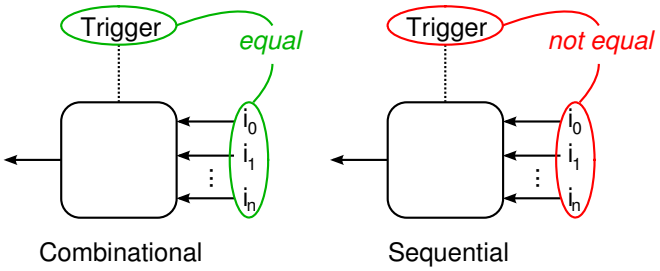


Figure 4.2. Types of Canonical RTL Processes

of the letter of the specifications. By way of example, consider the following code listing:

```
1 always @(posedge clk) begin
2     a <= 0;
3     if(enable) begin
4         a <= b + c;
5     end
6     c <= c_next;
7 end
```

An examination of this simple `always` block shows that line 2 cannot be separated from lines 3–5 into a separate process without changing the behavior of the code. The assignment of signal `c` on line 6, however, has no dependencies on lines 2–5. The process, therefore, could be coded as follows with identical results:

```
1 always @(posedge clk) begin
2     c <= c_next;
3     a <= 0;
4     if(enable) begin
5         a <= b + c;
6     end
7 end
```

This RTL process can therefore be decomposed into two CRPs:

```
c = [↑clk] c_next
a = [↑clk] enable ?  b + c :  0
```

The HDL specifications provide that multiple assignments to the same variable are allowed within a block of sequential code, and the variable takes on the value of the last executed

assignment [32, pp. 121, 160][33, p. 123] [2]. A CRP embodies the statements that produce the final value of a signal assignment, and since the order of evaluating non-blocking assignments to different signals is irrelevant, CRPs may be executed in any order.

4.1.3 Process Transformations

In order to accurately profile the execution of HDL code, processes must be delineated regardless of any particular coding style. If the canonical form were to be used, then each expression within the HDL would be reduced to its simplest form, but such a form would incur greater simulation event-processing overheads. There are optimizations that can be made on the canonical form to reduce the event-processing overheads without changing the functionality of the code.

Merging Combinational into Sequential

Consider the following Verilog code fragment:

Listing 4.4. CRP Merge Candidates

```
1 assign A = B + C;
2
3 always @(posedge clk)
4 begin
5     D = A - F;
6 end
```

The canonical forms of the two processes inferred by this code are

$$\texttt{A = [}\updownarrow\texttt{B|}\updownarrow\texttt{C] B + C, and}$$

$$\texttt{D = [}\uparrow\texttt{clk] A - F.}$$

`A` is a combinational CRP and `D` is a sequential CRP. This form indicates that the output of `A` is evaluated on any change of `B` or `C`, and the output of `D` is only evaluated on the rising edge of `clk`. Substituting the right-hand-side of `A` into `D` yields

$$\texttt{D = [}\uparrow\texttt{clk] ([}\updownarrow\texttt{B|}\updownarrow\texttt{C] B + C) - F.}$$

[2] In VHDL, this behavior is defined as a series of *transactions* on a signal's *driver*, and *old transactions* are deleted leaving only the last transaction active at the end of the process execution.

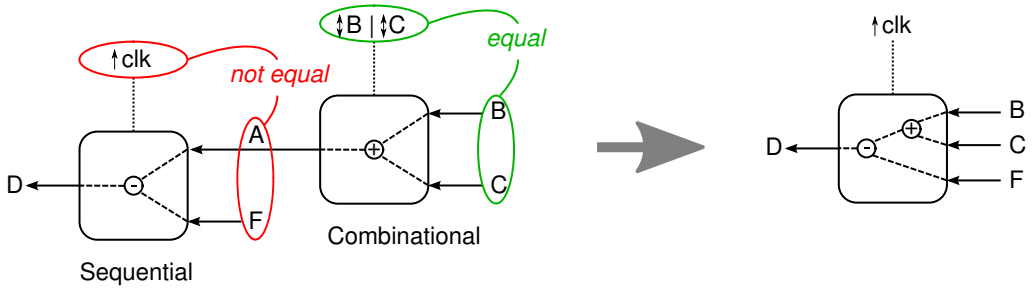


Figure 4.3. Merging a Combinational CRP into a Sequential CRP (from Listing 4.4)

The Verilog specification indicates that active events can be processed in any order, so if ↕B or ↕C occurs at the same simulation time as ↑clk, there is a race condition between the evaluations of A and D. The value of D is therefore indeterminate at the end of the simulation cycle [32, pp. 159-161]. Similarly, VHDL does not specify the order in which concurrent statements (including processes) are executed [33, p. 133], so if ↕B or ↕C occurs in the same simulation cycle as ↑clk, the result is undefined. If, however, ↕B or ↕C occurs before ↑clk, then the value of A is already determined when D is evaluated. Therefore, the events ↕B|↕C have no effect on the evaluation of D, and the simulator is free to optimize Listing 4.4 to the following CRP:

```
D = [↑clk] B + C - F
```

This merging is illustrated in Figure 4.3. At the source-code level, Listing 4.4 becomes the following, eliminating a simulation process:

```
1 always @(posedge clk)
2 begin
3     D = B + C - F;
4 end
```

This optimization is the merging of combinational processes into sequential processes in order to reduce the number of processes. It also reduces the number of events that must be serviced within the simulator. The vast majority of digital designs are synchronous, and through functionally equivalent process transformations, the design can be optimized so that only clock edges are simulated. For example, consider a synchronous design having a single

clock domain. After merging combinational processes into sequential processes, every process is now sensitive to the rising edge of the clock. Since every process has the same sensitivity list, the simulator can combine all of them into a single process, and the functionality remains identical. This optimization forms the basis for cycle-based simulators. Combinational processes cannot be eliminated, however, if they describe outputs of the design under test. A combinational output of a sub-module may be merged into a sequential process of its parent in the hierarchy only if the output of the combinational process is an input to the sequential process.

Converting Concurrent Assignments to Processes

As mentioned previously in Section 4.1 on page 52, the VHDL LRM defines concurrent assignments as being equivalent to a process, and it delineates the method for arriving at an equivalent [33, pp. 137–139]. For Verilog, an `assign` statement can be written as an `always` statement, but the target must be converted from a `wire` to a `reg`. This conversion introduces some subtle differences in behavior, but as far as synthesis is concerned, there is no difference, for example, between the following two constructs:

```
wire [31:0] A;
assign A =  B + C;
```

```
reg [31:0] A = {32{z}}; // wire default is z [32, p. 23]
always @(B or C) A <= B + C;
```

Since PLI calls cannot be inserted into `assign` statements, this transformation allows them to be inserted for profiling, and it is only required for the profiling described in Section 4.2.1.

Splitting Processes into CRPs

Consider the code of Listing 4.5. The output values of `A`, `B`, and `C` are independent because they are non-blocking assignments. The reordering of lines 4–6 does not change the behavior of the code, and so executing them in parallel is also possible. They may, then, be

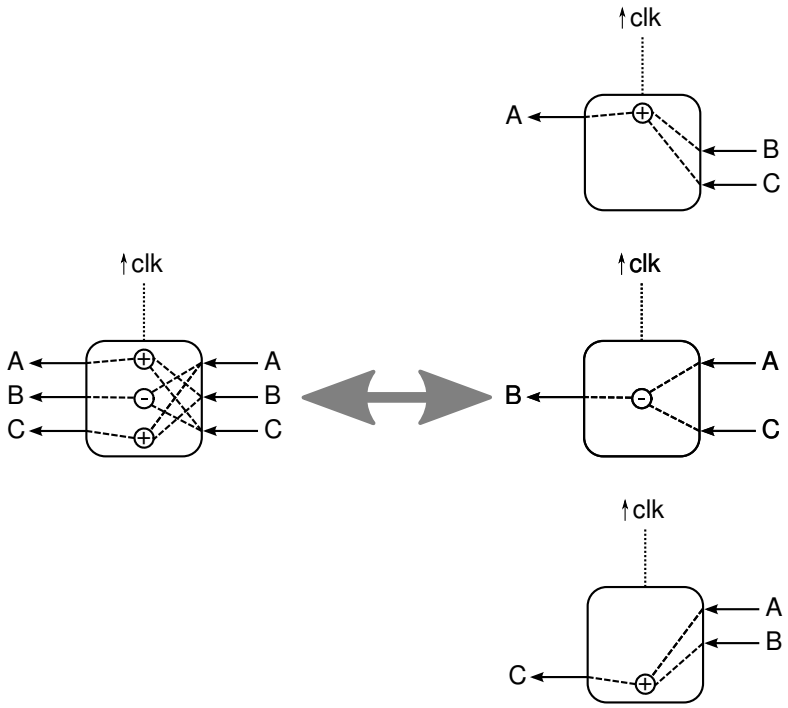


Figure 4.4. Splitting or Merging RTL Processes (from Listing 4.5)

separated into CRPs with no change in functionality. A progression in Figure 4.4 from left to right illustrates this transformation.

Merging Processes with Identical Sensitivity Lists

Because both HDL specifications allow concurrent statements to be executed in any order during a simulation cycle, there is no change of behavior when merging two or more processes that have the same sensitivity list. Doing so merely imposes an order that the specifications do not require. Such a transformation is demonstrated in Listing 4.6, and illustrated in Figure 4.4 in a right-to-left progression.

Application of Transformations

These transformations were applied to the Ethernet MAC, ATA Controller, and the AC97 Controller from OpenCores. The transformations were applied by hand with some assistance of a Verilog parser which automated some common transformations such as converting `assign` statements into `always` blocks. The goal of these transformations was to minimize the

Listing 4.5. Splitting Processes

```
1  // Three assignments in one process
2  always @(posedge clk)
3  begin
4    A <= B + C;
5    B <= A - C;
6    C <= A + B;
7  end
8
9  // Split into three CRPs
10 always @(posedge clk) A <= B + C;
11 always @(posedge clk) B <= A - C;
12 always @(posedge clk) C <= A + B;
```

Listing 4.6. Merging Processes

```
// Three independent processes
always @(posedge clk) begin
  A <= B + C;
  D <= C - A + D;
end
always @(posedge clk) B <= A - C;
always @(posedge clk) C <= A + B;

// Merged into one process
always @(posedge clk)
begin
  A <= B + C;
  B <= A - C;
  C <= A + B;
  D <= C - A + D;
end
```

number of processes within each module. In many cases, all the processes were combined into a single sequential process and a single combinational process. No attempt was made to remove the hierarchy of the design to combine processes across module boundaries, but further reduction in process count could be done in that manner. In all cases, the testbenches ran successfully after the transformations.

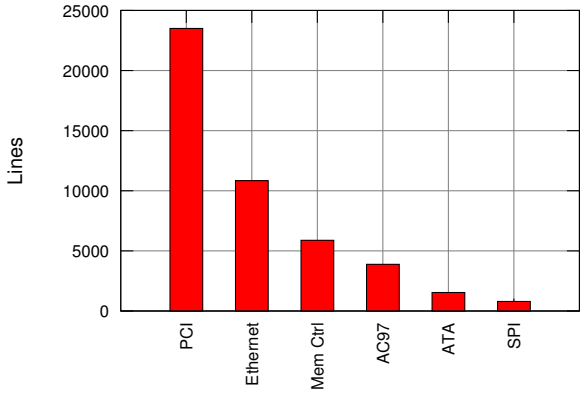
These transformations give precise control to a migratory simulator in monitoring and migrating processes. A CRP provides a function for a single signal in the design, and whenever the trigger occurs, the CRP is evaluated. A design can be decomposed into a set of sequential CRPs and combinational CRPs, but the number of combinational CRPs will be minimized by the transformation developed in 4.1.3. The remaining sequential CRPs can be simulated using cycle-based methodologies. Since CRPs with identical sensitivity lists can be combined in any way, a migratory simulator can apply optimizations with great flexibility. For example, all CRPs having identical triggers and identical sets of inputs can be combined to minimize process-to-process communication. Less-than-optimal combinations of CRPs can also be made if they share common subsets of inputs. These optimized processes can then be distributed across parallel execution units for parallel speedup and load balancing. In a migratory system that implements JIT compiling of processes to hardware, the CRPs form a basic unit of functionality that can be synthesized relatively quickly. The opportunities for combining CRPs are numerous and are left for future work.

4.2 Detecting Locality of Reference

To detect executive locality of reference, two things are necessary: First, the activity of processes must be detectable. Second, the activity must be measurable. The goal of this section is to demonstrate that executive locality does exist in RTL code in support of the thesis. The goal is not to investigate it to the fullest extent possible, but further investigation will be warranted in a complete system where algorithms must be developed to make migration decisions. To demonstrate executive locality, existing code that was not written specifically to exhibit locality must be analyzed.

Table 4.2. Sizes of Profiled Code

Code Base	*Lines*
PCI/WB Bridge	23,503
Ethernet MAC	10,841
Memory Controller	5,880
AC97 Audio Controller	3,884
ATA Controller	1,539
SPI Interface	794



4.2.1 Profiling Existing Code

In Chapter 2 on page 22, phases of activity in a simulation are discussed in a qualitative way. It is often the case within a testbench that functionality of the device under test is exercised independently, so as the simulation progresses different processes will be active and idle at different times. In order to gain a better understanding of the execution characteristics of simulations, the HDL descriptions of actual devices were profiled. The devices include an Ethernet MAC, an ATA disk controller, an AC97 audio controller, a Serial Peripheral Interface (SPI) controller, a PCI/Wishbone Bridge, and a memory controller from OpenCores [70]. These designs vary in size from a mere 794 lines of code to over 23,000 (Table 4.2). Cadence Design System's Logic Design Verification (LDV) tools version 5.1 were used. LDV provides a `-PROFILE` option to the ncsim simulator, but the output of that profiling is not very useful for run-time activity analysis. The report shows the number of "hits" a "stream" receives during an entire simulation, a stream being a "generated code stream" including `always` blocks, `initial` blocks, continuous assignments, etc. [71, p. 902]. In other words, "streams" are processes. A "hit" is an execution of a process, but the number of hits a process receives during the entire simulation is only useful for overall performance tuning, and not for demonstrating phases of simulation. The activity of the processes must instead be measured at various times as the simulation progresses.

It is therefore necessary to develop a different means of detecting process activity. Verilog provides a Programming Language Interface (PLI) that allows users to call functions

Listing 4.7. Record/Report Profiling Example[†]

```
always @(posedge clk, negedge reset_l)
begin
    $record(5); // Record a timestamp
    if(enable) begin
        a <= b;
        c <= d;
    end
    if(!reset_l) begin
        a <= 0;
        c <= 0;
    end
    $report(5); // Calculate the execution time
end
```

[†]The code structure for reset in this listing may be unfamiliar to some readers. Please see Appendix A for details.

outside the simulation environment and return values to it. Through this interface, profiling information was gathered using a Record/Report Sequence illustrated in Listing 4.7. Each process in the code is instrumented with a `$record()` PLI call to record a time stamp and a `$report()` call to calculate the elapsed time from the `$record()` and report it to a profiling infrastructure. The argument to the PLI calls is a uniquely-assigned number used to delineate the execution times among the profiled processes. The profiling infrastructure also delineates between process numbers that appear in multiple module instantiations.

The results were measured in units of CPU cycles on the host machine using the Time Stamp Counter (TSC) of Intel processors [72], but the results proved to be disappointing. Further investigation shows that the reason stems from two problems:

1. Some of the processes in this OpenCores code do very little while others include significant computation. For example, the general style of the MAC's code has the assignments of clocked processes separated into many `always` blocks with very little computation in each, and a relatively few larger blocks are the exceptions. It is difficult to see execution trends when the time delta between active and inactive processes is small, especially on a host system with a preemptive operating system.

2. Every clocked process is executed every simulation cycle, so at every cycle, some expression of every process is evaluated to determine, for example, if a control signal is active. Nothing else is done if the signal is not active, but this behavior incurs processor execution overhead for an otherwise idle process.

To address Problem 1 above, a system may apply the transformations described in Section 4.1 to balance the load across processors. The system also has significant flexibility in combining CRPs to keep the acceleration resources of a migratable system busy. Measuring the execution time of CRPs may be difficult in a preemptive system, but the count of the number of instructions may be a good alternative. Whereas, within the acceleration hardware, precise time measurement of execution could be implemented with little overhead.

The second problem of running processes when they may actually do nothing, can be addressed in at least two ways. One method is to detect when the inputs are unchanged which is discussed in Section 4.2.3. Another other method is through process guides.

4.2.2 Process Guides

A process guide is a user-supplied code sequence that informs the simulator whether the process is active or not. If the code is not active according to this guide, then the process need not be run. Often designs are modeled at several levels of abstraction, and utilizing a Transaction Level Model (TLM) would be useful as an RTL guide.

The OpenCores Ethernet controller contains a receive data path and a transmit data path. The transmit path is framed by a `TxStartFrm` signal and a `StartTxDone` as shown in Figure 4.5. After the `StartTxDone` signal is high at the rising edge of the clock, one more cycle is necessary for the block to complete its transactions and then it goes idle. The process guide used to delineate activity in this process implements the state machine shown in Figure 4.6. As long as this state machine is in the `IDLE` state, the process is not run. A simulator's infrastructure would have access to all the state of the signals and could control the execution of a process quite easily. Since the LDV source code is not available for modification, the guide must be implemented entirely with PLI calls. To bypass the execution of a process, then, the Record/Report sequence described in Section 4.2.1 on page 67 is modified to return a value,

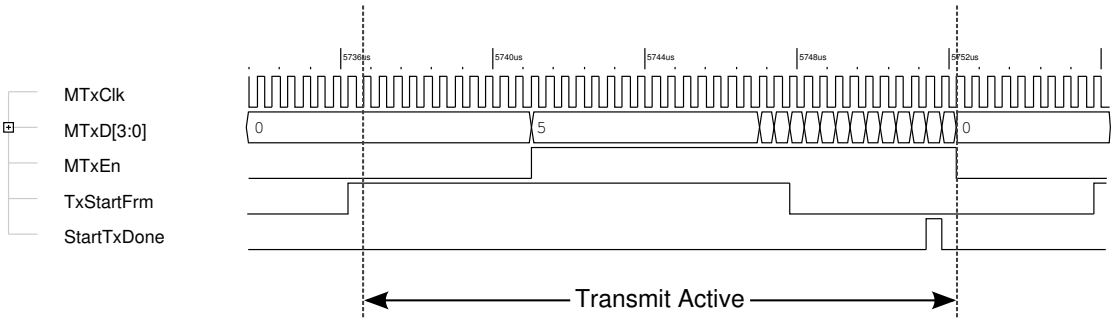


Figure 4.5. Framing of Transmit Activity

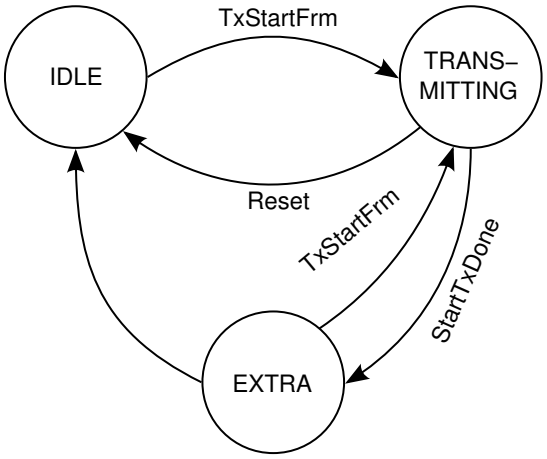


Figure 4.6. State Machine for the Transmitter's Guide

Listing 4.8. Conditional Record/Report Sequence

```
1  always @ (posedge MTxClk or posedge Reset) begin
2    if($record(12, 2, TxStartFrm, StartTxDone, Reset)) begin
3       if(Reset) begin
4          StopExcessiveDeferOccured <= 1'b0;
5          ColWindow <= 1'b1;
6          . . .
7       end else begin
8          . . .
9          PacketFinished_q <= PacketFinished;
10      end
11   end
12   $report(12);
13 end // always @ (posedge MTxClk or posedge Reset)
```

becoming a Conditional Record/Report Sequence. The relevant code from the Ethernet MAC's process is shown in Listing 4.8. The arguments to `$record()` include the process number as before followed by a guide function number and the signal arguments to the function guide. The `2` as the second argument on line 2 identifies which guide controls the activity of this code. A production implementation would hide such details and would provide a alternate method to supply the guide, perhaps as a pragma. The actual code of the guide is shown in Listing 4.9.

This example is typical of the types of guides that can be constructed for various processes of the Ethernet MAC. The information gathered by using guides is shown in Figure 4.7. Without guides, these sequential processes would be active 100 percent for each test, but with the guides, the absence of activity is easily detected. As a general rule, the receive processes shown on the first row are largely idle for the "Register" and "Transmit" tests, and the transmit processes in the second row are largely idle for the "Register" tests and the "Receive" tests. The activity for these processes on tests 0, 3, and 4 is due to resets. These tests are relatively short in terms of triggers, and they reset the MAC as part of the tests triggering some activity.

The cost of implementing guides is high in terms of time to implement, so not all designs lend themselves well to their use. Guides were only written for 7 of the 87 processes of the Ethernet MAC to demonstrate their use, and writing them for the entire design would be tedious. Some of the other designs available from OpenCores are small compared to the Ethernet MAC, and their small size makes the identification of signals on which to base guides difficult.

Listing 4.9. Actual Transmit Process Guide

```
static int transmit(void)
{
    int TxStartFrm, StartTxDone, Reset;
    static enum {
        IDLE, TRANSMITTING, EXTRA
        } state = IDLE;
    Reset = acc_fetch_tfarg_int(5);
    if(Reset) {
        state = IDLE;
        return 1;
    }
    TxStartFrm = acc_fetch_tfarg_int(3);
    StartTxDone = acc_fetch_tfarg_int(4);
    switch(state) {
        case IDLE:
            if(TxStartFrm) {
                state = TRANSMITTING;
                return 1;
            }
            break;
        case TRANSMITTING:
            if(StartTxDone) {
                state = EXTRA;
            }
            return 1;
            break;
        case EXTRA:
            state = IDLE;
            if(TxStartFrm) {
                state = TRANSMITTING;
            }
            return 1;
            break;
    }
    return 0;
}
```

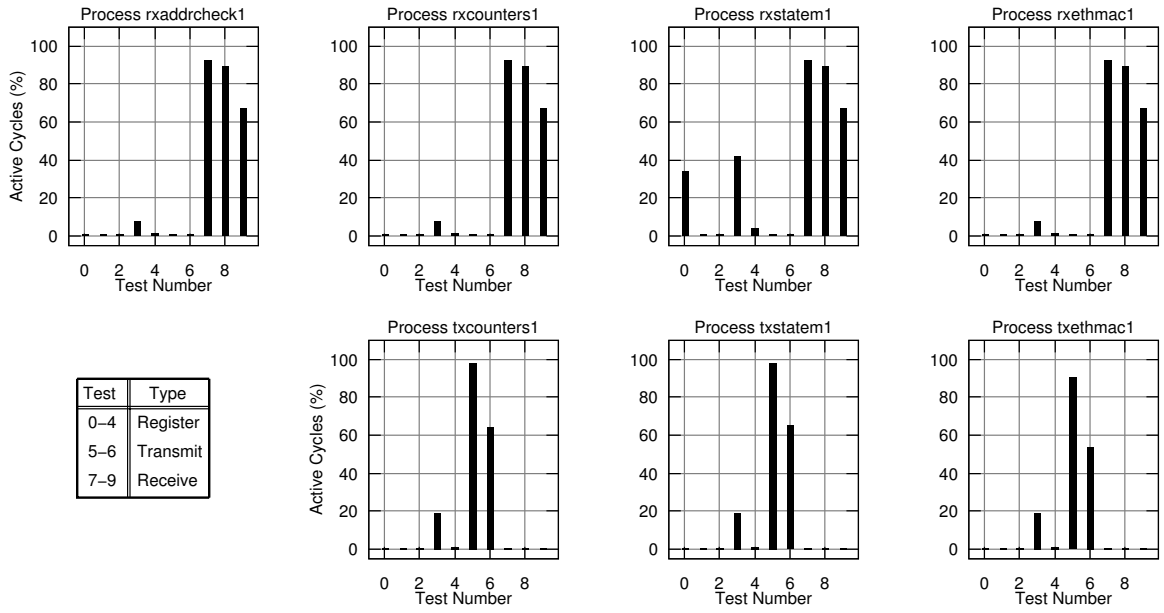


Figure 4.7. Percentage of Active Cycles Per Process

Guides, then, are applicable where the benefit is great compared to the cost of implementing them. Such a case exists in System on Chip (SoC) designs where multiple, independent devices are interfaced on a shared bus. A guide in that case could be very high-level, monitoring the inputs and outputs of an entire device on the bus. A better, general-purpose method of detecting locality of reference, however, is the use of input monitors.

4.2.3 Input Monitors

The basic principal of input monitoring is that a process does not need to be run if its inputs do not change. Each process in a simulation is essentially a deterministic function[3], and if the arguments of a function do not change, the value of the function does not change. The idea of monitoring inputs is not new and is often called "clock suppression," although it is not limited to clocked processes. Its use appears as early as 1993 where the idea was applied to gate level simulations where speedups of 2.8 to 4.7 were achieved [73].

Input monitors keep track of the values of all the inputs of a process, a function similar to the *sensitivity update mappings* of Kupriyanov's RTL simulation model [67]. The inputs

[3]The addition of PLI or system functions to code alters the determinism of simulation processes, but typical synthesizable code does not use them.

may or may not be included in the process sensitivity list; rather, they include signals on the right-hand side of expressions as well as signals in `if` statement conditionals, `case` choices, and the like. If, from one process trigger to the next, the inputs of a process do not change, then the process does not need to be executed since the output will not change.

The processes of the six OpenCores designs were analyzed for locality of reference using this input-monitoring method, and results are shown in a set of graphs for each design. Detecting a change in a signal is fairly easy to implement within a simulator's infrastructure, but the PLI needed to be used since the source code for LDV is not available. The PLI-based implementation adds `always` blocks to monitor inputs. A typical clocked process input monitor is shown in Listing 4.10 using `$check_sig()` along with `$record()` and `$report()`. The arguments to `$check_sig()` in this example associate the input monitoring to process 12 using signal values 0 through 2. The `$register_sig()` call tells the monitoring infrastructure at the start of the simulation that a process has monitored inputs and their number. None of these PLI function calls would be necessary if source code were available for the simulator.

There are three metrics gathered for each process as the test benches of the designs were run. First is the number of times a process is executed during a simulation, referred to as "hits." The second is the amount of time required to execute each process, shown in the graphs as the "Process Load." The third is the size of each process. The "size" of a process is difficult to define, and its definition varies depending on the context. In software simulation environments, the number of native processor instructions could be used to specify the size, while in hardware acceleration environments, the number of gates could specify the size. Neither of these metrics is known for the simulation of the OpenCores code, so the next reasonable metric available for size is the number of RTL code lines within each process. While this metric does not necessarily relate accurately to gate-count or instruction-count, it is sufficient for the qualitative nature of this study.

In a multitasking, pre-emptive system, getting precise measurement of a process's execution time is difficult. A time stamp is taken in the `$record()` and `$report()` functions, and the difference gives the time to execute the process. If, however, a preemption occurs in the middle of executing a process, the recorded time is artificially large. The means adopted in this study is to use the minimum time for any execution of the process. There is a high

Listing 4.10. Input Monitor Implemented with PLI Functions

```
always @out_le          $check_sig(12, 0, out_le);
always @sr              $check_sig(12, 1, sr);
always @sdata_in_r      $check_sig(12, 2, sdata_in_r);

initial $register_sig(12, 3);

always @ (posedge clk) begin
    if($record(12, 0)) begin
        if(out_le[0]) begin
            slt0 <= sr[15:0];
        end
        if(out_le[1]) begin
            slt1 <= sr;
        end
        if(out_le[2]) begin
            slt2 <= sr;
        end
        if(out_le[3]) begin
            slt3 <= sr;
        end
        if(out_le[4]) begin
            slt4 <= sr;
        end
        if(out_le[5]) begin
            slt6 <= sr;
        end
        sr <= {sr[18:0], sdata_in_r};
    end
    $report(12);
end // always @ (posedge clk)
```

probability (depending on the number of hits) that the minimum time represents the execution of the process without being preempted.

The LDV simulator tests were run on a 1.8 GHz Intel Pentium M processor (Family 6, Model 13, Stepping 6), and the process execution times were measured in units of processor clocks. The Intel family of processors have a Time Stamp Counter (TSC) within them that can be used by software for profiling [72]. This counter is a 64-bit counter that increments every clock cycle, and the Process Load is expressed in terms of cycles in the subsequent graphs.

4.2.4 OpenCores Execution Characteristics

After instrumenting the OpenCores code, the test benches were run to gather information about their execution. In some cases, such as the PCI/Wishbone Bridge, there are hundreds of tests, and space prohibits presenting all the data from them. In many cases, therefore, a subset of the testbench results is shown. Appendix D shows the specific tests that were run for the results shown.

Each set of graphs shows four types of data for a single simulation consisting of multiple regression tests. The topmost graph is the processes' RTL line count (minus comment lines and blank lines) so that conclusions can be drawn by comparing the size of the process and its activity. In the cases of the PCI/Wishbone Bridge and the Memory Controller, there are a relatively few processes that are significantly larger than the rest, so the line counts are shown as two plots with different vertical scales. The remaining graphs show data for each of several tests. The data include the Trigger Count of each process (the light red line using the red scale), the Hit Count of each process (the red bars), and the Process Load of each process (the green bars). In many cases, the Trigger Count tracks the Hit Count for a process (see processes 0 – 2 in Figure 4.8 for example). This is an indication that the process is a combinational process such that it is invoked only when an input changes (a "hit"). In the other cases, there is often a significant difference between Trigger and Hit Counts. This difference shows the benefit of using input monitors for clocked processes to avoid execution of the process when the inputs do not change. The Trigger Count plots may also show the use of different clock domains within the design as represented by different heights.

The purpose of these graphs is to identify exploitable executive temporal locality of reference. Temporal locality occurs when a process is active or inactive for the period the graph covers, which can be identified by the Hit Count and the Trigger Count. A low Hit Count with respect to the Trigger Count is directly exploitable because the process would not need any hardware resources. A high Hit Count is exploitable if the Process Load is also high, identified by relatively high Hit Count along with a relatively high Process Load in the graphs. The fewer the processes that exhibit this characteristic, the less hardware resources are required to accelerate the design. A high Hit Count with a low Process load may be a candidate for RM execution, but the benefit is not as great as one with a high Process Load. So the key to look for in the graphs is a strong correlation between Hit Count and Process Load, and when relatively few processes exhibit that pattern, the simulation exhibits exploitable executive temporal locality.

Ethernet Controller

Figure 4.8 and Figure 4.9 on pages 80 and 81 show the results for the Ethernet Controller. The first item of note in these graphs is the relatively small processes from about 10 to 50. These processes implement the software programmable register set of the MAC; their implementation could have been written much more efficiently. A module is instantiated for each register, and they use common control signals which results in activity in every register's process even though a single register is being accessed. This behavior can be seen in tests 0 through 4 (register tests) where all the register processes have identical hit counts with virtually no process load. These processes are almost completely idle in tests 5 through 9 which are transmit and receive tests. Processes 0 to 5 are also related to the registers, and show considerable process load for the register tests.

Of special note in these plots is process 83. This process is the largest in terms of lines of code, and it contributes the greatest to the process load. This process is a clocked process that handles the majority of the Wishbone Bus, the host-side interface to the MAC. This process is active for virtually every transaction to the device. Similarly, process 78 is a process which buffers signals to and from the Wishbone Bus and also contains synchronizing registers between the transmit and receive clocks into the Wishbone clock domain. Thus this

process has a high number of hits, but contributes little to the processor load. Processes around 50 to 75 are the transmit and receive processes which are largely idle except for tests 6, 7, and 9, the 100 Mbps traffic tests. Tests 5 and 8 are transmit and receive tests as well, but they are run at 10 Mbps instead of 100 Mbps, and thus the Wishbone Bus processes dominate those tests.

Memory Controller

The Memory Controller is a large design in terms of process count, but the line count graph at the top of Figures 4.10 and 4.11 shows that one process in particular dominates the code. This process 262 is the primary Wishbone Bus interface, and it contributes significantly to both the hit counts and the process load, making it the primary candidate for acceleration. Processes 100 through 160 comprise the "Open Bank and Row Tracking Block" and show virtually no activity except in Test 6. Yet the line count graph shows that it contributes significantly to the line count. Similarly, processes 4 and 5 have large contributions in terms of line count, but do not contribute much to the load of the simulation. As with the Ethernet MAC, only a few processes dominate the Process Load.

PCI/Wishbone Bridge

The most prominent aspect of the PCI/Wishbone Bridge is the erratic Trigger Count plots. This behavior is caused by the two clock domains of the Wishbone Bus and the PCI clock. Synchronizing logic is coded into modules which are then instantiated for the two domains. The profiling code places instantiations of the same process adjacent to each other in number thus causing the saw-tooth plot.

The tests for this bridge chip are listed in Appendix D in Table D.6. Concerning the execution characteristics, the plots show that processes 182–247 are almost completely idle. These processes deal with FIFO control and synchronization. The Line Count graph shows that these processes are not trivial in size. The process contributing the most load to the simulation is process 245. While not the largest in size, this process was consistently active and makes a good candidate for migration to the hardware.

Processes 80–125 are instantiations of the same process. Similarly, processes 126–171 are instantiations of a similar process. Both sets of processes form the output registers of the design. They are quite small in terms of lines of code, but they contribute significantly to the execution of the simulation. This behavior appears to be the result of bad RTL coding practice. Rather than implement the outputs as an array of vectors, the output buffers are instead instantiated one bit at a time. The RTL code would function much more efficiently if they were coded as vectors rather than bits. Nevertheless, as written, they provide good opportunities for migration to acceleration hardware as well as merging as described in "Merging Processes with Identical Sensitivity Lists" on page 63.

Processes 8–40 handle delayed transactions which occur relatively infrequently. Processes 255–273 constitute the Wishbone Master code which is idle until later tests such as 12–14. Processes 274–298 are related to the Wishbone Slave which is active in tests 1–7 and idle elsewhere.

AC97 Audio Controller

The AC97 controller shows the most dramatic locality of reference. There are a significant number of processes that are invoked often but have virtually no activity. In Test 0, for example, processes 0–57 have fewer than 2000 hits, and the majority of them have less than 600 hits. That is 0.1 percent of the 580,145 triggers. The reason is that processes 60, 61, 67, and 68 contain cycle counters used to trigger input and output on the serial port and to control reset de-assertion. The serial output and serial input is controlled by a much slower clock, and so these counters contribute the vast majority of the simulation time. It is a rather inefficient design.

ATA Disk Interface

The ATA Disk Interface is second-smallest of the designs in terms of lines of code with only 23 processes. Tests 0 and 1 are data tests and show activity throughout all the processes, although there are processes that clearly dominate the process load. Tests 2 and 3 are interrupt and reset tests and are therefore much shorter and exercise a smaller number of processes.

Serial Peripheral Interface

The SPI controller is the smallest of the designs, having a total of 794 lines of code, and yet it also demonstrates executive locality, especially in tests 1, 6, and 7. Test 0 is the writing of registers, and Test 1 is the reading (and verifying) of registers. The subsequent tests are the transmission and reception of data (through a loopback SPI model) with varying packet lengths, bit orders, and features. Table D.5 shows the specifics about the tests. The tests are quite short, but one can still see variation in process load and hits within each test and to some degree from test to test.

Summary

The conclusion of these graphs is that locality of reference exists in all of the designs, albeit to differing degrees. The red bars within the graphs demonstrate executive temporal locality where a few processes dominate the hit count. The green bars, which generally correlate well to the red bars, indicate when the locality is exploitable. Migrating the processes with the largest process load to parallel hardware gives the greatest benefit. The line count graphs show that in many cases, there are significant processes that are idle during the tests. Using acceleration resources for idle processes would be a waste of those resources.

4.3 Conclusion

This chapter explores the characteristics of RTL simulations to show that they exhibit executive temporal locality of reference. Through the use of process guides or input monitors, it was shown that temporal locality of reference is detectable and present in the code of several designs. Using the metrics shown in the graphs, a migration-capable simulator can identify candidates for migration. This executive locality of reference is the core property which migration-based accelerators can exploit. Furthermore, the chapter showed that a simulator has great flexibility with canonical RTL forms to decompose and combine processes to maximize efficiency in connectivity and load balancing of processes.

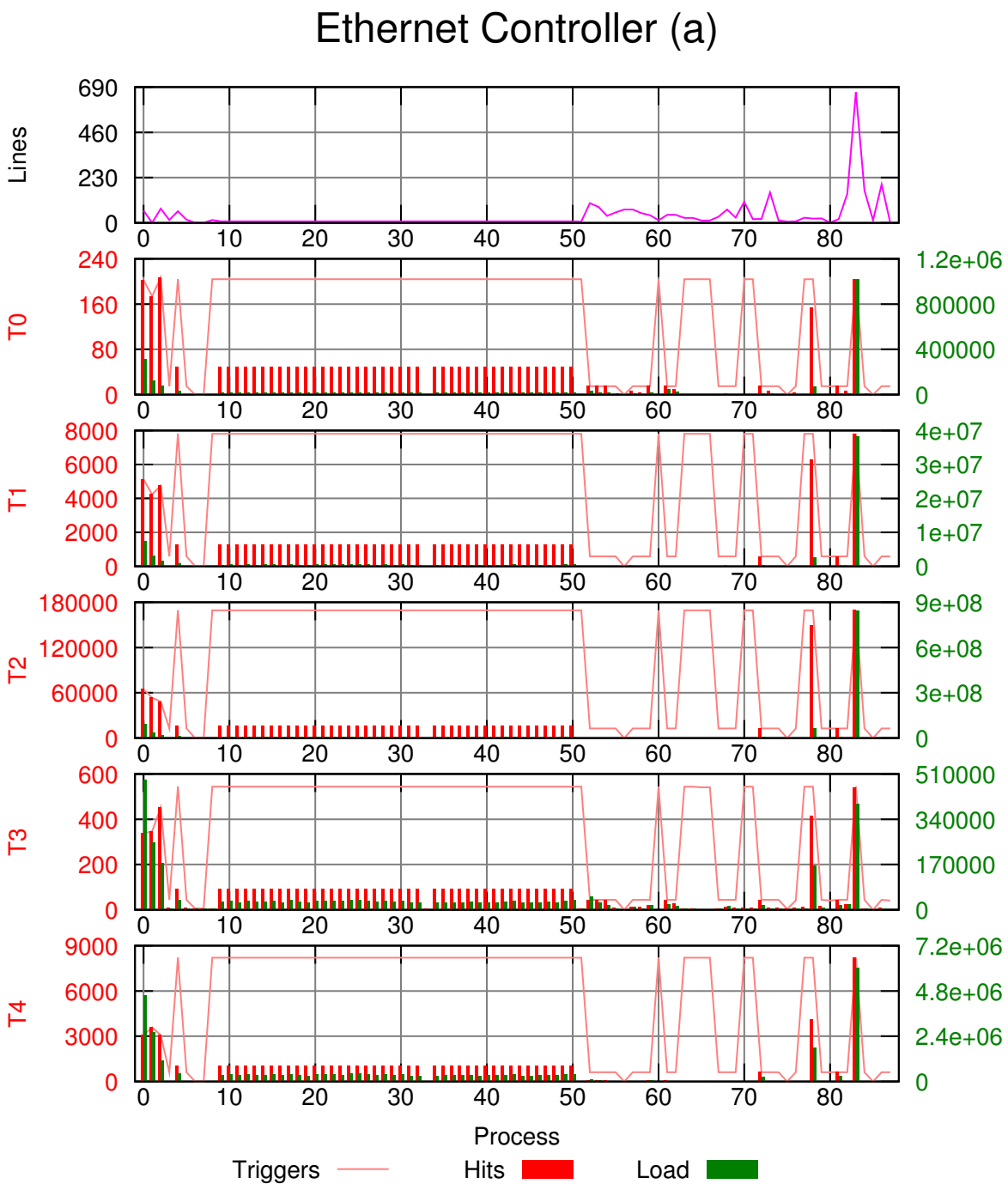


Figure 4.8. Process Characteristics of the Ethernet Controller (a)

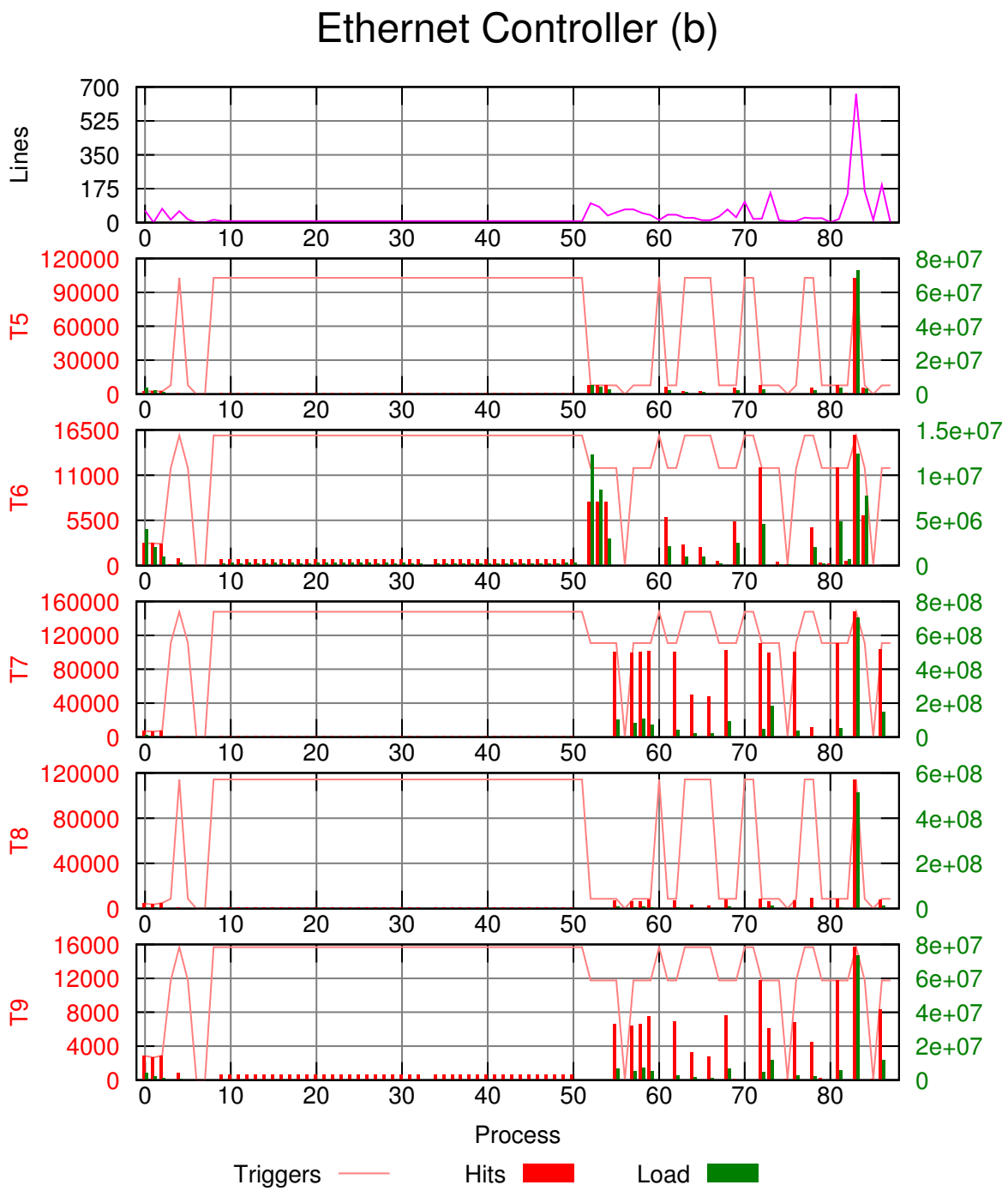


Figure 4.9. Process Characteristics of the Ethernet Controller (b)

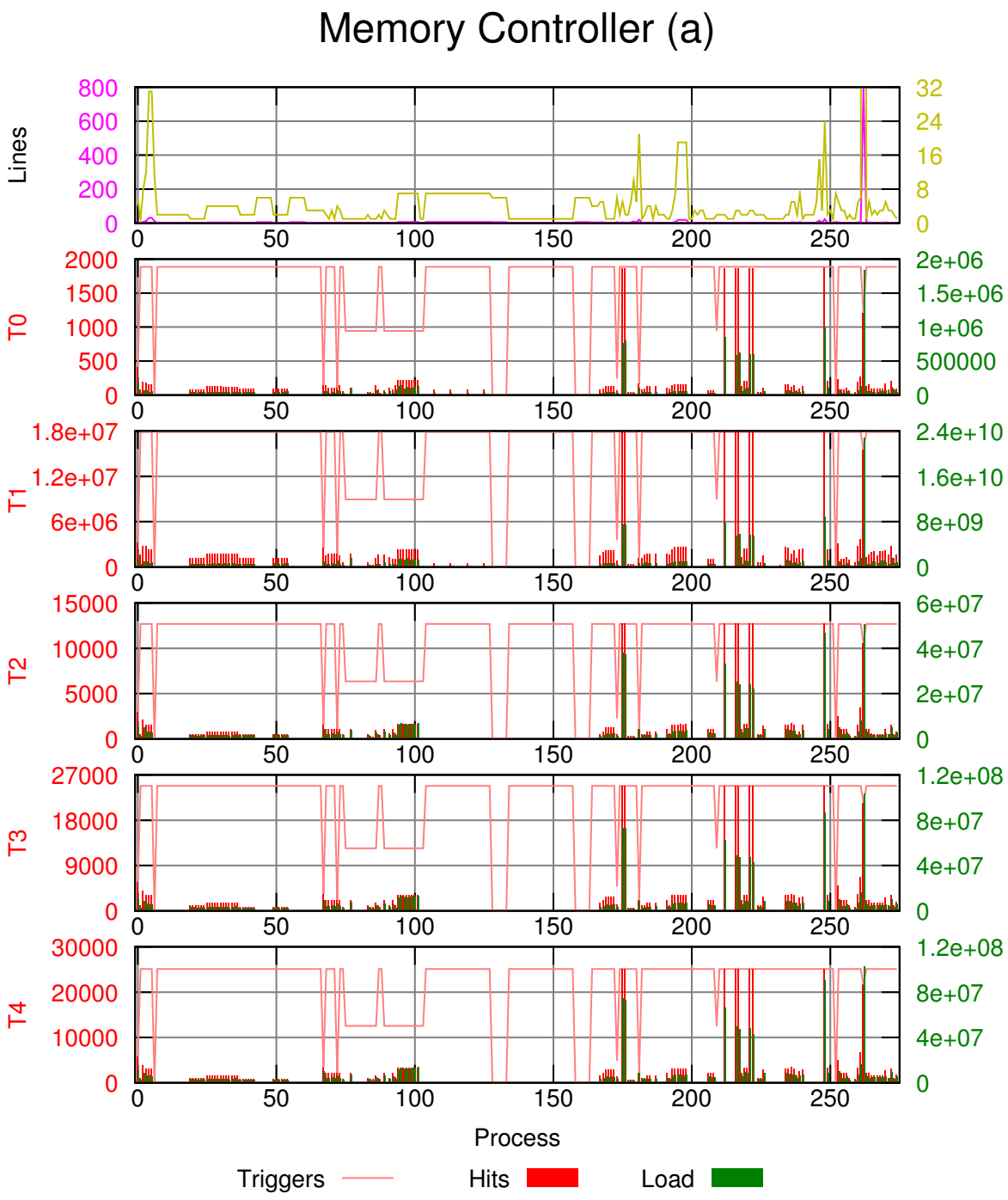


Figure 4.10. Process Characteristics of the Memory Controller (a)

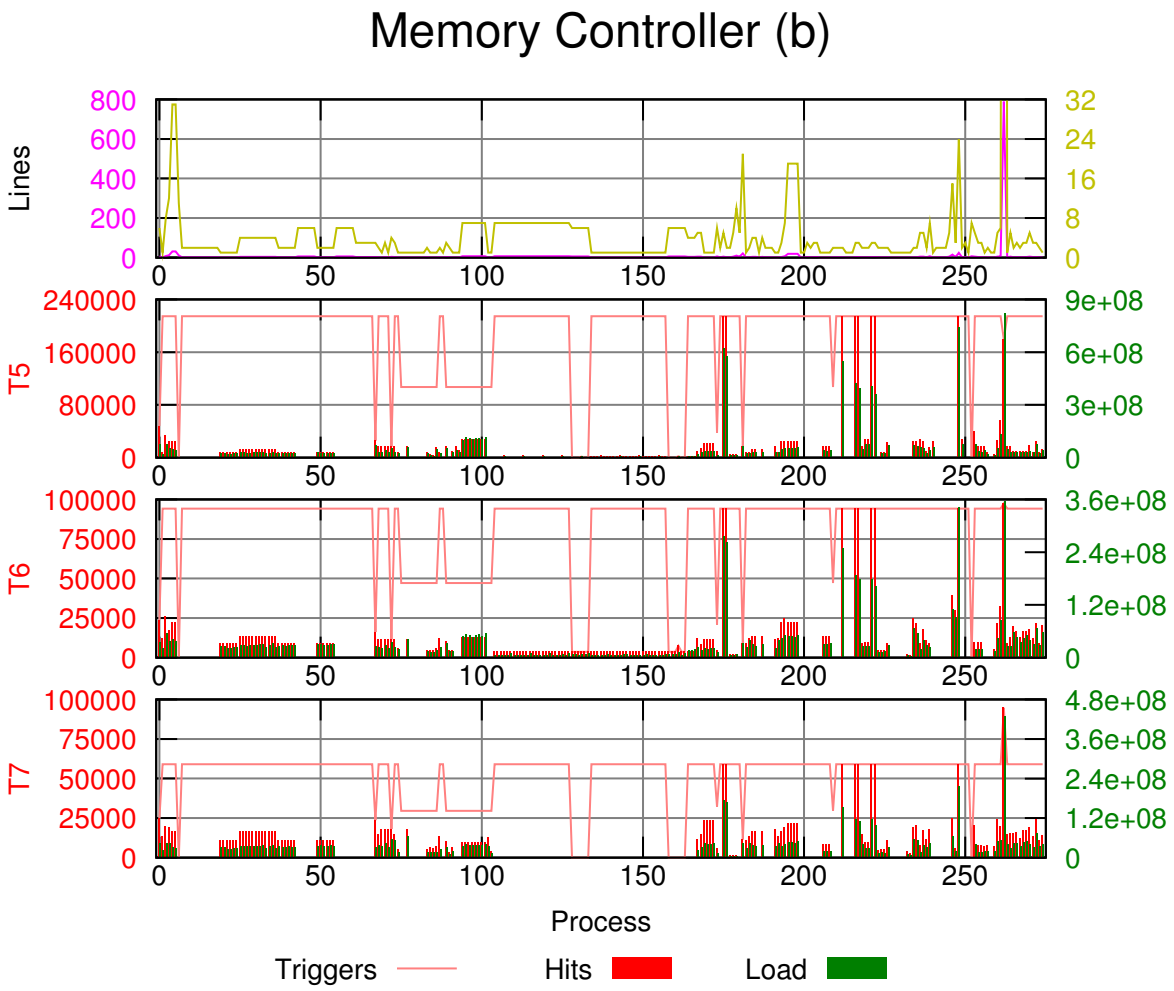


Figure 4.11. Process Characteristics of the Memory Controller (b)

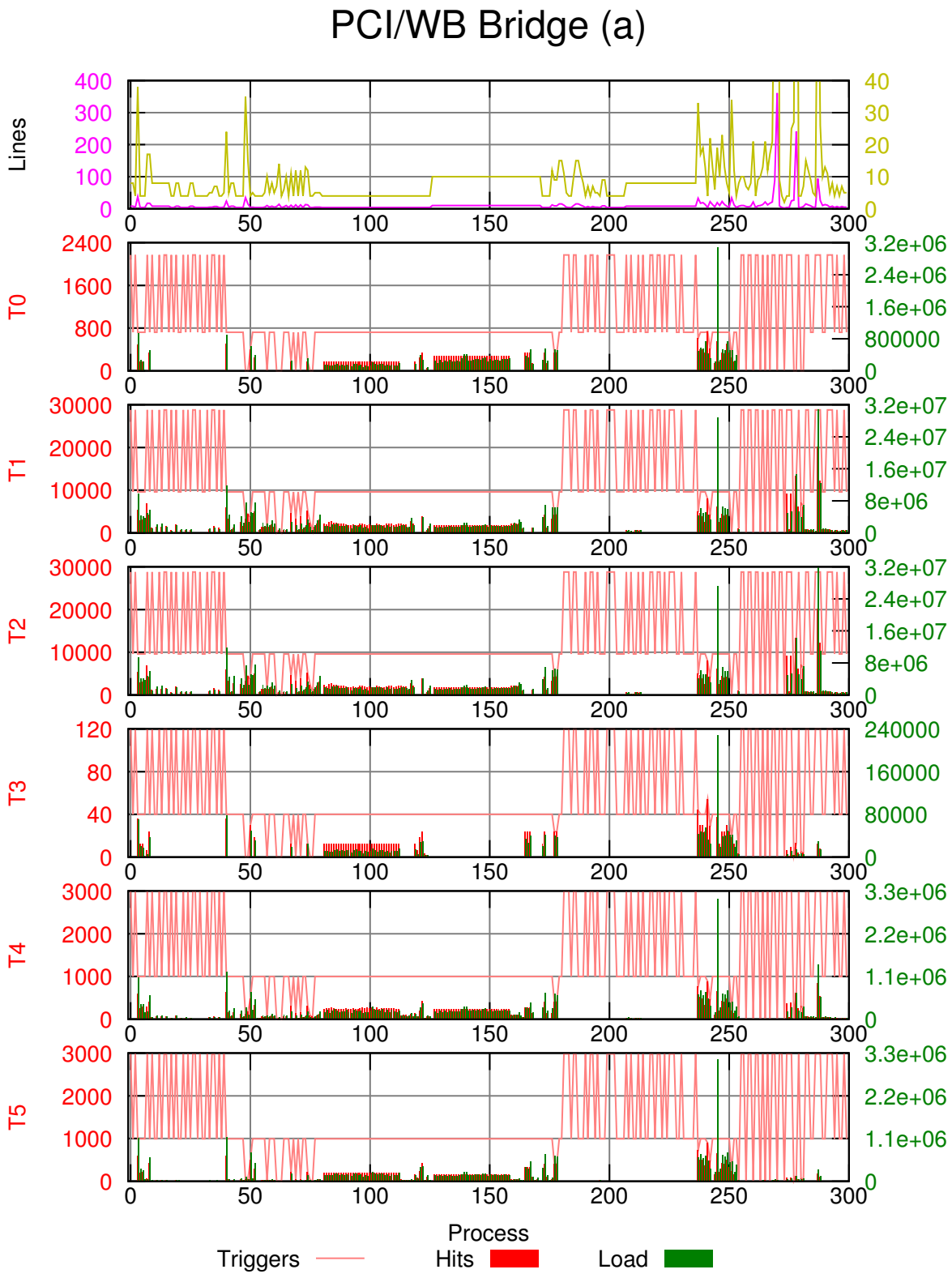


Figure 4.12. Process Characteristics of PCI/Wishbone Bridge (a)

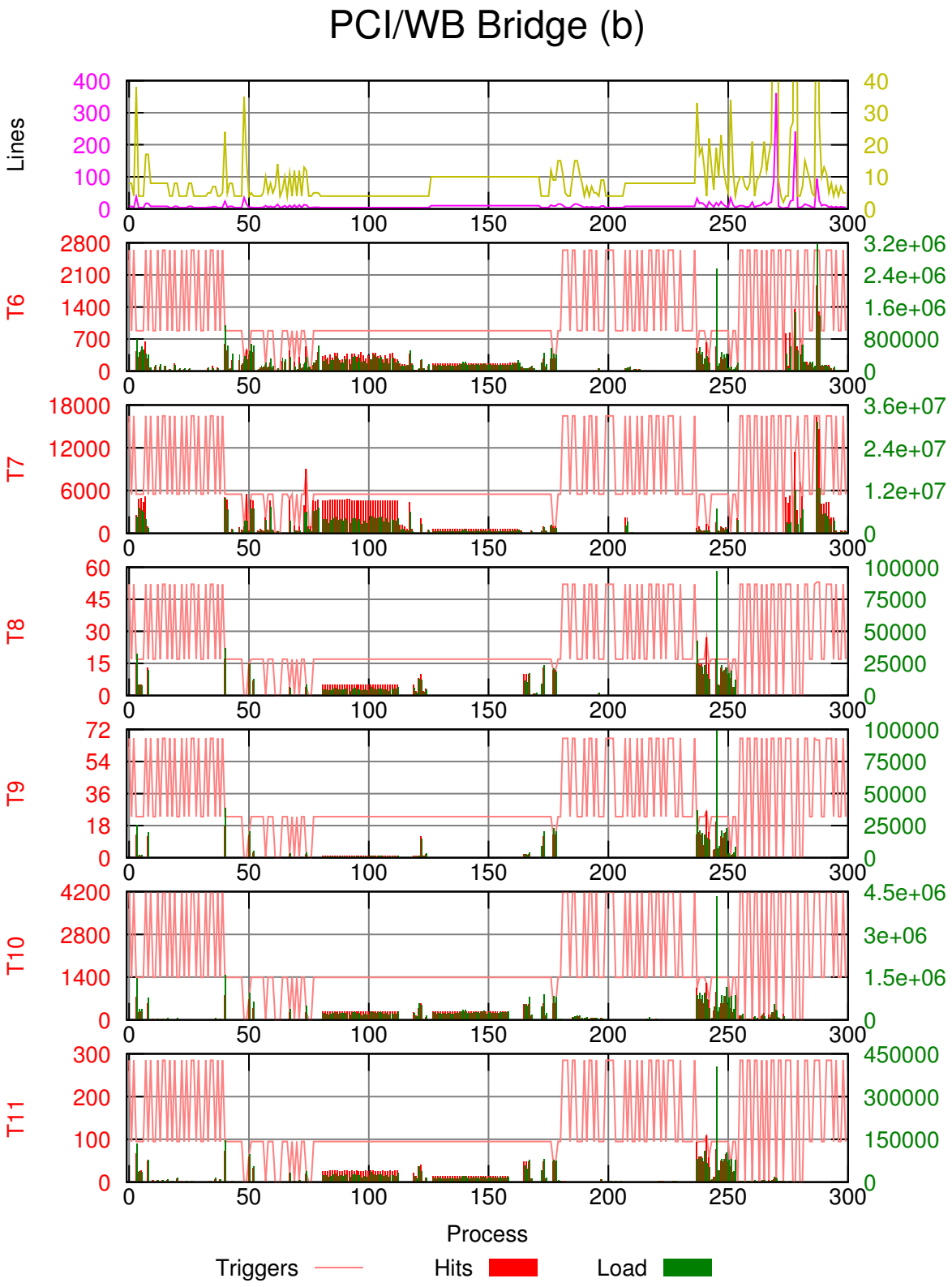


Figure 4.13. Process Characteristics of PCI/Wishbone Bridge (b)

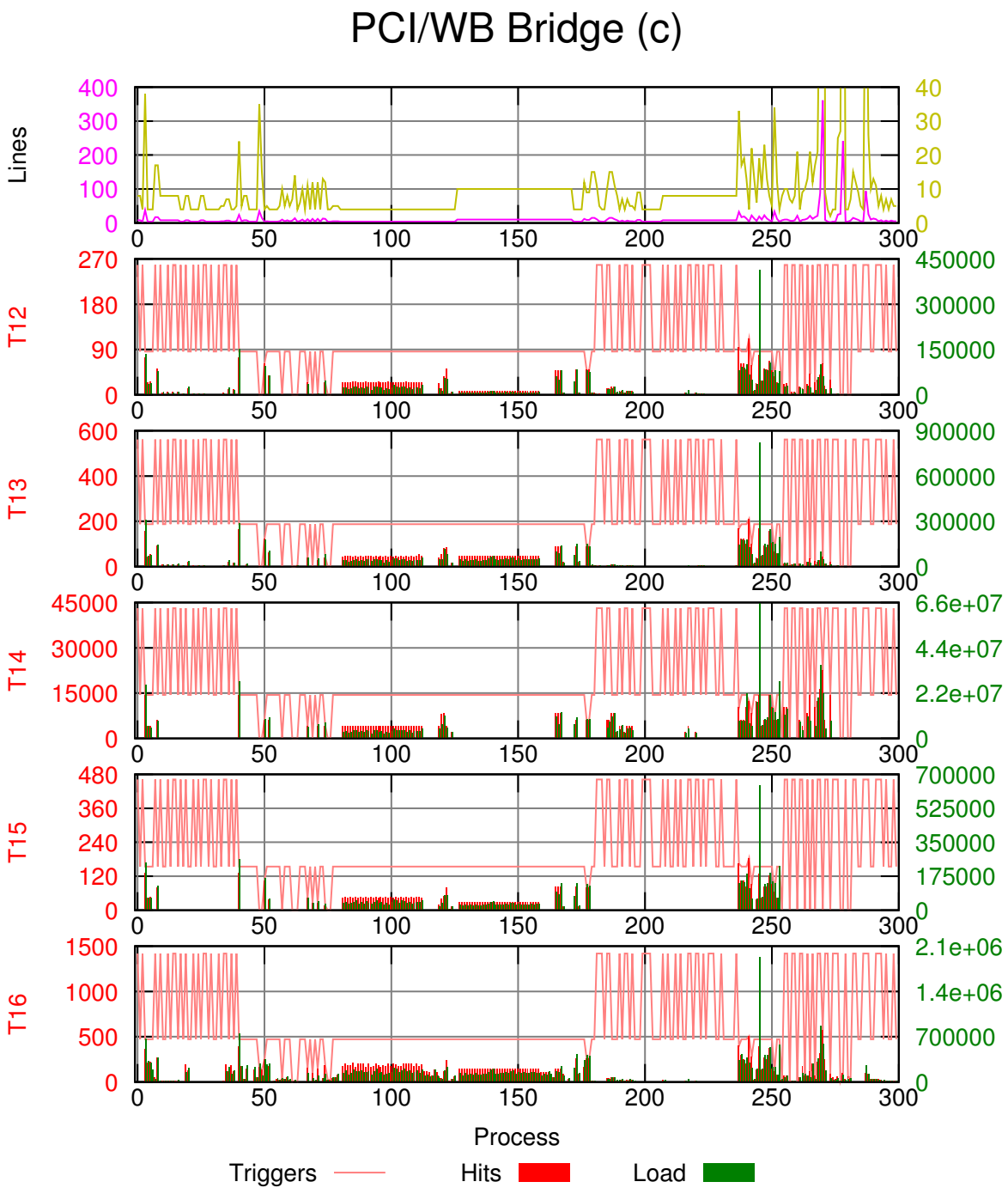


Figure 4.14. Process Characteristics of PCI/Wishbone Bridge (c)

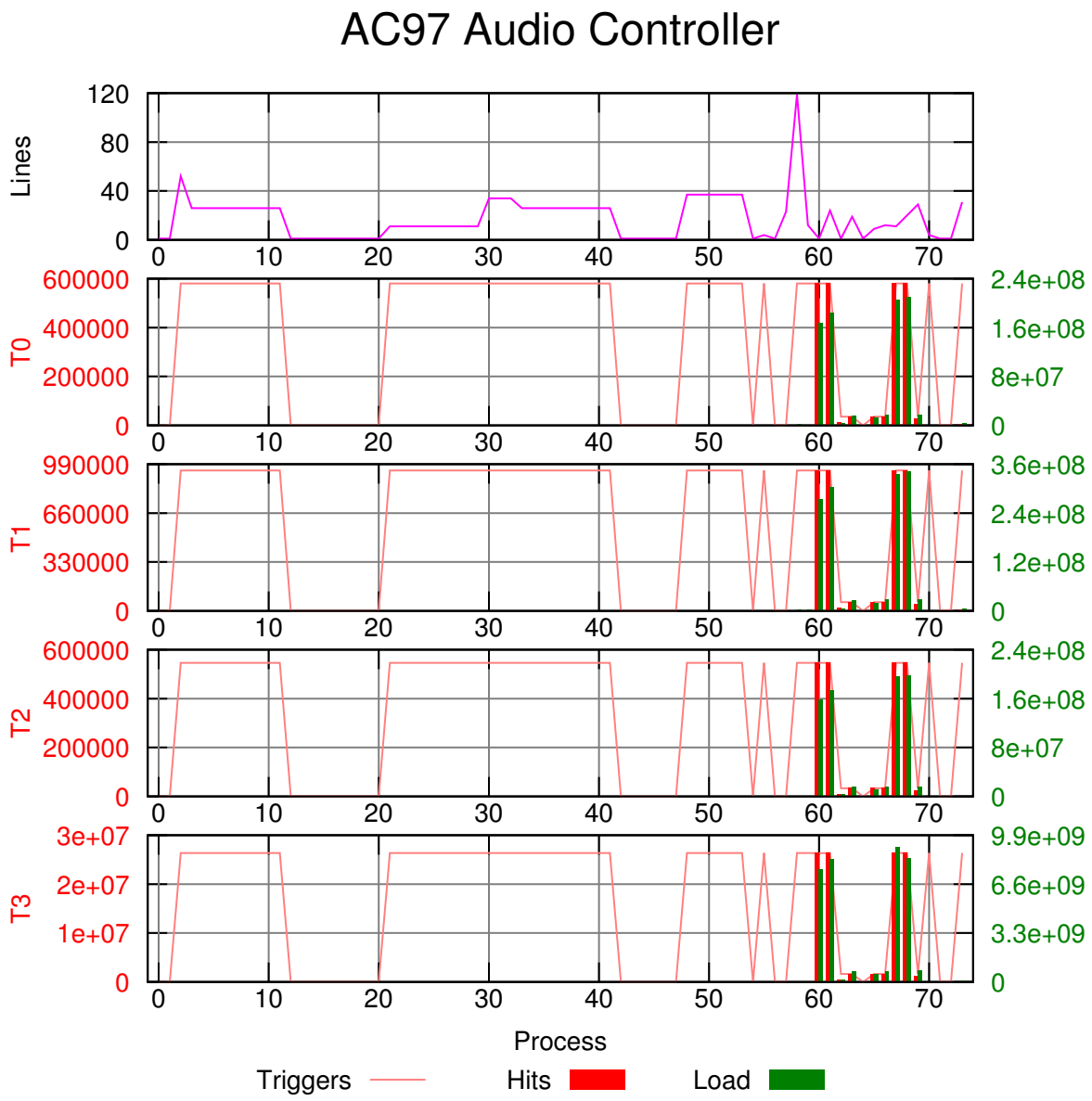


Figure 4.15. Process Characteristics of the AC97 Controller

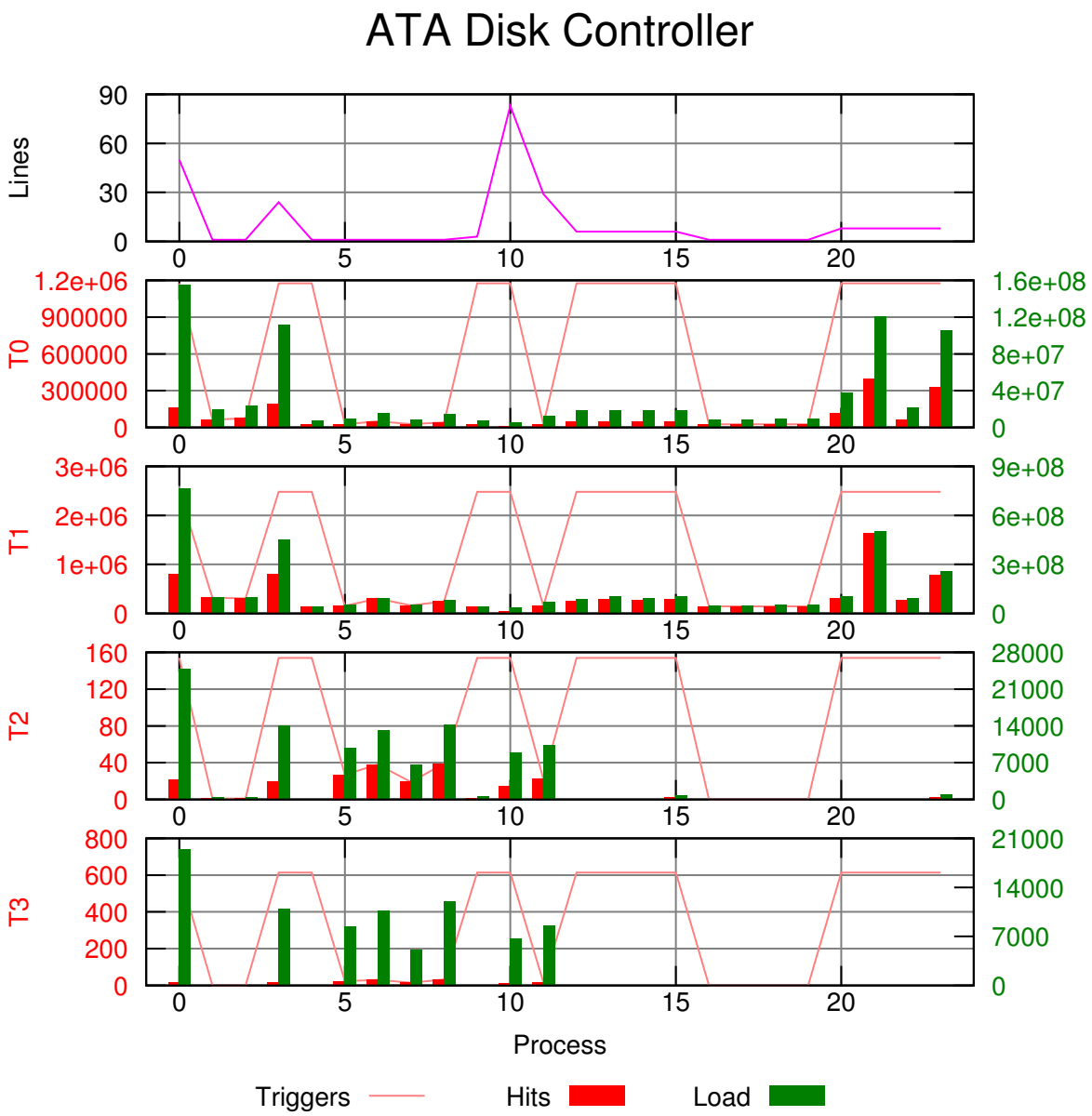


Figure 4.16. Process Characteristics of the ATA Controller

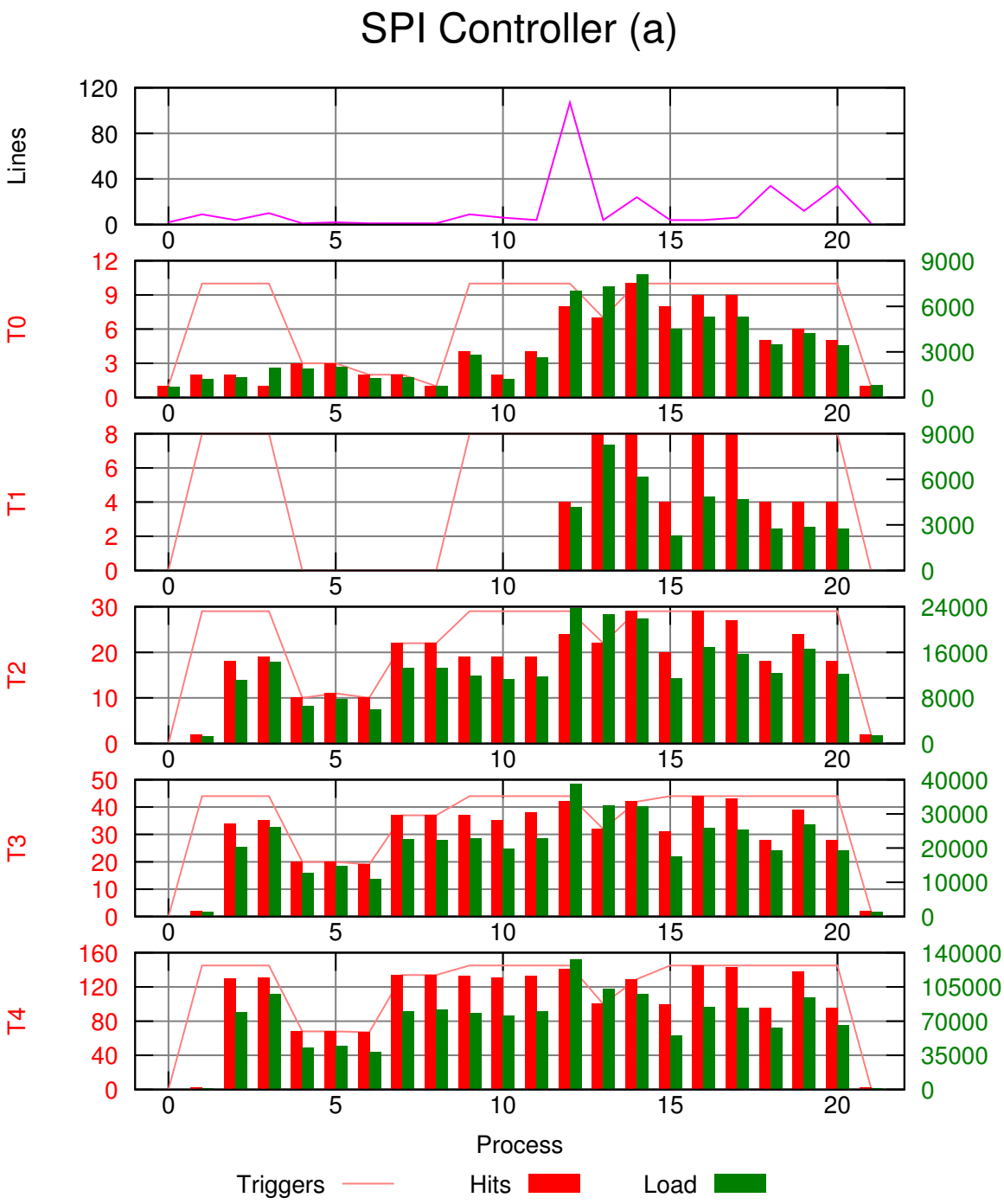


Figure 4.17. Process Characteristics of the SPI Controller (a)

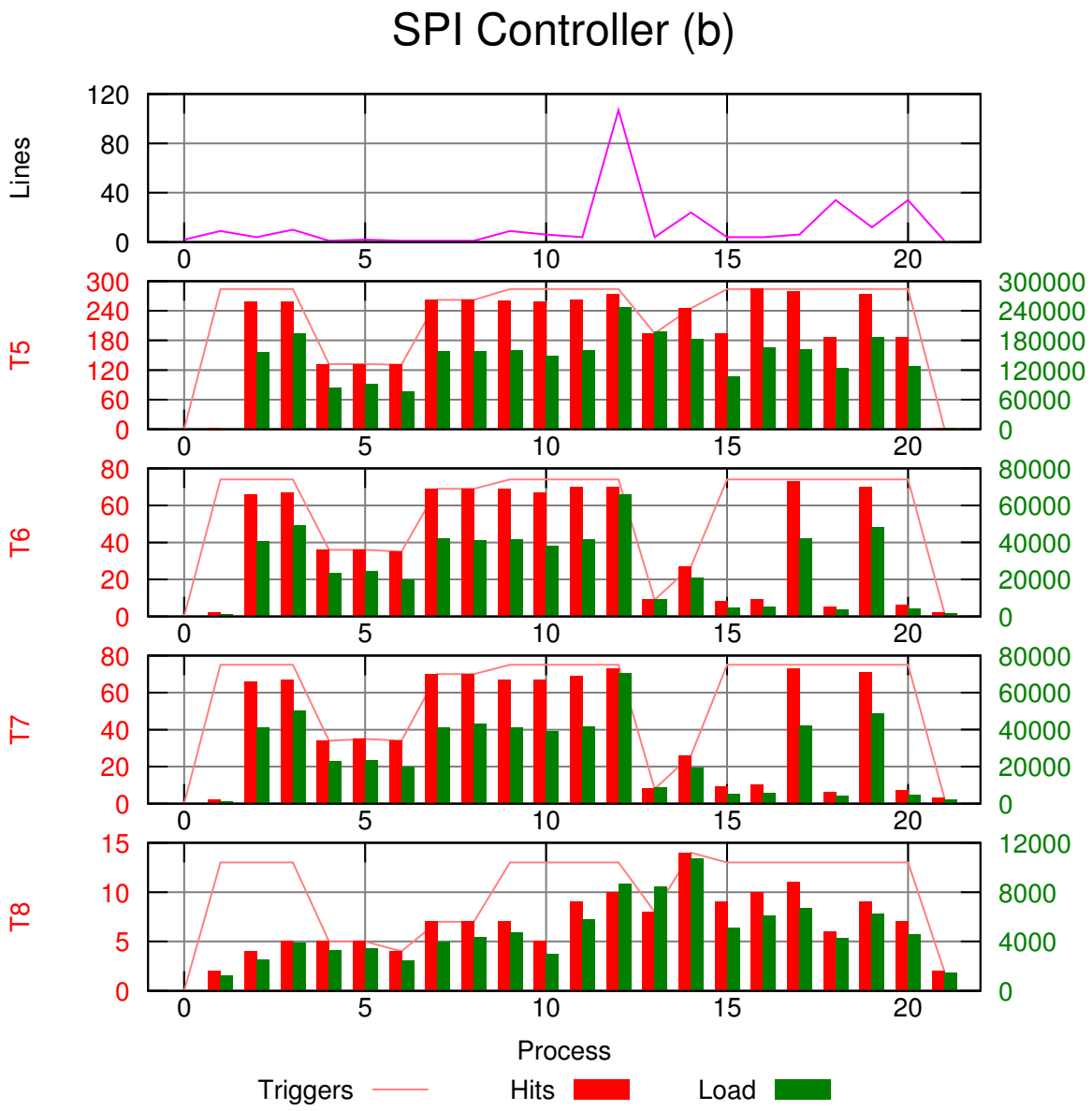


Figure 4.18. Process Characteristics of the SPI Controller (b)

Chapter 5

Implementation and Results

In Chapter 4 transformations of RTL code were presented to allow a simulation system to decompose a design into canonical processes. These canonical processes can be combined in any way when they share a common sensitivity list, which allows a migration system to identify locality of reference and balance the load across parallel execution units. In this chapter, an implementation of a migratory simulator is presented. The infrastructure required to implement a complete simulator would require several man-years of effort, but before such an investment should be made in both time and expense, the benefits and drawbacks of such a system must be understood. To this end, the simulation of a simple sort is presented which not only demonstrates the feasibility of a migratory simulator, but also provides the ability to measure system performance for empirical and algorithmic modeling. Modeling allows the impact of further developments to be evaluated.

The simulator platform is a Xilinx Virtex-II Pro FPGA, which contains a PowerPC processor, along with the basic bus infrastructure provided by the Embedded Development Kit (EDK). A Virtual Machine (VM) was coded, and an assembler was developed to convert assembly code into VM executable instructions. A simulator infrastructure was developed to allow these VMs to be connected together in a cycle-based method. The simulator is two-state to keep resource utilization to a minimum [74], which restricts the state of each bit to a `1` or a `0`, excluding resolved values such as `X` and `Z`. In addition to the software, pipelined and

non-pipelined implementations of an RM were made. An array of these Real Machines (RMs) was instantiated in the FPGA connected through a register-accessible mailbox. To facilitate run-time reconfiguration, a minimal Internal Configuration Access Port (ICAP) controller was developed that allows configuration frames to be read and written to the logic array.

Extraction code was developed to process Xilinx Design Language (XDL) and Logic Allocation files from the Xilinx tools in order to determine the location of every bit of state in the array of RMs. Using this information, cross referenced by both RM and configuration frame for efficiency, the simulator infrastructure can both extract and insert state. A frame caching system was also developed to allow single reads and writes of frames for multiple machine migrations to hide the large ICAP latencies.

The following items are necessary for a complete system, but are at present unimplemented:

1. A VHDL and Verilog compiler to produce VM code directly
2. Algorithms to route connections between RMs at run time
3. A JIT compiler for the VM code so it can be run natively on the PowerPC
4. Algorithms to monitor process activity to determine when and where to migrate the processes.

Items 2 and 3 are of particular import because they directly affect the performance at run time. It is imperative to understand the effect of these items on a migratory simulator, so these scenarios are modeled.

Section 5.1 describes the details of implementing the system, and Section 5.2 explains the empirical models developed for the system and the effect of various parameters on speedup.

5.1 Implementation Details

5.1.1 Platform Implementation

The migration system is implemented on an XUPV2P board developed by Xilinx for the Xilinx University Program [75]. The board is populated with a Virtex™-II Pro FPGA

Table 5.1. System Memory Map

Address	*Use*
0x00000000-0x0fffffff	DDR Memory 0
0x78030000-0x7803ffff	UART
0x78010000-0x7801ffff	LEDs
0x78000000-0x7800ffff	DIP Switches
0x78020000-0x7802ffff	Push Buttons
0x78080000-0x7808ffff	RM Mailbox
0xe0000000-0xefffffff	DDR Memory 1
0xffffc000-0xffffffff	Block RAM Boot Memory

(XC2VP30) and 512 megabytes of DDR memory. The XC2VP30 contains two PowerPC 405 processors along with 30,816 logic cells and 136 block RAMs. Using the Xilinx EDK version 7.1i, the design was instantiated with a DDR memory controller, the On-chip Peripheral Bus (OPB) infrastructure, the Internal Configuration Access Port (ICAP), an ICAP controller, and an array of RMs. A VM-based cycle simulator runs on the left PowerPC, equipped with code to migrate the state of the VMs to and from the RMs while seamlessly maintaining connectivity between the machines.

The Xilinx Platform Studio (XPS) manages the components to be instantiated into the FPGA. The PowerPCs are "hard cores," meaning that they are physically implemented into the die alongside the reconfigurable array. The XPS provides the means to attach "soft cores," such as the DDR controller, to the processors and import user designs in a convenient way to build a complete embedded system. A diagram of the complete system is shown in Figure 5.1, with the user- and Xilinx-provided components indicated. Each of the components on the buses is accessible with the memory map shown in Table 5.1.

5.1.2 Software Implementation

Chapter 4 discusses the simplification of RTL code into canonical processes which can be combined with other canonical processes having the same sensitivity to form larger processes, or they can be executed in parallel. When combinational processes are combined into their dependent sequential processes, the result is a cycle-based simulation in which the

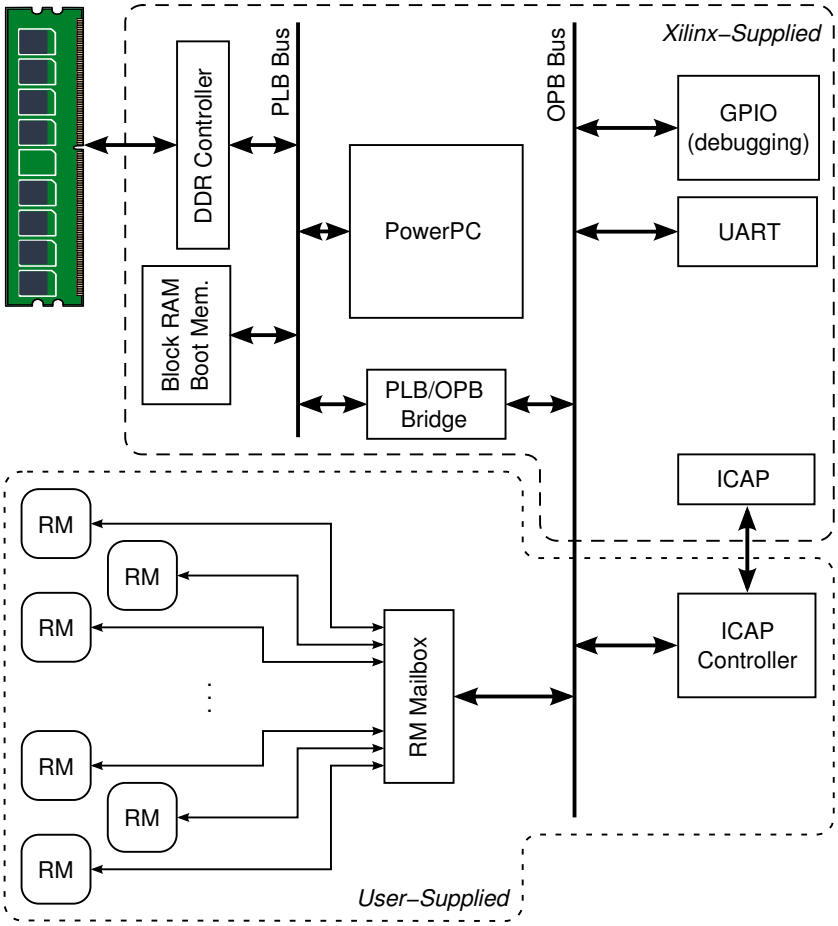


Figure 5.1. Block Diagram of the Implementation

processes need to be run only when the clock signal changes. A cycle-based simulator is significantly simplified compared to a general event-based simulator with a stratified event queue: Each simulation cycle is reduced to the two steps of 1) propagating the outputs to inputs and 2) the execution of the processes' code, corresponding to the *retrieve* and *clock* phases of a cycle-based simulation, respectively. (See Section 2.3.1 for further discussion.) With the addition of code for migration and performance analysis, the main simulation function of the VM/RM-based simulator is shown in Figure 5.2.

Xilinx provides several software platforms for use in the embedded systems produced by XPS, such as the Xilinx Microkernel, the Standalone Board Support Package (BSP), and Linux® [76]. The Standalone BSP was used in this implementation because of its simplicity. The simulator does not require preemption or interrupt handling, so a simple C program along with libraries for memory allocation and debugging I/O is sufficient. The software also utilized the libraries to access to the Time Base Counter (TBC) of the PowerPC core (`XTime_GetTime()`) to measure performance. This TBC returns a 64-bit counter value that increments every clock cycle. Since the PowerPC is run at 300 MHz, the time measurement resolution is 3.33 ns.

5.1.3 Virtual and Real Machines

The Virtual Machine (VM) infrastructure is implemented in an efficient way to reduce memory usage and to maximize performance, and a Real Machine (RM) is the hardware implementation of the VM. The internal structure of both the VMs and RMs is shown in Figure 5.3. The instructions and data are 16 bits in width, with separate memories for code, data, and registers (a Harvard architecture). The instruction set is RISC in nature with memory accesses only done with `load` and `store`. Appendix B contains the details about the instruction set. The instruction format is shown in Table B.1. The opcode field identifies the function of the instruction, and the remaining fields are utilized as shown in Table B.2. The assembler, recognizing the simple grammar in Listing B.1, converts instructions from a human-readable form to an executable form.

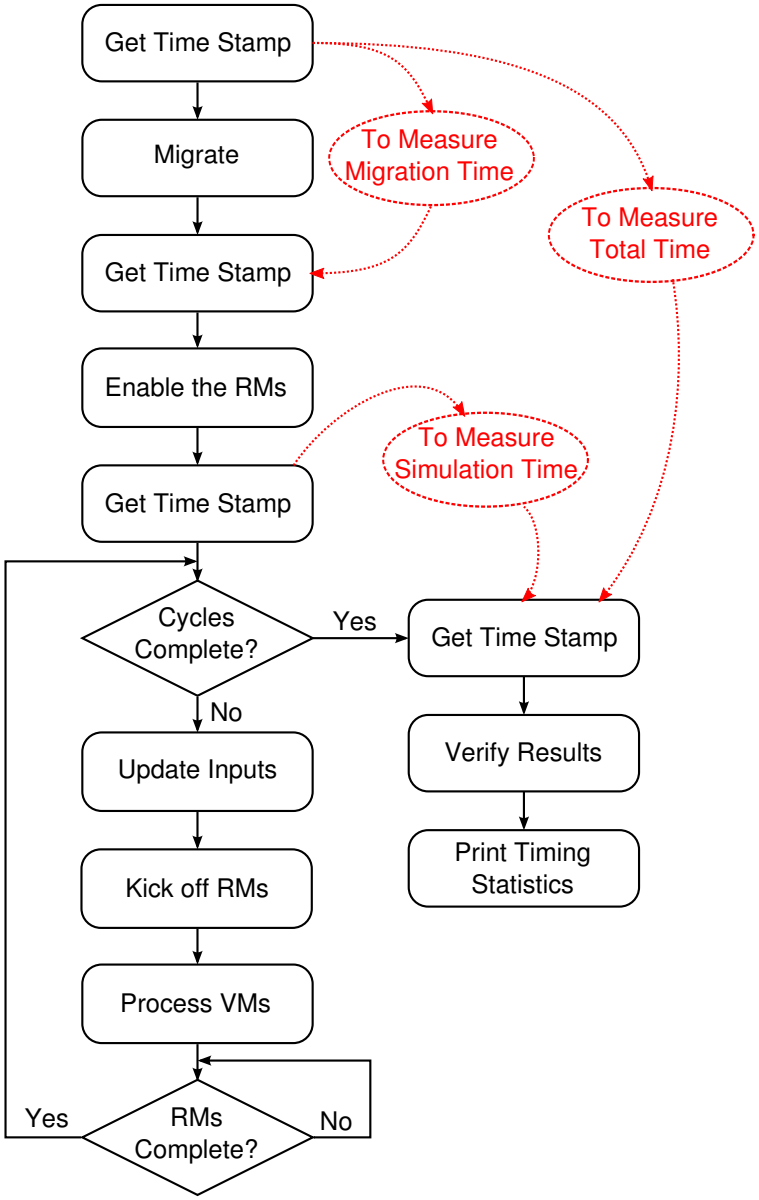


Figure 5.2. Flowchart of Main Simulation Loop

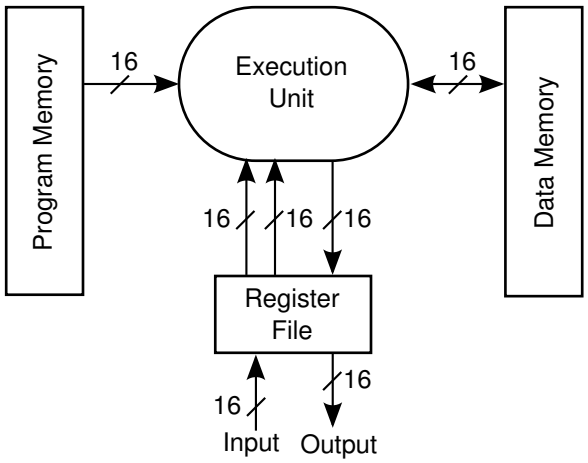


Figure 5.3. VM and RM Block Diagram

Migration transfers the state residing in the program memory, data memory, and register file between VMs and RMs using RTR. It is possible in a system with only RMs and VMs to migrate state without RTR, but there are two reasons for using it. First, one purpose of the implementation is to measure RTR-related overheads which cannot be known unless RTR is used. Second, the additional logic required to migrate state without RTR would increase the size of the infrastructure, reducing the number of RMs within the FPGA and making timing closure more difficult.

When considering the RM architecture, a common approach to processor implementation is an execution pipeline with a register file that provides two reads and one write per clock cycle. This single-in-double-out interface allows two registers to be read and a result to be written every cycle, and consequently an instruction can be completed every cycle. With a fully pipelined RM architecture complete with data hazard avoidance, the XC2VP30 FPGA holds 24 RMs. This architecture, however, needed revision due to limitations in the FPGA. RTR is a developing technology having much potential, but in practice it is not trivial to employ. Xilinx tools provide limited support, but there are some significant architectural restrictions in the FPGA itself. One restriction is the behavior of the `GRESTORE` configuration command [20], issued through the ICAP controller to modify the state of the register file flip-flops to migrate process state. In this RM application, state must be migrated on a *per-machine* basis. `GRESTORE`, however, updates all the flip-flops in the entire logic fabric, including those

flip-flops in the infrastructure logic and the ICAP controller itself. Therefore, migratable state cannot be stored in flip-flops.

A solution is to use Look-Up Table (LUT) distributed RAM rather than flip-flops for the register file. LUT-based RAMs can be read and written through the ICAP on a per-machine basis without changing the state of vital infrastructure. However, they cannot provide the single-in-double-out interface required by full pipelining. The Xilinx core libraries provide dual read port RAMs, but one of the read addresses is shared with the write port. Rather than doubling the RAM, the solution chosen is to execute an instruction every two cycles in a non-pipelined fashion. As will be seen, reducing the performance of the RMs by half does not affect the overall performance of the simulation application, and the elimination of data hazards makes the RMs smaller.

There is another limitation of LUT-based RAMs, however, that must be addressed. A frame is the minimum configuration unit within the FPGA. Within Virtex-II FPGAs, frames span the entire height of the device, and some of the logic used to load the configuration frames is reused by LUT RAMs during normal operation. Hence, no LUT RAMs within a frame can be read or written while the frame is being configured. This poses a problem with the infrastructure logic such as the DDR memory controller and the OPB bus interfaces, which use LUT RAMs in FIFOs. Merely reading the frames utilized by this infrastructure while running causes the system to malfunction. This issue does not affect RMs, however, because migration to and from them only occurs between simulation cycles when the RMs (and their LUT RAMs) are idle. Careful floorplanning is required to ensure that the RTR of RMs does not access any configuration frames of the infrastructure. To this end, each RM is constrained to occupy a 16x16 slice area, and they are floorplanned into columns. The remaining frames, which are never read or written, are reserved for infrastructure. Without protecting the infrastructure, 40 RMs fit into the device with a 92% slice utilization. After protective floorplanning, 35 RMs fit with an 82% slice utilization. The resource utilization, including the unusable area due to protective floorplanning, is shown in Figure 5.4.

Figure 5.5 shows the state machine of the non-pipelined RM. The machine begins in the `IDLE` state waiting for a `go` signal from the simulator. This is the only state in which the machines can be reconfigured, since there are no accesses to LUT RAMs in this state. The

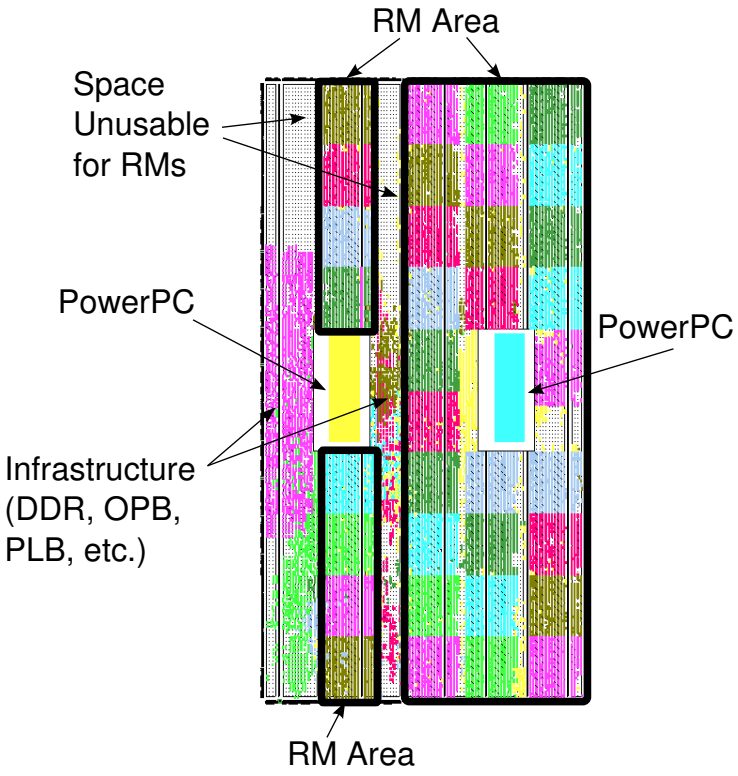


Figure 5.4. Device Utilization of the Migration System

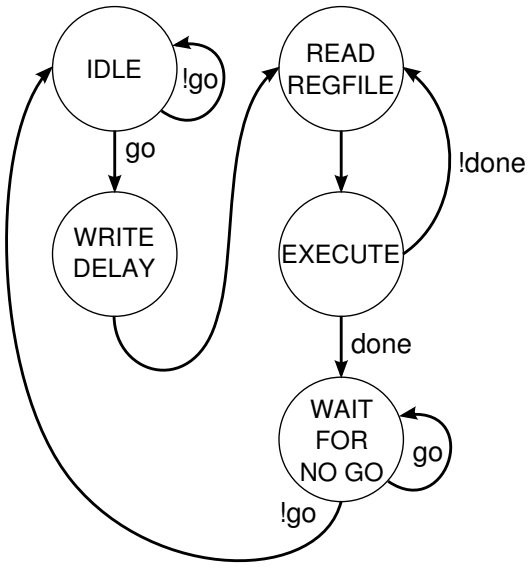


Figure 5.5. The States of the RM Processor

LUT RAMs are loaded through the ICAP with the code and data, including the initial output values for the process. While still in this state, an `update` signal causes the RM to read the initial output value from the register file and store it in the RM's output flip-flops. When the `go` signal is received—marking the beginning of the simulation cycle—the input value of the RM is written to LUT memory during the `WRITE_DELAY` state. These steps must be taken to propagate the input and output values of the RMs to and from the LUT RAMs because of the global behavior of `GRESTORE`. In `READ_REGFILE`, two registers are read from the register file LUT RAM or one entry from the data RAM, and during the `EXECUTE` state, the result is computed and written back. The program counter is also advanced. The states alternate between `READ_REGFILE` and `EXECUTE` until the `done` instruction is encountered, at which time the RM asserts the `done` signal and waits for `go` to be deasserted. The deassertion of `go` marks the end of a simulation cycle.

5.1.4 User-Provided Hardware Components

Section 5.1.1 mentioned that the XPS provides the means to instantiate modules supplied with the Xilinx EDK, but several parts of the system, in addition to the RMs, needed to be supplied for a working system. This user-supplied infrastructure is presented here.

The ICAP and ICAP Controller

The Internal Configuration Access Port (ICAP) is the device through which the entire configuration of the FPGA is available to the logic within the device itself. As discussed in Chapter 2, Section 2.1, an SRAM-based FPGA may be reconfigured many times to implement the desired functionality, and the minimum configuration unit is the *frame*. Frame data are assembled into a *bitstream*, with one or more contiguous frames' data preceded by a header which contains a frame address. The ICAP processes the header and routes the data to the correct frame. The ICAP also processes commands such as the reading and writing of configuration registers and capturing flip-flop state [20].

As the ICAP processes packets, it goes through a series of states, requiring an ICAP controller to know when to read and write each byte. An ICAP controller was implemented to

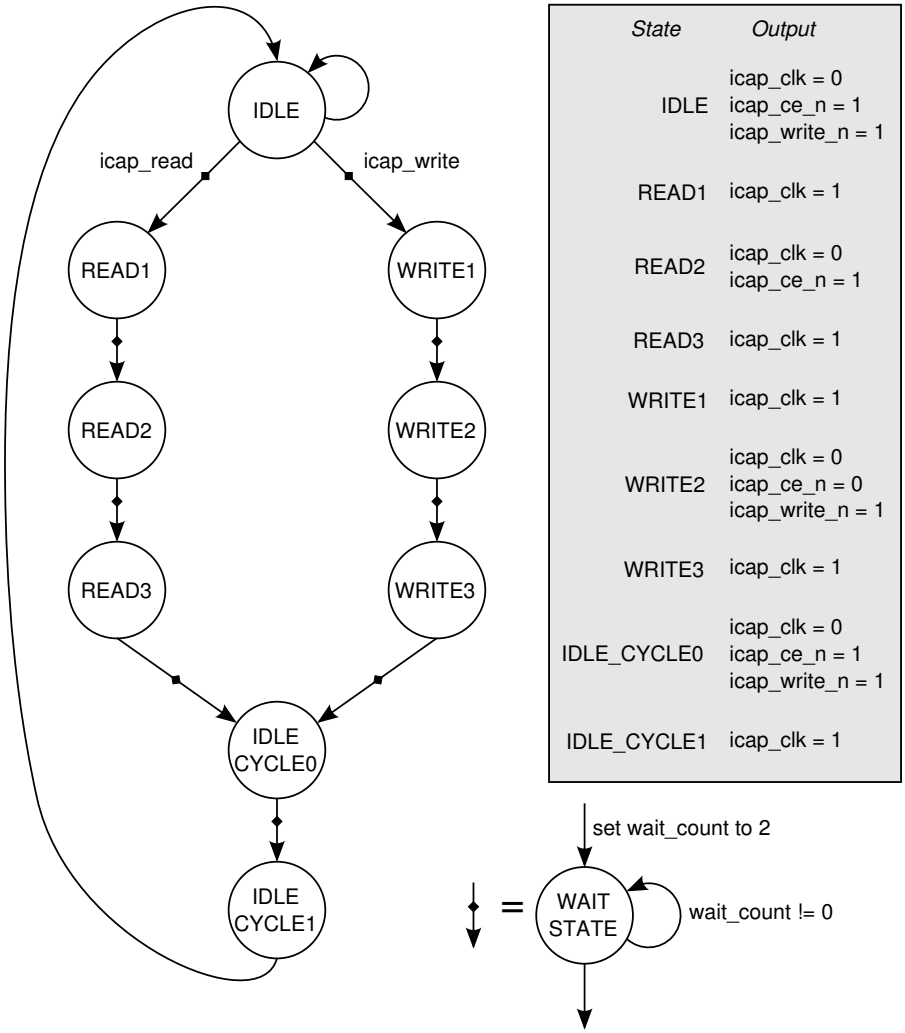


Figure 5.6. ICAP Controller States

perform the minimum number of states shown in Figure 6 on page 8 of [77] that are required for reading and writing state. An examination of that figure shows that the ICAP may be placed into a series of abort states, but the implementation of the simulator does not require them. The controller must also have wait states between the main states of the machine to supply the proper clock rate to the ICAP. There is some ambiguity in the Xilinx documentation concerning the maximum clock rate of the ICAP. It is limited to 33 MHz in application note [78], but the documentation of SelectMAP, which uses the same logic as ICAP, has a maximum rate of 50 MHz [79]. 33 MHz was assumed in this design, and the ICAP controller runs at 100 MHz. Therefore, the ICAP controller implements the states shown in Figure 5.6 with compile-time-specified wait states for ICAP clocking.

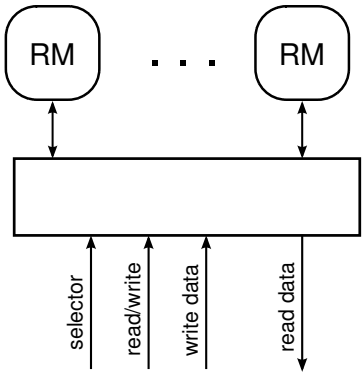


Figure 5.7. RM Mailbox

RM I/O Mailbox

A complete simulation system that utilizes RTR to its fullest would implement dynamic routing between RMs as processes are migrated to them. Run-time routing in an FPGA is being researched by others [80, 81], so an implementation is left to future work. For the implementation to date, the RMs are connected in some test scenarios through the register mailbox shown in Figure 5.7. Using a mailbox limits the amount of address multiplexing on the OPB Bus (i.e., reduces the number of OPB slave registers required) and instead places the multiplexing at the outputs of the RMs. Initial synthesis attempts with a large number of slave registers on the OPB failed to meet the bus timing. The mailbox allows the software to select an RM to read by writing an address in one register and reading the data from another. Writes are completed with a write to a data register followed by a write to the RM address register. During a simulation, the simulator updates the inputs and reads the outputs of the RMs in this fashion between every simulation cycle, and this is referred to as *software connectivity*. For some of the tests, the RMs were connected directly to one another in hardware, and this is referred to as *hardware connectivity*.

5.2 Results

As mentioned at the beginning of this chapter, the purpose of the implementation is to measure migration-related performance for use in empirical modeling. To fulfill that goal, a simple parallel sort was run in the implementation described in Section 5.1. The parallel sort is

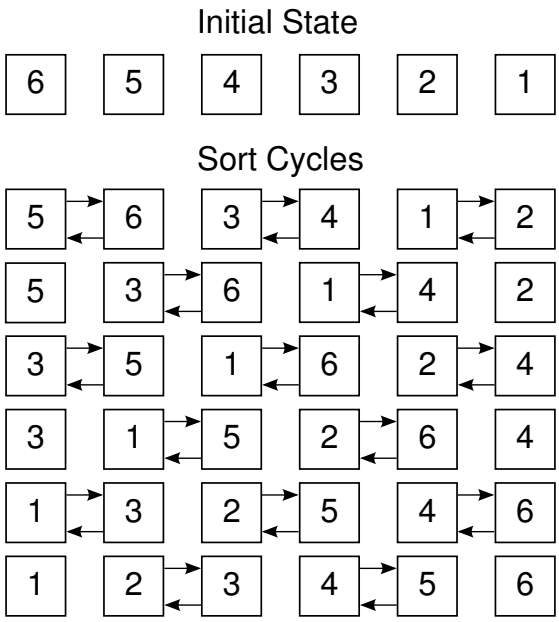


Figure 5.8. Cycles of the Even-Odd Transposition Sort

a good application for this purpose for two reasons. 1) Flexibility: The number of processes and the number of simulation cycles can be changed without altering the outcome of the algorithm. 2) Uniformity: Every process runs identical code providing the uniformity required for accurate and consistent analysis. If each process ran different code, the time measurements of migration and execution would not be consistent while varying processes and simulation cycles.

5.2.1 The Application and Measured Results

The even-odd transposition (EOT) sort is the algorithm used [82]. Each iteration of the sort is a swapping—or transposition—of the numbers to place the greater number on the right, as illustrated in Figure 5.8. When executed in parallel, it requires a maximum of n simulation cycles to sort n numbers when there are $\frac{n}{2}$ or $\frac{n}{2}-1$ iterations per simulation cycle. The sort was implemented in VHDL as an array of identical processes, each comparing its own output with its left or right neighbor's output on alternating simulation cycles. In an RTL simulator, this code would be compiled into one process for each instance (barring optimization), each running the same code.

```
signal even : boolean := ((INDEX mod 2) = 0);
signal myval: std_logic_vector(o_out'range) :=
  conv_std_logic_vector(INITIAL_VALUE,
  o_out'length);
. . .
o_out <= myval;
. . .
main: process(i_sysclk)
  variable myval_v:
    std_logic_vector(myval'range);
begin
  if(i_sysclk'event and i_sysclk = '1') then
    even <= not even;
    myval_v := myval;
    if(not even) then
      if(myval_v < i_left) then
        myval_v := i_left;
      end if;
    else
      if(myval_v > i_right) then
        myval_v := i_right;
      end if;
    end if;
    myval <= myval_v;
  end if;
end process;
```

Figure 5.9. VHDL Code of the Even-Odd Transposition Sorter

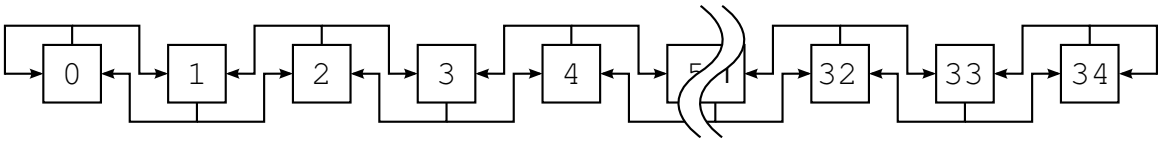


Figure 5.10. Logical Connection of VMs for EOT

The main VHDL process of the sorter along with some ancillary code is shown in Figure 5.9. The 8-bit inputs are `i_left` and `i_right`, and the 8-bit output is `o_out`. Each instantiation is assigned an `INDEX` generic equal to its instance number. The `even` signal is initialized to true or false depending on whether the `INDEX` is even or odd. The `INITIAL_VALUE` is a generic assigned during instantiation. Any practical sorting implementation would need to be programmable so that software can load the array to be sorted by hardware. Currently, the initial value is merely a place-holder which is assigned during run time in the VMs. Any number of these processes can be instantiated, with those on the end connected to themselves. (See Figure 5.10.)

For the proof of concept, the VHDL code was translated manually into the CIS in Figure 5.11, which is then assembled into executable VM/RM instructions documented pre-

```
 1 ; Even-Odd Transposition Sort
 2 .input leftin  7:0, r1. 7:0
 3 .input rightin 7:0, r1.15:8
 4 .output out 7:0, r0.7:0
 5 .data 0x0  ;even gets 1, odd gets 0
 6 .data 0x1
 7 .data 0xff ;Used for AND mask
 8
 9 load r4, [0]
10 load r5, [1]
11 load r3, [2]
12 xor r4, r4, r5
13 store r4, [0]
14 cmp r4, r5
15 bnz 0xc
16
17 ; Handle the number from the left
18 and r2, r1, r3
19 cmp r2, r0
20 bc 0x17
21 move r0, r2
22 done
23
24 ; Handle the number from the right
25 shiftr r2, r1, 1
26 shiftr r2, r2, 1
27 shiftr r2, r2, 1
28 shiftr r2, r2, 1
29 shiftr r2, r2, 1
30 shiftr r2, r2, 1
31 shiftr r2, r2, 1
32 shiftr r2, r2, 1
33 cmp r0, r2
34 bc 0x17
35 move r0, r2
36 done
```

Figure 5.11. VM/RM Assembly Code

viously in Section 5.1.3 on page 95. Each RM has eight 16-bit registers with some of them aliased to inputs and outputs. In this CIS code, `.input` and `.output` map the I/Os to `r1` and `r0`, respectively.

Each machine includes two 32x16 memories for data and code instantiated as LUT RAMs. An alternative for larger processes is to replace either or both memories with larger block RAMs, but the corresponding increase in state may require longer migration times. The `.data` directives initialize the data memory (beginning at location 0) with constants used by the algorithm: location 0 corresponds to the `even` signal in the VHDL code, and the remaining two are used for toggling and masking. Since barrel shifters require significant resources, the RM supports only a shift by one, and lines 25 through 32 implement a shift by eight. As mentioned in Chapter 1 on page 3, instantiating processor subsets whenever possible would reduce area. In this example, there is enough code memory for eight shifts in an RM instead of a single shift-by-eight, so it was elected to reduce area rather than execution time. The best option for this code would be to instantiate an RM using a Dynamic Module Library [81], that supports only a shift by eight.

The FPGA holds 35 RMs, so the EOT sort was run with 35 processes. Each process is instantiated in a VM connected to its left and right neighbors and is assigned an initial random value. These values are sorted in no more than 35 simulation cycles, but longer runs with correct results are also run for performance measurements.

To evaluate pure parallel performance, the speedups with no migration are measured for both software and hardware connectivity. Figure 5.12 shows the resulting plots. Because hardware connectivity is currently manual, its performance was measured with 0, 24, 30, 34, and 35 RMs, and Marquardt-Levenberg curve fitting provides the other points. The hardware connectivity case is just as deterministic as the software connectivity case, so no anomalies that may be missed by the curve fitting are expected. The speedup is the run time without RMs (T_0) divided by the run time with RMs (T_r) [10]. One result of executing code in VMs without native JIT compilation is a "super-linear" speedup because the RMs execute their processes faster than the VMs. Not shown in the plots due to scaling disparity is the speedup of 1005 when using hardware connectivity with 35 RMs. To illustrate superlinearity, consider a serial program that executes 10 tasks, each taking 1 s. If those tasks are parallelized on 10 processors, they all

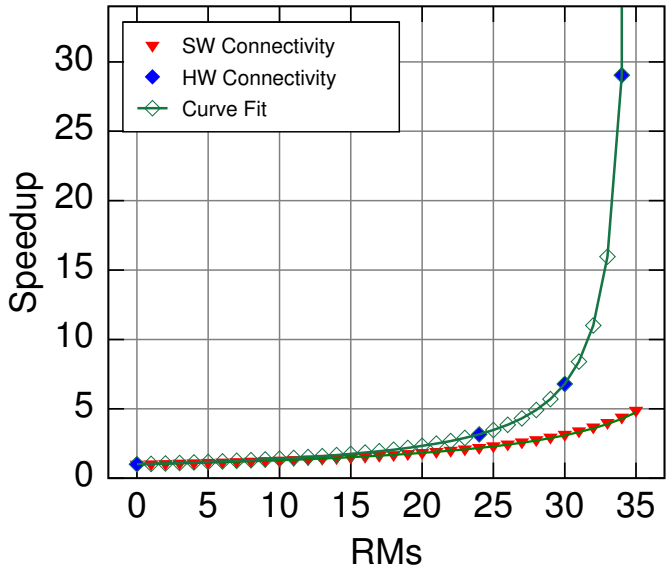


Figure 5.12. Speedup of HW and SW connectivity (35 simulation cycles)

execute in 1 s for a speedup of 10. If, however, the tasks are executed on parallel processors that are ten times faster, they complete in 0.1 s for a speedup of 100.

Figure 5.12 shows clearly that hardware connectivity is a necessary improvement. These direct connections could be accomplished with a reconfigurable crossbar switch, run-time routing, or a network-on-chip. Run-time routing and the use of a network-on-chip are explored later in Section 5.2.2 on page 110 and Section 5.3.1 on page 125, respectively. Note that the relatively low speedups on the left side of the plots demonstrate Amdahl's Law, a behavior exhibited by all parallel systems, and is not unique to this system. Even with existing simulation accelerators, Amdahl's Law implies that if 50% of the simulation is serial, then the maximum speedup achievable by parallelization is two. Also, as discussed in Section 2.5 on page 24, a cross section of Amdahl's Law plots is achieved when the parallel part of the problem varies with the number of parallel processors, and these plots demonstrate that behavior.

The time required to execute the sorting simulation is

$$T(r) = C(t_o + rt_r + [P - r]t_v) + rMt_m, \tag{5.1}$$

where t_o, t_r, etc. are described in Table 5.2. The equation above does not include t_R, the

Table 5.2. Modeling Parameters

Param.	Description	Value
C	Simulation cycles	*varies*
P	Processes	*varies*
r	Number of RMs	*varies*
M	Number of Migrations	*varies*
t_v	VM Execution Time	83.13 $\frac{\mu s}{\text{simcycle}}$
t_r	RM Execution Time	17.03 $\frac{\mu s}{\text{simcycle}}$
t_o	Simulator Overhead	2.96 $\frac{\mu s}{\text{simcycle}}$
t_m	Migration Overhead	6.02 $\frac{\text{ms}}{\text{migration}}$
t_R	Run-time Routing Overhead	*varies*

run-time routing time per migration, which is used in the empirical model discussed later. The sort tests use $C = P = 35$, while r varies from 0 to 35. t_r includes any RM-related overhead (including the software connectivity accesses) plus any time spent waiting for the RMs to complete. The CIS code consists of 24 instructions, but not all are executed each simulation cycle due to branching. A pessimistic estimate of 50 instructions per simulation cycle gives an RM execution time of 1 μs. Since the RMs are notified to execute a simulation cycle just before the PowerPC executes the VMs, the RMs are expected to be finished before the VMs.

Next, consider the effect of migration overhead on performance. Figure 5.13 shows the performance of the simulation including all migration overheads. Note two things. First, the blue plot of Figure 5.13 corresponds to the red plot of Figure 5.12. The migration overhead, which is greater than the entire simulation time, causes a speedup of less than one. Second, the red and green plots of Figure 5.13 show that the speedup can be improved for longer simulation times. In these cases, the migration time is amortized over a longer simulation time, a property described by the Simulation-cycle-to-Migration Ratio (SMR). The larger the SMR, the less impact the migration times have on the simulation, improving the speedup. Appendix D shows statistics of the actual tests run on the OpenCores code discussed in Chapter 4. The "Max. Hits" columns shown in the tables indicate that an SMR of 10240:1 is not unreasonable for many of the tests, some requiring millions of simulation cycles. For small SMRs (e.g., 35:1),

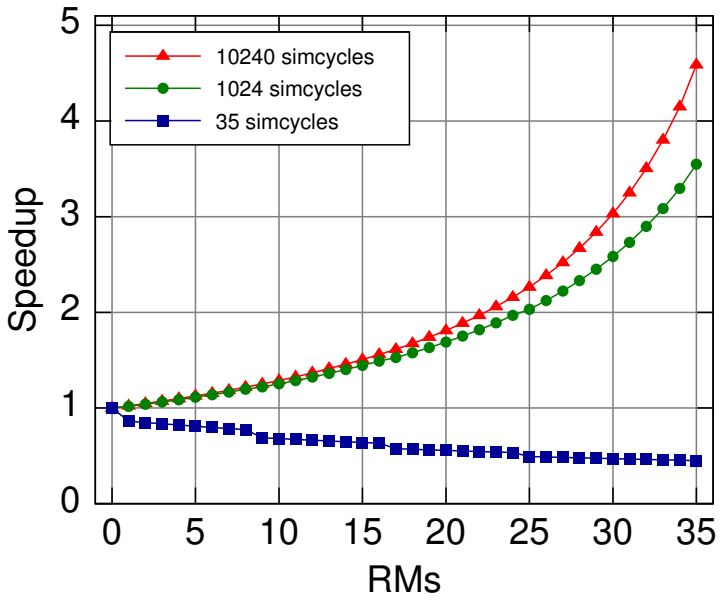


Figure 5.13. Actual Software Connectivity Speedups

the migration time may exceed the simulation run time, and the system exhibits a speedup less than one.

Migration overhead is affected by FPGA reconfiguration time, and the Virtex-II architecture does not lend itself well to quick reconfiguration. Each frame spans the entire height of the FPGA, thus incurring large overheads for the migration of relatively small amounts of state. Additionally, every read and write of configuration frames requires a "dummy" frame to be read or written, and the ICAP has only an 8-bit interface run at a lower clock rate [78, p. 4]. Multiple frames must be transferred to migrate state, and the unoptimized migration time per machine was measured to exceed 220 ms. However, the full-height span of frames can be exploited. The RMs are floorplanned in columns of 8, except for the second column which also contains RMs 32 and 33, and the fourth column with also contains RM 34. For each RM, an RLOC constraint places the program, data, and register file memories into the same frames as detailed in Table B.3 in Appendix B. This optimization reduces the number of frames containing memories from 65 to 16. The astute reader will notice that Table B.3 only shows 12 columns which would correspond to 12 frames rather than 16. As mentioned in Section 5.1.3 on page 98, LUT RAMs are used for the register files which are dual ported. Because they are dual ported, they utilize two adjacent frames although the placement is specified as one

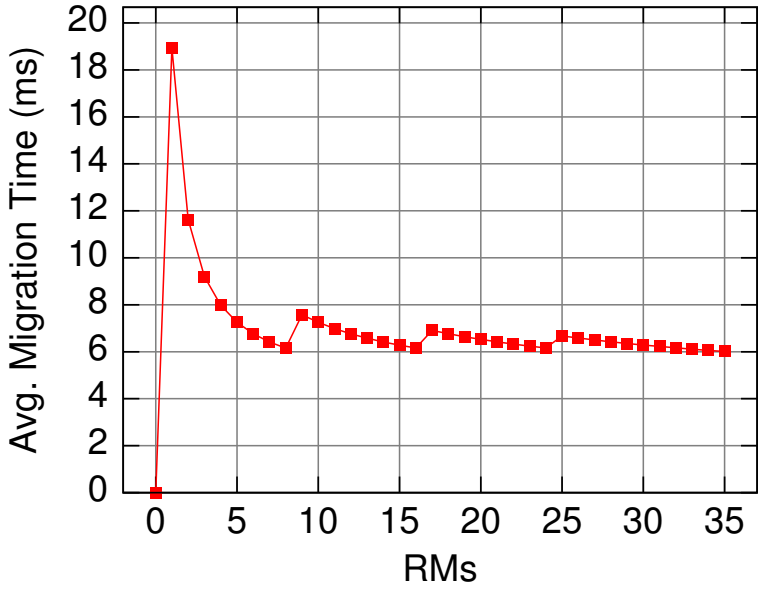


Figure 5.14. The Effect of Frame Caching on Migration Times

column. Sharing frames among RMs also allows frame caching, which minimizes ICAP accesses by reading and writing a shared frame only once during a series of migrations. These optimizations reduce the average migration time from 220 ms to 6.0 ms, and the maximum for one RM is 18.9 ms. The measured migration times are shown in Figure 5.14. The last 11 RMs do not have an irregularity after 8 RMs because RMs 32–34 are updated with no additional overheads due to frame caching.

5.2.2 Empirical Modeling

Speedup is essentially the run time without parallelization divided by the run time with parallelization. Mathematically, speedup $S = T_0/T(r)$, which can be modeled as follows for hardware connectivity, where T_0 is the measured time without RMs:

$$S = \begin{cases} \dfrac{T_0}{C(t_o + Pt_v)}, \text{ for } r = 0, \\ \dfrac{T_0}{C(t_o + t_r + [P-r]t_v) + rMt_m}, \text{ for } 0 < r < P, \\ \dfrac{T_0}{Ct_o + rMt_m}, \text{ for } r = P. \end{cases} \tag{5.2}$$

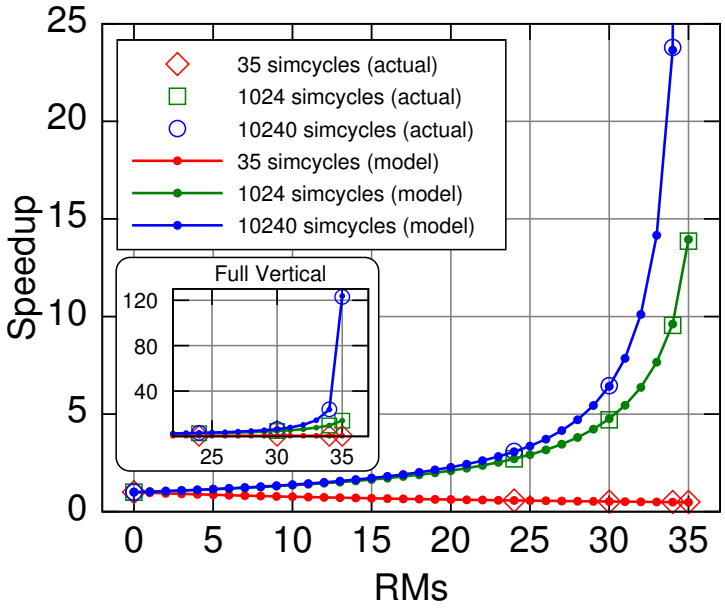


Figure 5.15. HW Connectivity Speedups Compared to the Empirical Model

This piecewise equation results from the fact that when zero RMs are in use, there are no RM overheads t_r and t_m. For the middle region, there is a single t_r for the single RM that has hardware connectivity on one side and software connectivity on the other. The remaining RMs have only hardware connectivity. For the final case, nothing is run in VMs, so there is only the simulator per-cycle overhead (t_o) and migration overhead (t_m).

As mentioned previously, the EOT sort gives correct results for runs longer than 35 simulation cycles, and Figure 5.15 shows speedup plots for 35, 1024, and 10240 simulation cycles, including all migration overheads for one migration per RM ($M = 1$). Using these measured values with Equation 5.2, the Marquardt-Levenberg curve fitting algorithm solves for t_r, t_v, and t_m as shown in Table 5.2 on page 108. t_o was calculated directly from the third case of Equation 5.2 when $M = 0$. These values are used for the subsequent graphs unless otherwise noted. Since the execution time within the RMs is estimated to be no more than 1 μs, the RMs are indeed finished before the VMs which execute in 83.13 µs. One may conclude that the reduced performance of the non-pipelined RM architecture does not affect system performance adversely in this sort application.

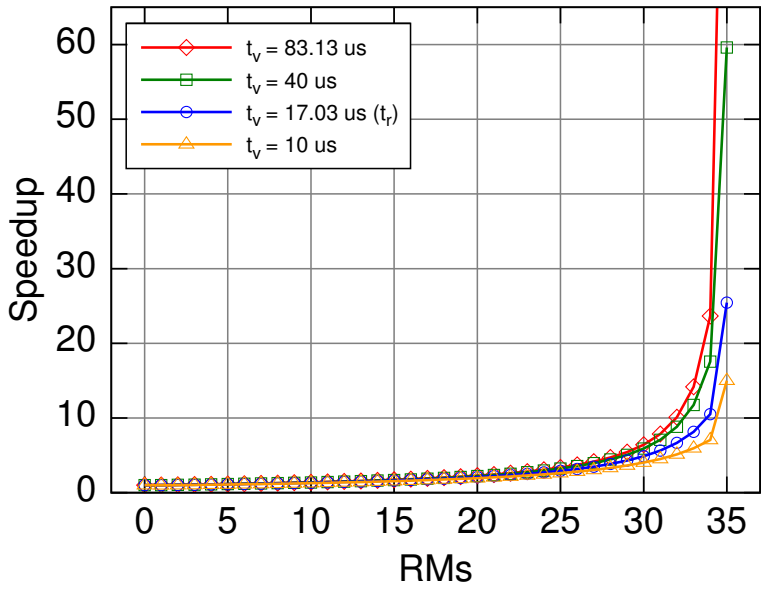


Figure 5.16. Modeled Effect of t_v on Speedup (10k simulation cycles)

The following equation extends Equation 5.2 with $T_0 = T(0) = Ct_o + PCt_v$ and the addition of t_R to model the general speedup of this migration system:

$$S = \begin{cases} \dfrac{Ct_o + PCt_v}{C(t_o + [P-r]t_v) + rM(t_m + t_R)} = 1, \text{ for } r = 0, \\ \dfrac{Ct_o + PCt_v}{C(t_o + t_r + [P-r]t_v) + rM(t_m + t_R)}, \text{ for } 0 < r < P, \\ \dfrac{Ct_o + PCt_v}{Ct_o + rM(t_m + t_R)}, \text{ for } r = P, \end{cases} \tag{5.3}$$

This model fits the 35-, 1024-, and 10240-simulation-cycle plots of Figure 5.15 to within 2.3%, 1.0%, and 0.61% error, respectively.

No complete system would run CIS code in VMs without compiling it to native code. The empirical model is used to explore the effect of VM-related run times (t_v). Figure 5.16 shows that speedups of 15 are possible even if $t_v < t_r$. Since t_v includes the time to execute a process and to update process inputs and outputs, it is unlikely to ever be less than t_r.

Finally, consider the theoretical effect of run-time routing for hardware connectivity, modeled with t_R. Figure 5.17 shows that routing overhead could have a significant detrimental effect on the speedup, unless efficient ways of routing are developed. Route times are difficult to estimate, but one aspect of the system suggests that routing can be done quickly: Because

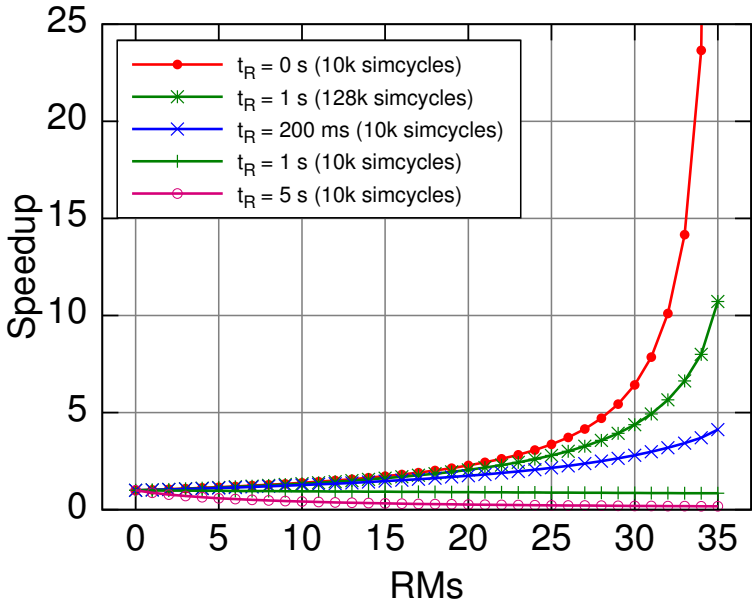


Figure 5.17. Modeled Effect of t_R on Speedup ($t_m = 6.0$ ms)

hardware connected I/Os are updated between simulation cycles during t_o, the RMs' inputs can tolerate multi-cycle delays. Thus, timing is not critical in the path calculation, and any path is likely to be acceptable. Regardless, as with migration overhead t_m, the speedup can be improved if the simulation characteristics allow us to amortize t_R over a greater number of simulation cycles. As shown on the plots, an SMR of 128k:1 for the $t_R = 1$ s plot improves the speedup from 0.85 to 10.7. Another possibility is that the other PowerPC processor in the XC2VP30 is utilized to calculate routes in parallel while using software connectivity in the meantime. This scenario is modeled in the next section.

5.2.3 Algorithmic Modeling

A mathematical model based upon the empirical data is useful when simplifying assumptions are made about the behavior of the system. A more complex model is required to better understand the behavior in real systems. For example, the modeling of parallel route calculations is a difficult behavior to model with mathematical formulas. This section explains an algorithmic model based on Equation 5.3 that is executed as a C program. (See Appendix C for the actual code used.) The main benefit of the algorithmic model is the ability to simulate

various combinations of complex behavior to see the effect. Table 5.3 shows the parameters used in this model.

Table 5.3. Empirical Model Parameters

Parameter	***Symbol***	***Description***
Processes	`P`	The number of processes in the system.
RMs	`R_max`	The number of RMs available in the system.
Cycles	`C`	The number of simulation cycles being modeled.
RM Execution Time	`t_re`	The amount of time required to execute a process in an RM every cycle.
HW Switch Time	`t_ro_hwswitching`	The amount of time required to transfer outputs to inputs between RMs in hardware. This is modeled as 0, because nothing is required to be done by software.
SW Switch Time	`t_ro_swswitching`	The amount of time required to transfer VM/RM outputs to RM inputs with software.
VM Execution Time	`t_ve`	The amount of time required to execute a process in a VM each cycle.
VM Overhead	`t_vo`	The amount of overhead time required to prepare a process for execution in a VM each cycle.
Cycle Overhead	`t_o`	The overhead time required to finish a simulation cycle and begin the next.
Process Activity	*n/a*	Whether a process is active or not is determined by a function call in the model, and various algorithms are used.
Routing Time	`t_R`	The time required to build the routes between RMs after a process has been migrated to one or both of them.

(Continued on next page)

Table 5.3. Empirical Model Parameters (continued)

Parameter	*Symbol*	*Description*
Migration Time	*n/a*	The time required to migrate state and/or code to a processor. This parameter is implemented with a function call that is selected during compile time. General options are to use the empirical step function shown in Figure 5.14 (which is specific to the EOT test), the worst-case migration time, the average migration time, or an explicit value.
Code Size	`CODE_SIZE`	A multiplier for `t_re` and `t_ve` used to model process size.

The algorithmic model is a C program that executes in three phases per simulation cycle. The first phase is to calculate the number of idle RMs there are in the array and the number of VMs that are active (model code line 351). If an RM is idle, then it may be replaced with an active VM, so the number of migrations is calculated. In the implementation of Section 5.1, it is required to read, modify, and write frames to migrate state *into* an RM, so this model assumes that migrating process state *out* of an RM incurs no additional overhead. The second phase of the model then accumulates the total migration time of the simulation cycle for every active VM that is placed into an idle RM (model code line 378). The final phase of the model accumulates the run times and overhead times for each VM and RM for the simulation cycle (model code line 425). In the cases where Parallel Route Calculation (PRC) is done, an additional phase of the model is executed (model code line 467), but this phase is described in “Parallel Route Calculation” on page 120.

Calibrating the Model

To ensure that the model reflects the behavior of the system, the graphs of Figure 5.13, Figure 5.15, and Figure 5.16 were reproduced using the model. The behavior of frame caching was also implemented in the model (Appendix C, line 229), so the result shown in Figure 5.18 demonstrates the same steps due to frame caching as the implementation. Figure 5.19 shows

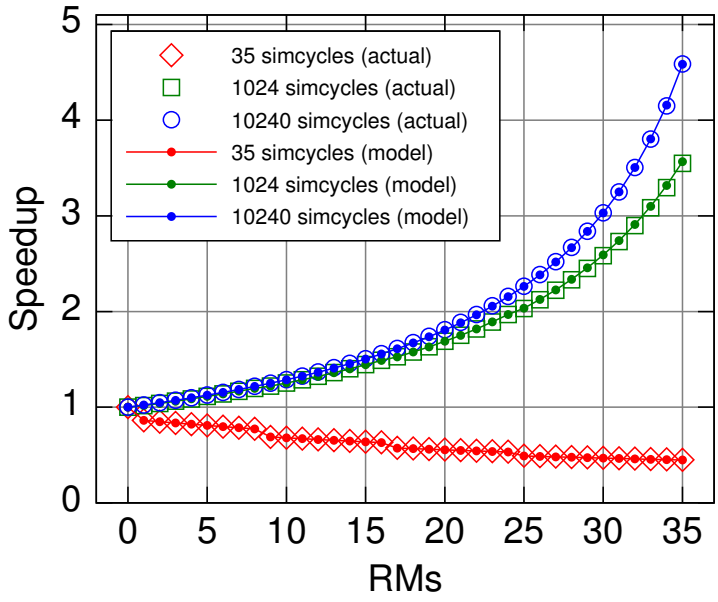


Figure 5.18. Software Connectivity Calibration Graph

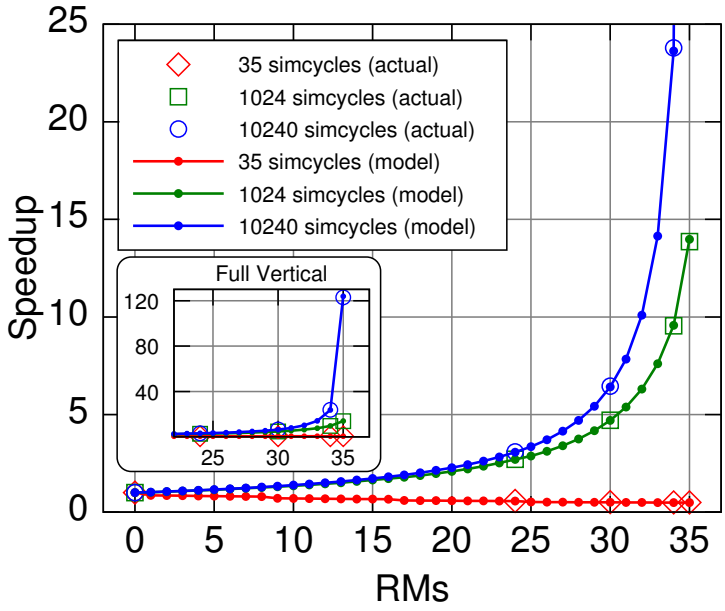


Figure 5.19. Hardware Connectivity Calibration Graph

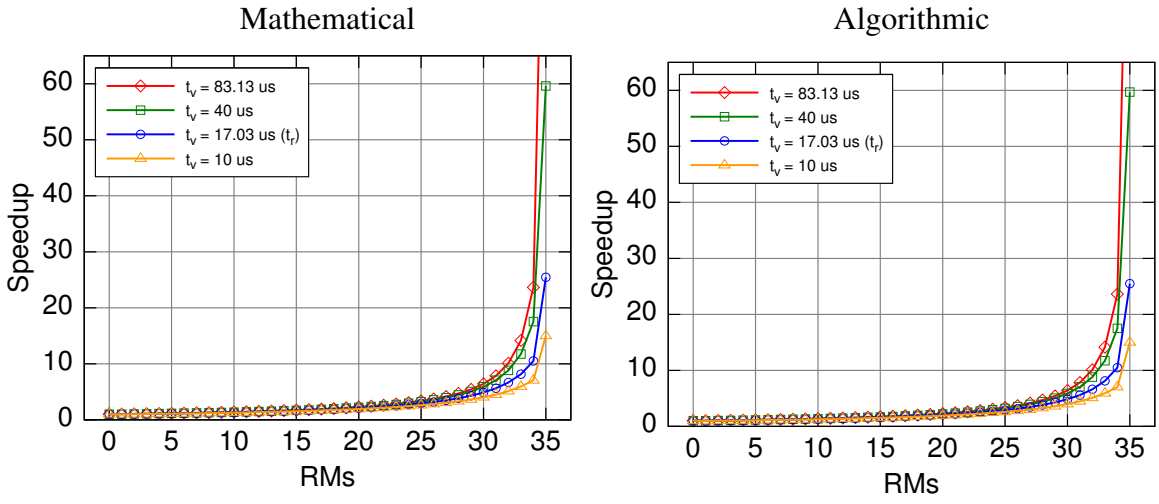


Figure 5.20. Comparison Between Mathematical and Algorithmic Models Varying t_v

the model-produced data that mimics the behavior of the implemented system. Finally, the comparison of the theoretical calculations of Figure 5.16 are shown in Figure 5.20 side by side. There is only a slight variation in the graphs between the two models that is not discernible without animation between the respective plots. This difference is due to the algorithmic model implementing the frame caching function.

There are several assumptions made within this model for the results that follow:

1. All processes are the same size, and they have the same execution times, although the execution times may be different if run in an RM versus a VM. This assumption is justified by the fact that a simulator is able to take canonical RTL processes and combine them to balance the load among RMs and VMs. (See Section 4.1.)

2. The average migration time of 12.47 ms is used. This is the average of the maximum migration time and the minimum migration time from Figure 5.14. This assumption is justified simply because the worst-case migration time would be too pessimistic, and the best case would be too optimistic. The average is a reasonable estimate without further data.

3. The VM execution time is equal to the RM execution time (`t_ve` = `t_re` = 1 µs). This assumption is justified as follows: The processor executing the VMs may be faster than the RMs on a clock rate basis, but it suffers from cache misses where the RMs do not,

and a JIT compiler may result in code that contains more instructions than the RMs need to execute.

4. VM overhead is slightly less than the RM software connectivity overhead. Specifically, `t_vo` = 15.0 µs, and `t_ro_swswitching` = 17.03 µs. This is assumed because the software structures are kept within the simulator to manage interprocess connectivity regardless of the location of the process, so the only difference between RM software connectivity and VM connectivity is that RM connectivity requires access through a mailbox to the inputs and outputs of the RMs. It is assumed these mailbox accesses are slower than the memory accesses for the VMs.

Active Process Ratios

In Section 4.2.3 on page 72, the OpenCores code was analyzed using Trigger Count, Hit Count, and Process Load. For the sake of modeling, the processes are assumed to be identical (see the modeling assumptions on page 117) and so the Process Loads are not used, since they would be equal anyway. Rather, process activity is specified as the *Active Process Ratio*, a ratio between the active processes and the total number of processes. Within the model, process activity is determine through a call to a user-supplied function which returns zero for inactive and non-zero for active on a per-process basis. See Appendix C, line 211. Shell scripts write the Activity Function code to `activity_code.c`, which is included during compile time. The function is passed the number of the process (`p`), the cycle number (`c`), and the number of available RMs (`RMs_avail`).

The APR models the behavior of the OpenCores code shown in the graphs such as Figure 4.8. In those graphs, a relatively small portion of the processes are active during the phases of the simulations. If 17 of the 35 processes of a simulation are active, then the APR is 17:35. APRs of 17:35 and 30:35 are shown in Figure 5.21 with an APR of 1:1 as a baseline. If a single graph can summarize the thesis, Figure 5.21 does. It shows that the APR (which is an indication of exploitable executive locality of reference) allows a simulation speedup to be achieved with less hardware. The maximum speedup for each plot shows that overhead is also reduced because inactive processes are not migrated to hardware.

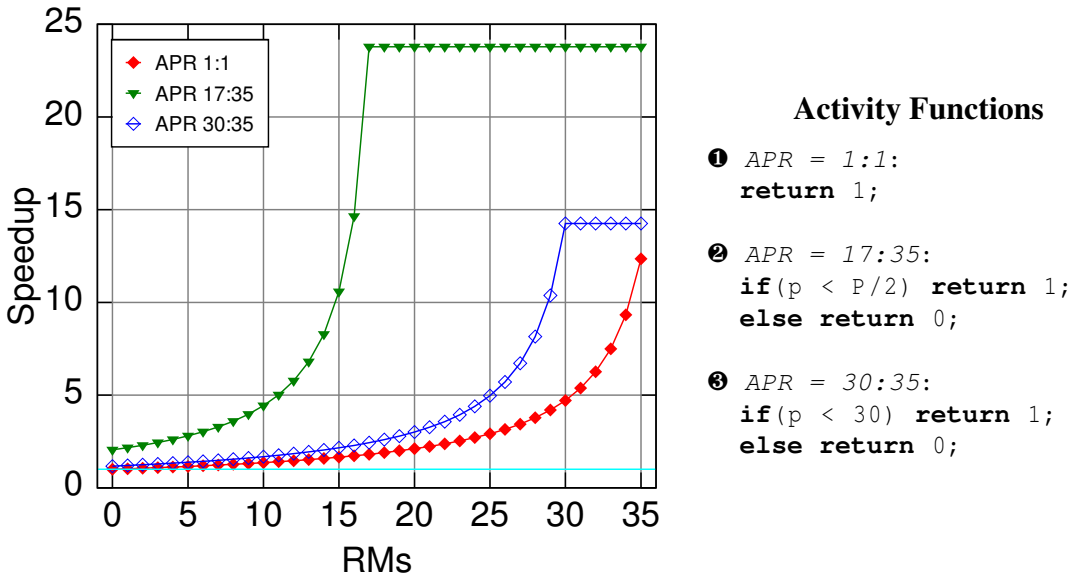
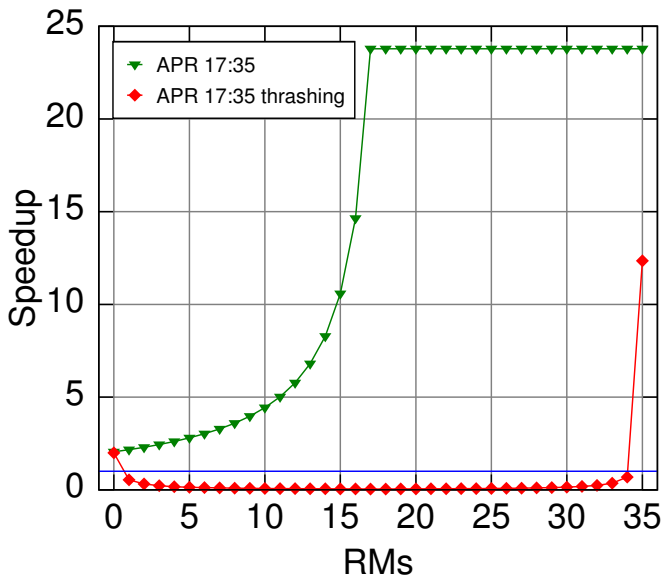


Figure 5.21. Modeled Effect of APR on Speedup

The previous graph was generated with static Activity Functions: The same processes are active or idle for each cycle of the simulation. It is possible, however, to change a process's activity during the simulation, but at any given time, the overall APR can be kept constant. A naïve migration algorithm may migrate idle processes out of RMs as soon as it detects that they are idle, and with certain execution patterns this algorithm would result in a slowdown because of migration overheads. In typical computing contexts, "thrashing" is an over-subscription of the physical memory space so that swaps to and from tertiary storage (e.g., magnetic disks) dominates the execution time. Similarly, if migration occurs too often, then migration overheads dominate the simulation time. Such behavior is shown in Figure 5.22.

The first conditional of the second activity function generates changes in the process activity. The `p == 0` compare ensures that a change is made once for every evaluation of all the processes' activity. The `c > 0` ensures that there are no changes for the first simulation cycle. The `(c % 0x10) == 0` causes the activity to be swapped every 16 simulation cycles. The second conditional (`if(p & 1)`) causes even and odd process numbers to generate opposite activity indications. This code works together to swap the activity of processes every 16 cycles.

Requiring special explanation is the difference in speedup of the final point of each plot (RMs = 35). For the non-thrashing case, only 17 of the 35 processes are ever active, so there are only 17 migrations to RMs. For the thrashing case, all 35 processes are active at one time or



Activity Functions

❶ *Non-Thrashing Case*:

```
if(p < P/2) return 1;
else return 0;
```

❷ *Thrashing Case*:

```
static
 unsigned int retval;

if((p == 0) &&  (c > 0)
 && (c % 0x10) == 0) {
    retval = !retval;
}
if(p & 1)
  return retval;
else
  return !retval;
```

Figure 5.22. Modeled Thrashing

another, so there are 35 migrations during the simulation. The additional 18 migrations for the thrashing case causes an overall speedup that is less than the non-thrashing case. Note also that the final point increases significantly in speedup for the thrashing case because all processes reside in RMs at that time, so thrashing becomes irrelevant.

How often a process changes activity, using this naïve migration algorithm, determines the effect of thrashing. As the number of cycles between activity increases, the negative impact of thrashing decreases as shown in Figure 5.23. This plot shows a simulation of 131,072 simulation cycles shown with the number of cycles between activity change. The speedups are so much higher than those of Figure 5.22 because migration times are amortized over a longer simulation. Also, in order to keep the APR to 17:35, one process was designated as always inactive, and so an increase from 34 to 35 RMs has no effect. The clear point of these plots is that any algorithm that determines when migration occurs must take care to avoid thrashing.

Parallel Route Calculation

As seen with the mathematical model, the time required to route connections between RMs can have a detrimental effect on speedups (see Figure 5.17). A possible optimization is to continue the simulation using software switching after a migration while the routes between RMs are calculated in parallel by another processor. Modeling of such an algorithm is difficult

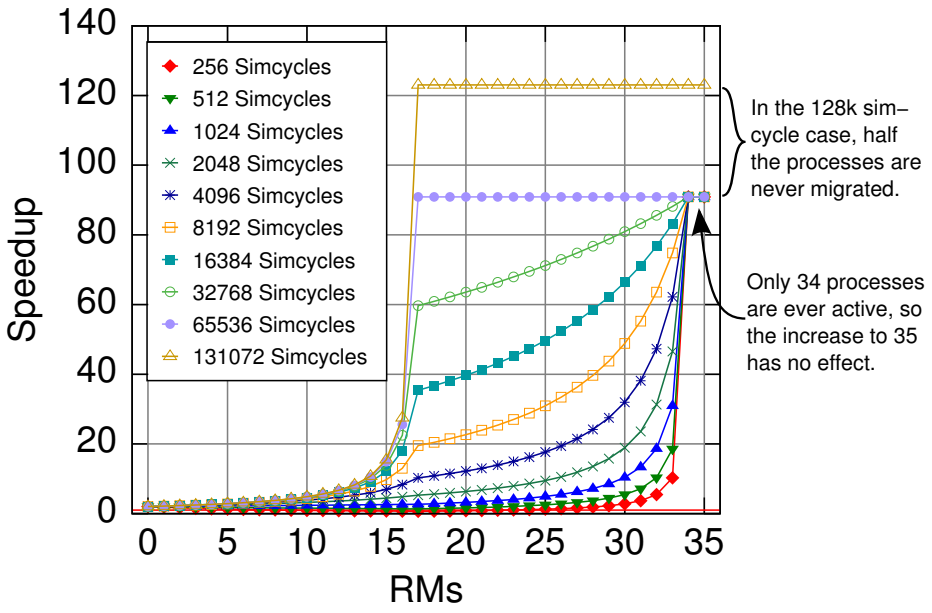


Figure 5.23. Thrashing Comparison (C = 128k Simcycles)

using mathematical equations but is much more easily attained with an algorithmic model. In the algorithm, the behavior is modeled by keeping track of the amount of time left until a route is complete on a per-process basis. While the pending route time is greater than zero, the model accumulates software connectivity route times. When the timer reaches zero, the model accumulates hardware connectivity route times. The implementation is shown in Appendix C at line 467.

The results of such an algorithm are shown in Figure 5.24. The knee in the plot of the figure is related to the length of the simulation. The comparison plot, labeled "Serial Calc.", is a simulation for 10,240 simulation cycles using purely software switching. The simulation is not long enough for the parallel route calculations to complete for the cases where the number of RMs is above 22. Figure 5.25 shows the same behavior as the length of the simulation increases. Longer simulations allow the parallel route calculation to complete and run with the faster route times for the remainder of the simulation to yield even better speedups.

Process-Level Optimizations

Figure 5.26 illustrates eight processes executed in a single simulation cycle with *intra-cycle migration* (a) and with no migration (b). Intracycle migration is the migration of one or

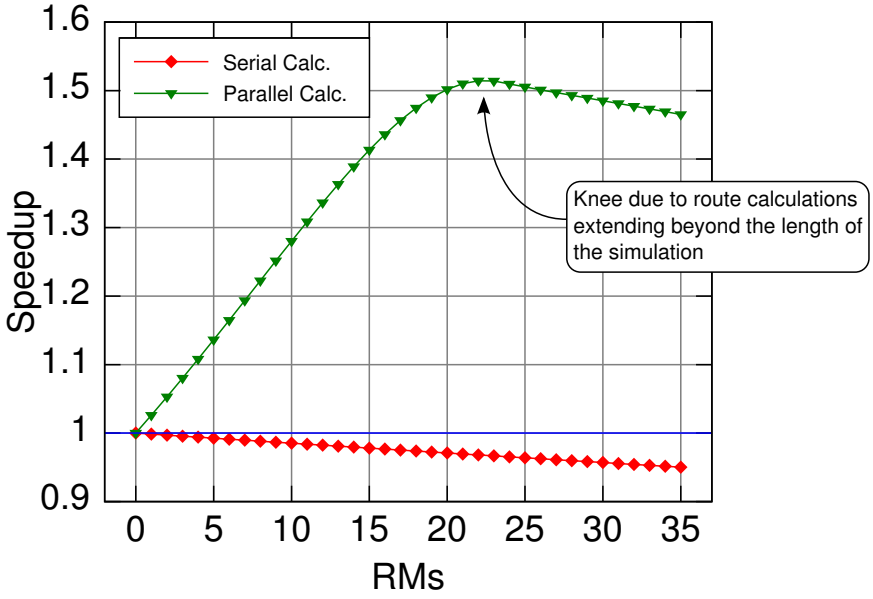


Figure 5.24. Parallel Routing Calculation

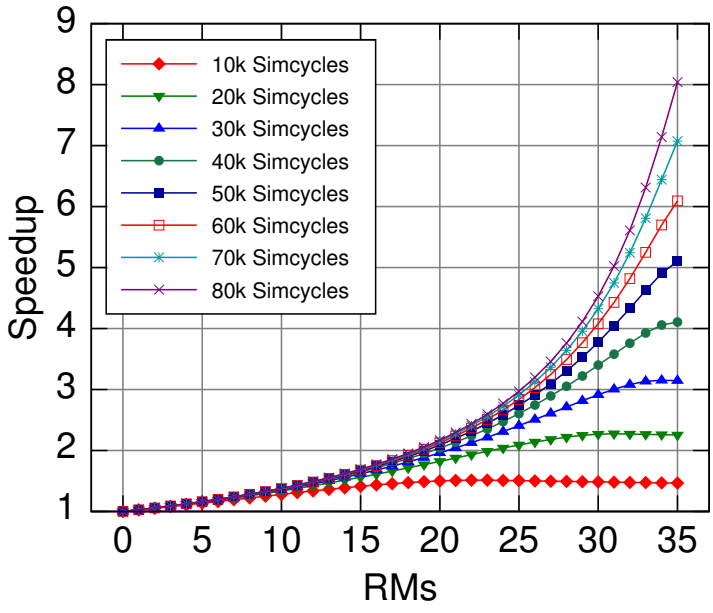


Figure 5.25. Parallel Routing Calculation for Longer Simulations

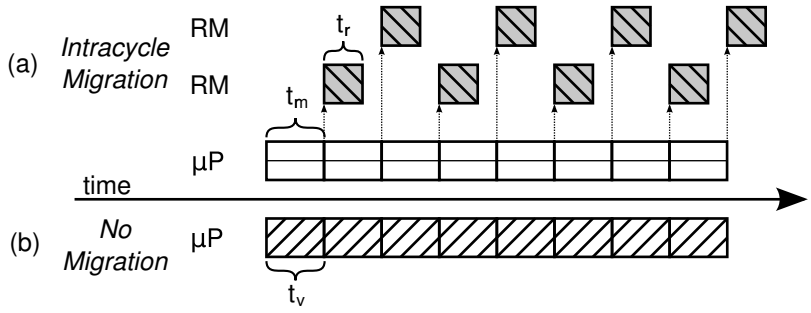


Figure 5.26. Intracycle Migration

more process to a single RM during a simulation cycle. There is no speedup with intracycle migration unless $t_m + t_r < t_v$. Using two RMs may allow the migration overhead and the execution of a process to be overlapped, but when $t_m = t_v$ as shown in the figure, the intracycle migration is slower, and the RMs no longer execute in parallel. In the implementation described in Section 5.1, t_m in the best case is 6.0 ms, whereas t_v is approximately 80 μs, two orders of magnitude smaller. The combining of processes, discussed next, could be used to increase t_v to improve the situation, but the improvement must be significant for intracycle migration to be tenable.

Another optimization is to combine processes at run time using the principles established in Section 4.1.3 for executing multiple processes in a single RM. This method balances the workload more evenly to use the processing resources as efficiently as possible. Figure 5.27 shows three scenarios to illustrate the practical benefit of process merging. In subfigure (a), no load balancing is done, and a single process is assigned to each RM, which are idle for

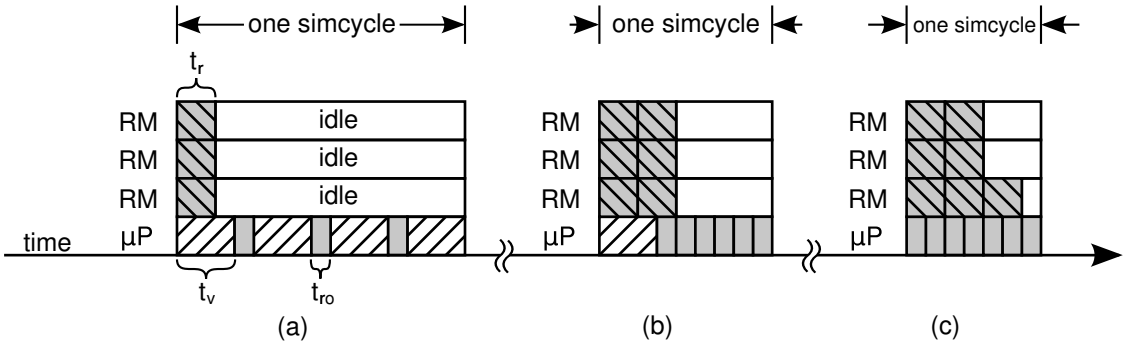


Figure 5.27. Load Balancing

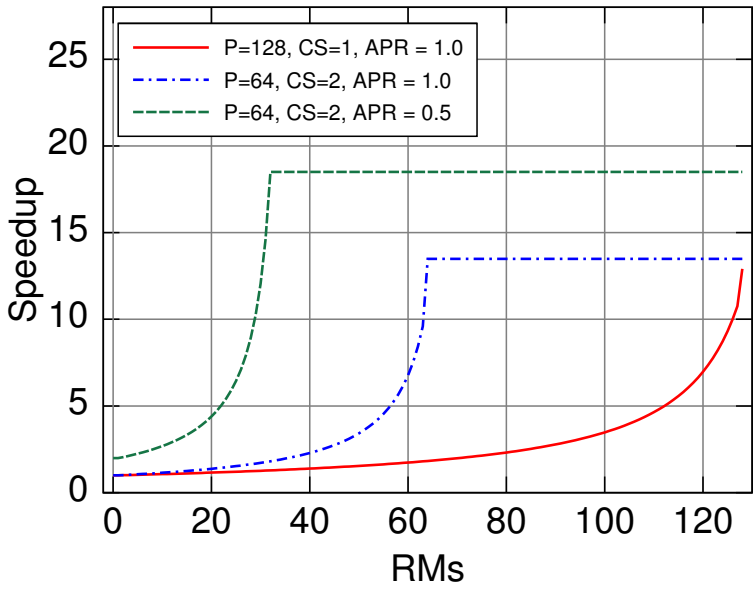


Figure 5.28. Combining Processes

most of the simulation cycle. In (b), each of the RMs is given two processes leaving just one for execution by the microprocessor, and (c) shows further optimization by migrating all the processes into the RMs, and the microprocessor is busy with only RM overheads (t_{ro}). A fully balanced system occurs when $P_r t_r = P_v t_v + P_r t_{ro}$, where P_r is the number of processes in RMs, and P_v is the number in VMs. Achieving this balance may be difficult in a dynamic system where t_r and t_v are unique for each process and are changing as the simulation progresses. The development of algorithms to accurately measure t_v and t_r and dynamically determine how processes should be merged to balance the system is a topic for future work, but the effects of combining processes can be estimated with the algorithmic model. Figure 5.28 shows that doubling the code size to execute a program while halving the number of processes significantly improves performance, especially if used in conjunction with an Active Process Ratio of 50%. The code size is implemented in the model at line 431 and line 435.

5.3 The Theoretical Effect of Custom Hardware

The results thus far have indicated that there are two important parameters of the system that make it difficult for existing FPGAs to be used as a migratory simulator. These parameters

are the migration times which are dominated by the FPGA reconfiguration times and potentially the run-time routing time. This section explores the performance of a system which addresses these bottlenecks.

5.3.1 On-Chip Ring Network

A possible solution to time-consuming routing at run time is a fixed, on-chip network. At the end of each simulation cycle, the RMs and VMs would need to transmit their output values to other RMs and VMs needing them as inputs. This transfer would occur, in the worst case, during the simulation overhead stage (t_o) but before the next simulation cycle commences. The transfer between only RMs could also occur when the VMs are still executing the simulation cycle but the RMs are finished. Transferring at this time would not increase the overall simulation time.

There are many topologies that a network-on-chip could implement, but presented here is a simple ring network. Since one goal is to fit as many processors as possible in the hardware, a unidirectional ring network is modeled because of its simplicity, resulting in lower area utilization. The buses between the RMs are modeled as 8 bits in width in order to lessen the routing requirements, and Figure 5.29 shows a method of connecting each RM to its physically nearest neighbors. When a packet arrives, its destination identifier is compared to the RM's identifier. If they are equal, the packet is received. If not, the packet is forwarded to the next RM. An error condition would occur if a packet is not claimed by another RM, but for the sake of simplicity in this analysis, this error condition is ignored. The trade-off of a ring network is a higher average network latency than other topologies, but it may be possible to hide these latencies while the simulator infrastructure is busy with other things. Furthermore, the speed-up gained from a greater number of processors may make a high-latency network worthwhile.

The following parameters are used in modeling a ring network to estimate performance. Each point-to-point link is 8 bits wide and runs at the 100 MHz clock that the RMs currently employ. Each packet consists of a header containing the number of bytes in the packet and the destination address (see Figure 5.29). For this analysis, it is assumed that the packets consist

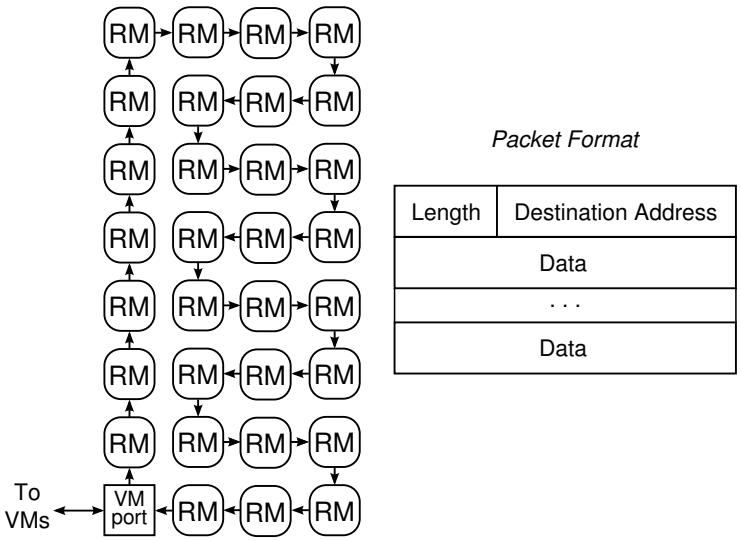


Figure 5.29. Simple Ring Network of RMs

of 16 bits of data plus the header for a total of three cycles per packet. Therefore, each packet requires 30 ns to transfer. At each RM receiver, a pessimistic processing time of 40 cycles is assumed per packet for a total of 430 ns per packet. In the implementation, there are 35 RMs plus one receiver required for the VMs, so the worst-case packet latency is 35 hops of 430 ns each, or 15.05 µs. t_o was measured in the implementation to be 2.96 µs. There is likely some overlap that may occur with the overhead modeled with t_o, but for this analysis the network overhead is simply added. The latency of the ring network, therefore, is modeled with $t_o = 18.01$ µs. The results are shown in Figure 5.30, and one can conclude that the high-latency implementation does degrade performance but less than slow run-time routing times potentially would.

5.3.2 Fast Reconfiguration Logic

The ICAP is certainly a bottleneck within the system. This device has an 8-bit interface, and it is run at 33 MHz in the implementation. One architecture that has been proposed to decrease configuration times uses several planes of configuration SRAM. These planes are easily selectable, so multiple configurations can be loaded, and the configuration can be changed very quickly [83, 84, 85]. Such an architecture could be utilized by a migration system

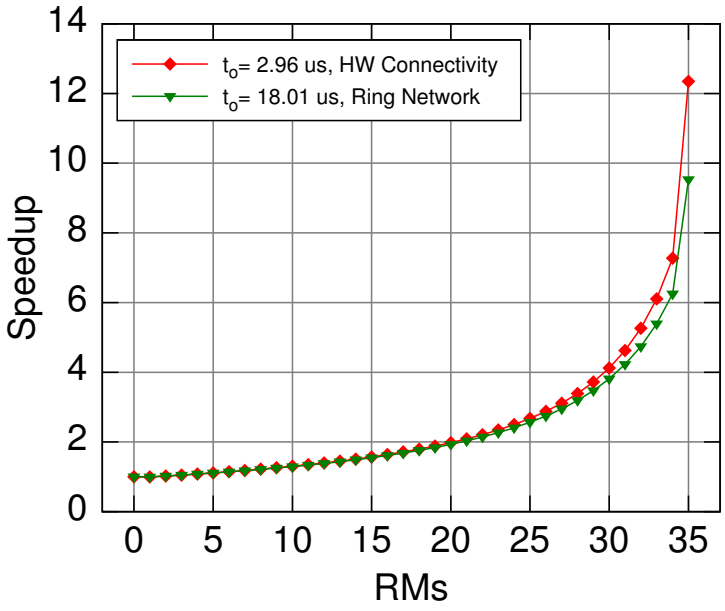


Figure 5.30. Model of Ring Network Connectivity

by loading a configuration into an SRAM plane while the system runs, and then switching to the new configuration, which effects the migration, occurs almost immediately. While such a system could be used to change RM routing or instantiations quickly, it cannot be used by a simulator to migrate state because of inter-cycle dependencies on state. One simulation cycle requires the output of the previous cycle, so state migration can only occur between simulation cycles. The planar architecture would give a performance gain only if the loading of the configuration can occur while the simulation runs.

Another solution is merely to increase the bandwidth of the existing ICAP, and the Virtex-4 and Virtex-5 families have done so. These devices improve the configuration bandwidth in two respects. The ICAPs are able to run at 100 MHz with a 32-bit interface [86, 87], and the frame sizes have decreased from 206 32-bit words (for the XC2VP30) to 41. Since the ICAP requires a dummy frame to be transfered with valid frames, the reconfiguration overhead decreases significantly. Just taking into account the increased clock frequency and data width, there is a 12-fold increase in bandwidth. Figure 5.31 shows the estimated performance gain in these devices. It is estimated that a twelve-fold increase in ICAP performance results in a twelve-fold decrease in migration times, so t_m decreases from 12.47 ms to 1.03 ms for the plot.

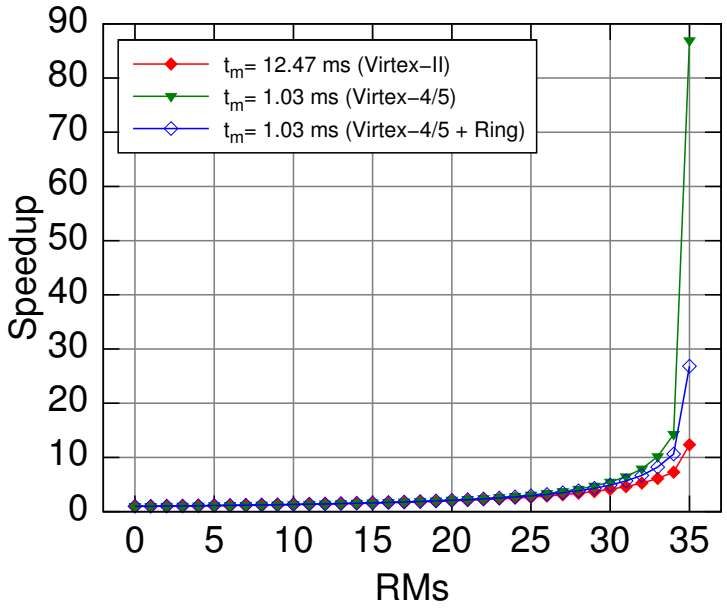


Figure 5.31. High-Performance Reconfiguration

5.4 Extrapolation

If the implementation described in Section 5.1 were compared directly (e.g., in terms of simulation cycles per second) to an Intel Pentium processor running the same VM code, the implementation would be significantly deficient in performance. An Intel Pentium M processor, by no means the fastest Pentium model, running at 1.8 GHz can execute the non-native-compiled VM code for the EOT sort in 0.969 µs[1] compared to the 83 ms of the PowerPC in the XC2VP30. There are at least two reasons for the difference: The PowerPC core is running at 300 MHz rather than 1.8 GHz, and its caches are much smaller than a Pentium's. For this reason, the results thus far have been expressed in terms of relative speedup, rather than absolute simulation cycles per second. The goal has been to describe the relative performance obtained by a processor with and without an execution cache. It is desirable, however, to estimate how a faster, larger system would perform, and this section presents a simple evaluation of performance through extrapolation.

The algorithmic model is used for this purpose with the same assumptions as those delineated on page 117, with some adjustments:

[1] Measurement obtained using the Time Stamp Counter (TSC) as described in Section 4.2.1 on page 67.

- It is assumed that the main processor is capable of running a VM process in 0.1 µs, with RM execution time remaining at 1.0 µs (`t_ve` = 0.1 µs, `t_re` = 1.0 µs).
- The number of RMs and simulation processes is increased to 1024 (`P` = `R_max` = 1024).
- Based on the modeled behavior of the Virtex-4 and -5 FPGAs (Section 5.3.2), it is assumed that the migration time is 1 ms.
- The RMs are interconnected with an on-chip network that increases the cycle overhead (`t_o`) to 5 µs, through the use of a more efficient topology than the ring network of Section 5.3.1.
- The SMR is 10240:1.

The initial results are shown in Figure 5.32, and it is easy to see that a speedup is very difficult to achieve. The abrupt change in the behavior above 1000 RMs is due to the fact that the VMs are faster than the RMs. When the number of processes running in VMs becomes low, the RMs are the execution bottleneck, so increasing the number of RMs increases the execution time and decreases the speedup.

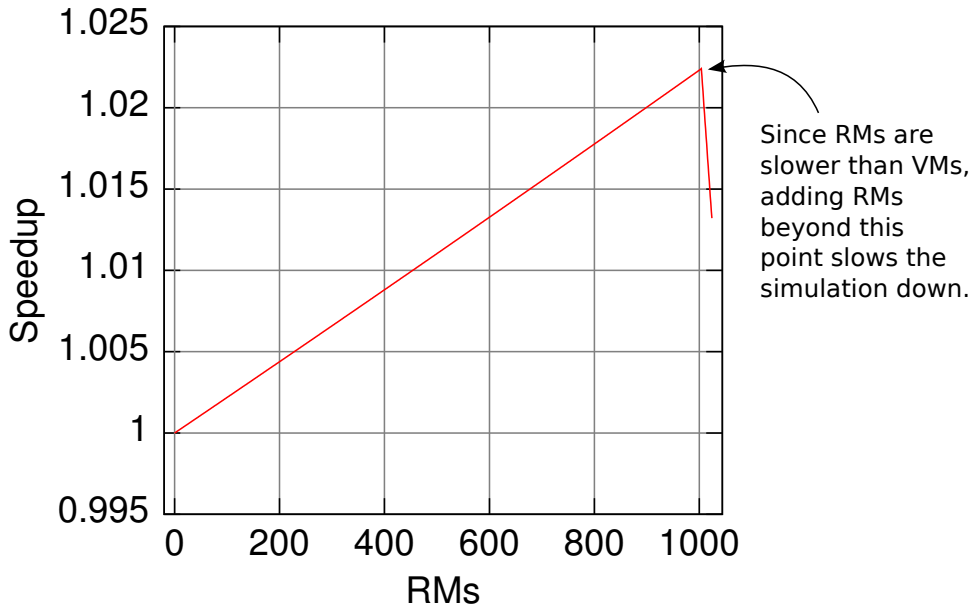


Figure 5.32. Extrapolating to a Larger, Faster System

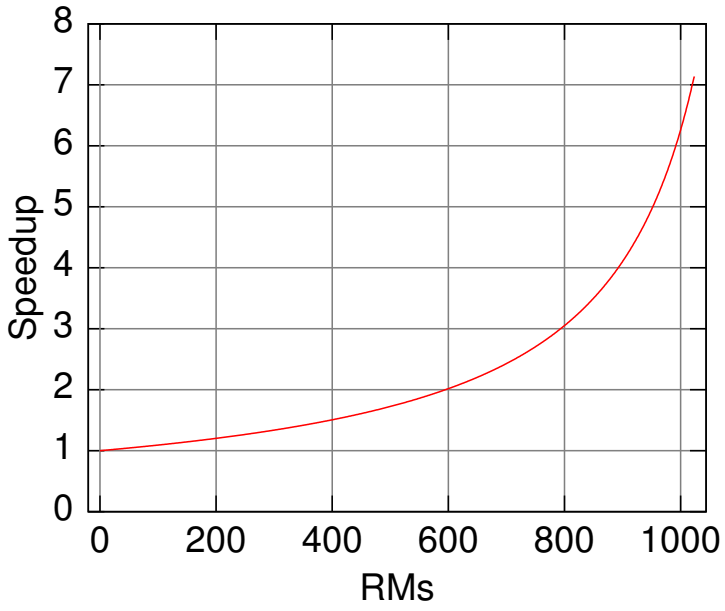


Figure 5.33. Faster System with a Ten-Fold Decrease in Migration Overhead

One conclusion of the performance measured in Section 5.2.1 is that migration time is a significant obstacle to simulation performance, and in a faster system, that migration time is a greater contributor to the execution time. Even with a faster reconfiguration time of 1 ms, Figure 5.32 shows that it is still significant enough to suppress the speedup. Figure 5.33 shows an additional ten-fold decrease in migration overhead (to 100 µs), and even then the maximum speedup only exceeds 7 slightly. The conclusion of these graphs is that the migration system must scale in performance—particularly migration performance—along with the main processor, or speedup is severely limited.

Another look at Equation 5.3 on page 112 supports this conclusion and provides a general rule for speedup behavior in terms of migration overhead and main processor performance. The third case of the speedup, which normally corresponds to the maximum speedup, is

$$S = \frac{Ct_o + PCt_v}{Ct_o + PMt_m}, \tag{5.4}$$

when run-time routing is not used. Note that this equation, when evaluated with the parameters used to produce Figure 5.33, yields 7.16, the maximum speedup of that plot. Equation 5.4 shows that a speedup greater than 1 is possible only if $PMt_m < PCt_v$. This relation can be

further simplified to

$$t_m < \frac{C}{M}\, t_v, \tag{5.5}$$

which shows that at least one of two things must occur as system performance increases: Either t_m must decrease with t_v as the system performance increases, or $\frac{C}{M}$, which is the SMR discussed on page 108, must increase. SMRs are highly dependent on the simulation test bench, and may not be arbitrarily increased.

5.5 Conclusion

This chapter presented an implementation of a migration system that uses run time reconfiguration to update the state of memories within an FPGA. The purpose of the implementation was to provide measurements for a starting point for empirical modeling. The empirical models are used to explore the effect of various timings upon the relative performance of the system, and the two primary hurdles that the models identified are the migration time and the dynamic routing time. The slow migration times may be addressed with faster reconfiguration support within FPGAs. The dynamic routing time may be addressed with a Network-on-Chip, obviating the need for run-time route calculations, or with parallel route calculations in a dual processor system.

Even with slower migration and routing times, it is possible to get a simulation speedup because of locality of reference. The Active Process Ratio, which models the activity and inactivity of processes exhibited by the OpenCores code, demonstrates that significant gains can be achieved with less hardware when only the active processes are migrated. A system must be careful, however, when process activity changes often to avoid thrashing.

In Chapter 3, a formal model of process migration is developed based on Finite State Machines. That model was limited to state-equivalent realms in which the state spaces of all the machines are identical for two reasons: The implementation described in this chapter is a state-equivalent realm, and the proofs for non-state-equivalent realms are much more difficult and are left to future work. A taxonomy resulted from the definitions developed in that chapter, and the implementation presented in this chapter forms a heterogeneous, state-equivalent superset

realm. It is heterogeneous because the VMs implement a larger instruction set than the RMs, therefore, the machines' alphabets differ and the realm is heterogeneous. It is a superset realm because both the VMs and RMs support a greater instruction set than the EOT application requires.

Chapter 6

Conclusion

As the size of integrated circuits continues to increase, the need for fast simulation technologies will also increase. The thesis of this dissertation is that when RTL code exhibits executive locality of reference, that locality can be exploited to achieve greater performance in simulation acceleration. To support that thesis, three primary areas of research were conducted. First, an analysis of RTL code showed that a simulator has great flexibility in decomposing and combining processes in order to maximize acceleration resources. Second, six code bases of varying sizes and functionality were profiled to demonstrate executive locality of reference. Third, an implementation was developed on the XUPV2P board to explore the feasibility of using existing FPGAs or custom hardware.

6.1 Review of Contributions

Formal model of process migration and related taxonomy

A formal model allows us to reason about a problem in an abstract way, but it is of little use unless it represents concrete behavior. The formal model presented in Chapter 3 describes a system of machines that can each process a portion of a program even when the machines are heterogeneous. A process is defined in an abstract form which can be bound to a concrete Finite State Machine (FSM) and later unbound when the concrete machine has completed its task. In terms of an implementation, the abstract form of the process represents the portable

CIS form described in Section 2.4 on page 23 and in Section 5.2 on page 104. When a process is migrated, the CIS can be translated into a form that the target processor can understand. In the implementation of Chapter 5, no translation is required because the migration realm is a superset, state-equivalent, restricted realm, but in future work the translation may be the synthesis of the process to hardware resources. The proofs of the formal model give confidence to an implementation when it fulfills the requirements of the theorems.

Canonical RTL Processes and behavioral-level acceleration

The introduction of canonical RTL processes into a migration system gives a simulator a basic, event-based building block that it can use in managing resources. These canonical processes can be executed in parallel, giving a simulation speedup, while still maintaining the RTL behavior. The canonical processes may also be combined, according to the rules set forth in Chapter 4, to produce processes that are balanced to maximize the efficiency of each RM in the system (see page 123 within "Process-Level Optimizations").

Definition and demonstration of executive locality of reference and process caching

The principle of executive temporal locality of reference states that an active process will tend to be active again in the near future. Conversely, an inactive process will tend to remain inactive. Such behaviors can be exploited with the use of a process cache. To demonstrate exploitable executive locality of reference, an analysis was made of six sets of RTL code of varying size and function, and using a method of input monitoring, active and inactive processes are identified. The results, in terms of Process Load, indicate that a significant number of processes of all the designs are inactive during the tests. In traditional simulation acceleration systems, a device under test is mapped statically to the acceleration hardware before run time, but the activity of a process can only be determined at run time. Compile-time, static mapping is not the most efficient use of the hardware resources. Furthermore, the results show that some processes change activity during a simulation; these are prime candidates for mid-simulation migration to optimize the utilization of the hardware.

The successful implementation of a migration system using existing FPGAs and tools

A demonstration system was implemented to prove the concept of hardware/software process migration and to take measurements of the system for evaluation and modeling. A simple even-odd transposition sort was used to demonstrate the capabilities because of its consistent properties: The number of processes migrated to hardware is easily varied, and the length of the simulation can be changed without affecting the simulation result. The system demonstrated speedups in excess of 20 including all overheads. Such speedups, however, are artificially high due to the execution of processes within virtual machines without just-in-time compilation to native code and due to the pre-configuration of hardware routes. These limitations are addresses by modeling.

Empirical and algorithmic models to explore migration system performance

Since a significant amount of effort is required to implement JIT compilation and routers for a production system, any benefits must be understood before committing the resources. Furthermore, the development of neither of these functions advance the state of the art because JIT compilation in VMs is a well-known technology, and run-time routing is the research topic of others. Nevertheless, those unimplemented parts of the system do affect system performance and need to be evaluated, so the results of the implementation were applied to empirical and algorithmic models. These models show that speedups of 15 are still achievable with JIT compilation, and various methods can be employed to address run-time routing times. The models further show that state migration through a reconfiguration path that is faster than that of Virtex-II FPGAs is beneficial.

6.2 Future Work

The results of this research indicate that RTL simulation acceleration can continue to benefit from advances in technology, but with any research, there are questions introduced along the way that require further research. The extent of the implementation described in Chapter 5 included the migration of state between VMs and RMs, but it was mentioned that the migration of processes directly to hardware structures is an important part of such a sys-

tem. It is expected to significantly reduce area utilization in the FPGA as well as execution time. Furthermore, by identifying active processes at run time and only synthesizing those to hardware, the simulation system would end up with a more efficient method of doing what existing simulation accelerators do today. The difference is run-time synthesis of only active processes, rather than compile-time synthesis of all processes.

Future work of synthesizing processes to hardware bears directly on the formal model of Chapter 3: It is often the case that a synthesis tool will remove flip-flops as it optimizes an RTL design for hardware, and when that occurs the result is no longer state equivalent to the original. It could be argued that the state was removed because it was unnecessary, and therefore state equivalence is still preserved; however, such claims must be proved. Furthermore, there are applications where only the state required for a specific computation can be migrated to a specialized machine, and in this case there is definitely not state equivalence. When state equivalence is not guaranteed, it should be understood what must be true of the migration realm in order to guarantee correct results, and so the proofs of non-state-equivalent realms are necessary.

The emphasis of this dissertation has been on the use of run-time, partially reconfigurable devices, but multicore devices, which have received significant interest in the computing industry of late, may also provide a good platform for an execution cache. One such device is the picoChip PC202 [88]. Designed for communications applications, the PC202 consists of an array of 248 DSPs connected with an on-chip network that allows point-to-point and point-to-multipoint communication with dedicated bandwidth. Such a system may be harnessed to allow process state to be migrated among the processors. A similar device is the Rapport Kilocore architecture which purportedly supports multicore chips with thousands of processors that can be "dynamically reconfigured" [89]. A version of this architecture with 256 processors is available today. These platforms may provide the ideal architectures for a migratory simulator, but in both cases there would be no possibility of migrating directly to gates as in an FPGA.

The idea of process migration may also be useful in other applications. Any application that involves the execution of parallel processes with varying activity can utilize process migration in a parallel execution cache. Modeling applications such as automobile traffic,

electric energy consumption, telephone or computer network usage, or virtually any model involving people's behavior would contain components whose activity varies with the time of day. Other modeling applications such as flock behavior or weather prediction would likely involve periods of activity and inactivity which could benefit from process caching.

As mentioned in Chapter 1, some industry observers have characterized the current state of RTL verification technologies as "impractical" and "inadequate." It is hoped that the technology presented and developed in this dissertation will prove useful to the digital verification industry to alleviate these shortcomings as Gordon Moore's predictions continue to be fulfilled.

References

[1] Intel Corporation. Moore's Law: Raising the bar. ftp://download.intel.com/museum/Moores_Law/Printed_Materials/Moores_Law_Backgrounder.pdf, 2005.

[2] Sunil Kakkar. Proactive approach needed for verification crisis. *EETimes*, April 2004.

[3] Alain Raynaud. The new gate count: What is verification's real cost? *Electronic Design*, October 2003.

[4] Richard Goering. Panelists explore system-to-silicon link. *EETimes*, March 2006.

[5] Bryan Bower. The 'what and why' of TLM. *EETimes*, March 2006.

[6] Peter Varhol. Is software the new hardware? *EETimes*, August 2006.

[7] Mario Larouche. Infusing speed and visibility into ASIC verification. http://www.synplicity.com/totalrecall_wp/syn_web.html, January 2007.

[8] Richard Goering. Tools ease transaction-level modeling. *EETimes*, January 2006.

[9] Robert W. Brodersen Chen Chang, John Wawrzynek. BEE2: A high-end reconfigurable computing system. *IEEE Design & Test of Computers*, pages 114–125, March-April 2005.

[10] John Hennesey and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[11] Torsten Kempf, Malte Doerper, R. Leupers, G. Ascheid, H. Meyr, Tim Kogel, and Bart Vanthournout. A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms. In *Proceedings of DATE*, 2005.

[12] Aric D. Blumer, Henning Mortveit, and Cameron D. Patterson. Formal modeling of process migration. In *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications*, 2007.

[13] Aric D. Blumer and Cameron D. Patterson. Hardware/software process migration and RTL simulation. In *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications*, 2007.

[14] Richard Wawrzyniak. Changing the systems landscape with low-cost FPGAs. *Xcell Journal*, Second Quarter 2005.

[15] Michael D. Ciletti. *Advanced Digital Design with the Verilog HDL*. Prentice Hall, 2003.

[16] Patrick Schaumont, Ingrid Verbauwhede, Kurt Keutzer, and Majid Sarrafzadeh. A quick safari through the reconfiguration jungle. In *Proceedings Design Automation Conference*, pages 172–177, 2001.

[17] Reiner Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings Design, Automation and Test in Europe, Conference and Exhibition*, pages 642–649, 2001.

[18] André DeHon and John Wawrzynek. Reconfigurable computing: What, why, and implications for design automation. In *Design Automation Conference (DAC)*, pages 610–615, 1999.

[19] Cindy Kao. Benefits of partial reconfiguration. *Xcell Journal*, pages 65–67, Fourth Quarter 2005.

[20] Xilinx, Inc. *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*. Xilinx, Inc., March 2005.

[21] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, March 1993.

[22] John R. Hauser and John Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proceedings FCCM*, pages 12–21, 1997.

[23] Xilinx, Inc. *Virtex-4 User Guide*. Xilinx, Inc., September 2005.

[24] Xilinx, Inc. *PowerPC Instruction Set Extension Guide: ISA Support for the PowerPC APU Controller in Virtex-4*. Xilinx, Inc., April 2005.

[25] Altera Corporation. Excalibur devices. http://www.altera.com/products/devices/arm/arm-index.html, 2005.

[26] Michael J. Wirthlin and Brad L. Hutchings. A dynamic instruction set computer. In *Proceedings FCCM*, pages 99–107. IEEE, 1995.

[27] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. Piperench: A reconfigurable architecture and compiler. *IEEE Computer*, 2000.

[28] Cameron D. Patterson. A dynamic module server for embedded platform FPGAs. In *Proceedings ERSA*, pages 31–40, 2003.

[29] Brian Bailey, Andrew Piziali, and Grant Martin. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers, March 2007.

[30] Open SystemC Initiative. Home - Open SystemC Initiative (OSCI). http://www.systemc.org/, 2007.

[31] Celoxica Limited. Handel-C. http://www.celoxica.com/technology/c_design/handel-c.asp, 2005.

[32] IEEE Computer Society. *IEEE Standard for Verilog® Hardware Description Language (Std 1364-2005)*. Institute of Electrical and Electronics Engineers, Inc., 2006.

[33] IEEE Computer Society. *IEEE Standard VHDL Language Reference Manual (Std 1076-2002)*. Institute of Electrical and Electronics Engineers, Inc., 2002.

[34] Roger D. Smith. Simulation article. *Encyclopedia of Computer Science*, 2000.

[35] Denis Döhler, Klaus Hering, and Wilhelm G. Spruth. Cycle-based simulation on loosely-coupled systems. In *Proc. of the 11th Annual IEEE International ASIC Conference*, 1998.

[36] M.C. Cogswell and D.E. Wood. A hybrid event-simulation/cycle-simulation environment for VHDL-based designs. In *Proc. VHDL International Users' Forum*, pages 258–263, 1997.

[37] Inc. Quickturn Design Systems. Form:10-K Filing Date:3/31/1997. http://sec.edgar-online.com/1997/03/31/00/0000912057-97-010967/Section2.asp, 1997.

[38] Peter Varhol. Simulation tools speed design debug and verification. *Electronic Design*, 2000.

[39] Richard Goering. Synopsys revs up Verilog simulator. *EETimes*, 1999.

[40] Synopsys. Mixed-hdl simulation. http://www.synopsys.com/products/simulation/mixed_hdl_wp.html, 2001.

[41] Synopsys, Inc. Leveraging VCS optimizations for faster simulation. *Verification Avenue: The Synopsys technical bulletin for deisign and verification engineers*, 2001.

[42] Richard Goering and Michael Santarini. Analysis: Cadence rides to Quickturn's rescue. *EETimes*, 1998.

[43] Carl Pixley, Aruna Chittor, Fred Meyer, Steve McMaster, and Dan Benua. Functional verification 2003: technology, tools and methodology. In *Proc. 5th International Conference on ASIC*, volume 1, pages 1–5, 2003.

[44] Gregory D. Peterson. Evaluating simulation acceleration techniques. In *Enabling Technology for Simulation Science V*. SPIE, 2001.

[45] Richard M. Fujimoto. Distributed simulation systems. In *Proceedings 35th Conference on Winter Simulation*, pages 124–134. Winter Simulation Conference, 2003.

[46] Cadence Design Systems. Palladium accelerator/emulator. http://www.cadence.com/products/functional_ver/palladium/index.aspx, 2003.

[47] Cadence Design Systems. Xtreme server. http://www.cadence.co.in/products/functional_ver/xtreme_server/index.aspx, 2005.

[48] Jerry Bauer, Michael Bershteyn, Ian Kaplan, and Paul Vyedin. A reconfigurable logic machine for fast event-driven simulation. In *Proc. Design Automation Conference*, pages 668–671, 1998.

[49] EVE Corporation. How zebu works. http://www.eve-team.com/howzebuworks.html, 2006.

[50] Chen Chang, Kimmo Kuusilinna, Brian Richards, and Robert W. Brodersen. Implementation of bee: a real-time large-scale hardware emulation engine. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 91–99, New York, NY, USA, 2003. ACM Press.

[51] Arvind, Krste Asanović, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Mark Oskin, David Patterson, Jan Rabaey, and John Wawrzynek. RAMP: Research accelerator for multiple processors—a community vision for a shared experimental parallel HW/SW platform. Technical Report UCB//CSD-05-1412, UC Berkeley, September 2005.

[52] Srihari Cadambi, Chandra S Mulpuri, and Pranav N Ashar. A fast, inexpensive and scalable hardware acceleration technique for functional simulation. In *Proceedings 39th Conf. Design Automation*, pages 570–575. ACM Press, 2002.

[53] Richard Goering. Startup liga promises to rev simulation. *EETimes*, 2006.

[54] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[55] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[56] Frédéric Petrót, Denis Hommais, and Alain Greiner. Cycle precise core based hardware/software system simulation with predictable event propagation. *Euromicro*, 00:182, 1997.

[57] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[58] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun. Mobile UNITY: reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, 1997.

[59] Shiyong Lu and Cheng-zhong Xu. A formal framework for agent itinerary specification, security reasoning and logic analysis. In *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on Distributed Computing Systems Workshops*, pages 580–586, 2005.

[60] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[61] Wikipedia. Finite state machine. http://en.wikipedia.org/wiki/Finite_State_Machine, 2006.

[62] Mentor Graphics Corporation. *ModelSim® SE User's Manual: Software Version 6.2f*. Mentor Graphics Corporation, January 2007.

[63] Mentor Graphics Corporation. *ModelSim® SE Reference Manual: Software Version 6.2f*. Mentor Graphics Corporation, January 2007.

[64] Klaus Hering. Parallel cycle simulation. Technical Report 13, des Instituts für Informatik der Universität Leipzig, 1996.

[65] Raimund Ubar, Adam Morawiec, and Jaan Raik. Cycle-based simulation with decision diagrams. In *Proceedings, Design, Automation and Test in Europe*, 1999.

[66] Hannes Muhr and Roland Höller. Accelerating rtl simulation by several orders of magnitude using clock suppression. In *2006 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 123–128, 2006.

[67] Alexey Kupriyanov, Frank Hannig, and Jürgen Teich. High-speed event-driven RTL compiled simulation. In *Proc. 4th SAMOS*, 2004.

[68] Wikipedia. Common subexpression elimination. http://en.wikipedia.org/wiki/Common_subexpression_elimination, 2006.

[69] IEEE Computer Society. *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis (Std 1076.6-2004)*. Institute of Electrical and Electronics Engineers, Inc., 2004.

[70] Opencores.org. http://www.opencores.org/, 2006.

[71] Cadence Design Systems, Inc. *Cadence® NC-Verilog® Simulator Help: Product Version 5.1*. Cadence Design Systems, Inc., September 2003.

[72] Intel Corporation. Using the RDTSC instruction for performance monitoring, 1997.

[73] Rahul Razdan, Gabriel P. Bischoff, and Ernst G. Ulrich. Clock suppression techniques for synchronous circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(10):1547–1556, 1993.

[74] Lionel Bening. A two-state methodology for RTL logic simulation. In *Proceedings of 36th DAC*, 1999.

[75] Xilinx, Inc. Xilinx University Program: Xilinx XUP Virtex-II Pro Development System. http://www.xilinx.com/univ/xupv2p.html, 2005.

[76] Xilinx, Inc. *OS and Libraries Document Collection*. Xilinx, Inc., July 2005.

[77] Vince Eck, Punit Kalra, Rick LeBlanc, and Jim McManus. Xapp662: In-circuit partial reconfiguration of RocketIO attributes. http://direct.xilinx.com/bvdocs/appnotes/xapp662.pdf, May 2004.

[78] Derek R. Curd. Xapp660: Dynamic reconfiguration of RocketIO MGT attributes. http://direct.xilinx.com/bvdocs/appnotes/xapp660.pdf, February 2004.

[79] Xilinx, Inc. Change to the maximum cclk frequency for virtex-ii selectmap mode. http://www.xilinx.com/bvdocs/notifications/pcn2004-03.pdf, October 2004.

[80] R. Lysecky, Frank Vahid, and Sheldon X.-D. Tan. A study of the scalability of on-chip routing for just-in-time FPGA compilation. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 57–62, 2005.

[81] Peter Athanas, John Bowen, Tim Dunham, Cameron Patterson, Justin Rice, Matt Shelburne, Jorge Suris, Mark Bucciero, and Jonathan Graf. Wires on demand: Run-time communication synthesis for reconfigurable computing. In *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications*, 2007.

[82] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, second edition, 2003.

[83] S.M. Scalera and J.R. Vazquez. The design and implementation of a context switching FPGA. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 78–85, 1998.

[84] Kiran Puttegowda. Context switching strategies in a run-time reconfigurable system. Master's thesis, Virginia Polytechnic Institute and State University, 2002.

[85] Wesley J. Landaker. Using hardware context-switching to enable a multitasking reconfigurable computer system. Master's thesis, Brigham Young University, 2002.

[86] Xilinx, Inc. *Virtex-4 Configuration Guide*. Xilinx, Inc., August 2007.

[87] Xilinx, Inc. *Virtex-5 FPGA Configuration User Guide*. Xilinx, Inc., July 2007.

[88] Picochip Designs Ltd. PC202 integrated baseband processor product brief, 2007.

[89] Rapport, Inc. Kilocore overview. http://www.rapportincorporated.com/kilocore/kilocore.html, 2007.

Appendix A

Coding Resets in RTL

The traditional method for coding asynchronous resets in HDL is as follows (shown in VHDL) [1, p. 144] [2, p. 113][3, p. 326][4, p. 251][5, p. 199][6, p. 85]:

Listing A.1. Typical VHDL Asynchronous Reset Structure

```
1 if(i_reset_l = '0') then
2     -- Reset signals here
3     A <= '0';
4 elsif(i_sysclk'event and i_sysclk = '1') then
5     if(i_enable = '1') then
6         A <= A_next;
7     end if;
8 end if;
```

The goal of this code is to infer a flip-flop with a feedback multiplexor and asynchronous reset as shown in Figure A.1.

Code of this form, however, can lead to difficulties. Suppose that an ASIC is very tight on area resources and that unreset flip-flops are to be used when possible to save space. Control signals within a bus protocol must be reset or their arbitrary startup state can cause unexpected transactions on the bus. Datapath signals, however, do not need to be reset because they are qualified by the control signals. Let us suppose that signal `A` is a datapath signal, and we wish to save space by making it an un-reset flip-flop. The code would take this form:

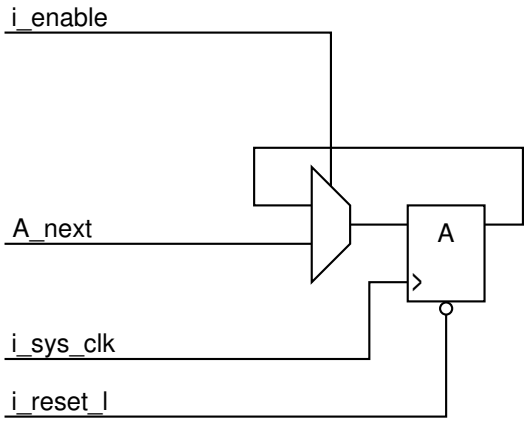


Figure A.1. A Flip-Flop with Feedback Mux and Asynchronous Reset

Listing A.2. Typical VHDL Asynchronous Reset Structure with Unreset Signal

```
1  if(i_reset_l = '0') then
2      -- Reset signals here
3      -- A <= '0';  Save space by not resetting
4      . . .
5  elsif(i_sysclk'event and i_sysclk = '1') then
6      . . .
7      if(i_enable = '1') then
8          A <= A_next;
9      end if;
10     . . .
11 end if;
```

This code, when synthesized, can result in an unexpected component of logic which can affect timing. The signal `A` is synthesized with a feedback multiplexor as shown in Figure A.2 so that `A` will retain its value when `i_enable` is `'0'`. With the code above, `i_reset_l` is placed into the logic of the feedback multiplexor's select signal along with `i_enable`. Examining the HDL code, we see that this is the behavior we described: If `i_reset_l` is `'0'`, `A` must retain its value. This is not the intended behavior (or at least it should not be the intended behavior) when coding resets. Rather, the `i_reset_l` signal should be connected to the reset port of flip-flops and nothing else in the design. Often reset signals are treated like clock trees within a design so the resets can be balanced to ensure that all flip-flops come out of reset at the same time (asynchronous assert, synchronous de-assert), so adding undesired logic into the reset path is undesirable.

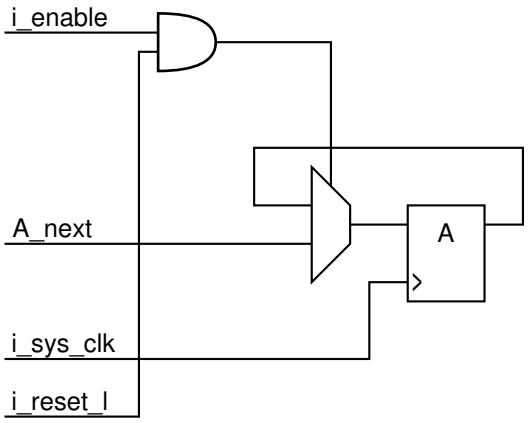


Figure A.2. A Flip-Flop Feedback Mux with Reset Controlling the Mux Selector

To alleviate this unintended behavior, the HDL code should be coded as follows, which is an approved method of coding sequential logic in the *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis* [7, p. 10]:

Listing A.3. Better VHDL Structure for Asynchronous Reset

```
1 if(i_sysclk'event and i_sysclk = '1') then
2     if(i_enable = '1') then
3         A <= A_next;
4     end if;
5 end if;
6 if(i_reset_l = '0') then
7     -- Reset signals here
8     A <= '0';
9 end if;
```

In this case, `i_reset_l` will not be part of the logic for the feedback multiplexor's select whether we reset signal `A` or not.

Similarly, code to instantiate synchronous resets is usually done as follows:

Listing A.4. Typical VHDL Synchronous Reset Structure

```
1 if(i_sysclk'event and i_sysclk = '1') then
2     if(i_reset_l = '0') then
3         -- Reset signals here
4         A <= '0';
5     elsif(i_enable = '1') then
6         A <= A_next;
```

```
7       end if;
8   end if;
```

It should rather be coded as shown below to avoid driving logic with the reset signal for unreset flip-flops:

Listing A.5. Better VHDL Structure for Synchronous Reset

```
1  if(i_sysclk'event and i_sysclk = '1') then
2      if(i_enable = '1') then
3          A <= A_next;
4      end if;
5      if(i_reset_l = '0') then
6          -- Reset signals here
7          A <= '0';
8      end if;
9  end if;
```

References

[1] Roland Airiau, Jean-Michel Bergé, and Vincent Olive. *Circuit Synthesis with VHDL*. Kluwer Academic Publishers, 1994.

[2] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 1996.

[3] Ben Cohen. *VHDL Coding Styles and Methodologies*. Kluwer Academic Publishers, 1995.

[4] Douglas Perry. *VHDL*. McGraw-Hill, 1994.

[5] Joseph Pick. *VHDL Techniques, Experiments, and Caveats*. McGraw-Hill, 1996.

[6] Stefan Sjoholm and Lennart Lindh. *VHDL for Designers*. Prentice Hall, 1997.

[7] IEEE Computer Society. *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis (Std 1076.6-2004)*. Institute of Electrical and Electronics Engineers, Inc., 2004.

Appendix B

VM/RM Instruction Set and Constraints

B.1 Instruction Format

Table B.1. Instruction Format

15:12	11:9	8:6	5:3	2:0
opcode	register A (`rA`)	register B (`rB`)	register C (`rC`)	`extra`

B.2 Opcodes

Table B.2. Opcodes

Opcode	***Encoding***	***Operation*** assignment: ←, concatenation: · , shifts: « »
`NOP`	0 (`b'0000`)	*Do nothing*
`LOAD`	1 (`b'0001`)	`reg[rC]` ← `mem[rA · rB · extra]`

(Continued on next page)

Table B.2. Opcodes (continued)

Opcode	*Encoding*	*Operation*
STORE	2 (b'0010)	mem[rC · extra] ← reg[rA]

COMPARE	3
	(b'0011)
		if((reg[rA] - reg[rB]) == 0) z ← 1
		else z ← 0
		if((reg[rA] - reg[rB]) < 0) c ← 1
		else c ← 0

BRANCH	4 (b'0100)	pc ← rB · rC · extra, if *condition* true. bit 9 inverts *condition* bits 11:10 encode *condition*: 0 := true 1 := z $\stackrel{?}{=}$ 1 2 := c $\stackrel{?}{=}$ 1
SHIFT	5 (b'0101)	if(extra[0] == 1) reg[rC] ← reg[rA] »rB else reg[rC] ← reg[rA] «rB
MOVE	6 (b'0110)	if(extra[0] == 1) reg[rC] ← rA · rB else reg[rC] ← reg[rB]
Reserved	7 (b'0111)	
ADD	8 (b'1000)	if(extra[0] == 1) reg[rC] ← reg[rA] + rB+ (extra[1] and carry) else reg[rC] ← reg[rA] + reg[rB] + (extra[1] and carry)
SUBTRACT[1]	9 (b'1001)	if(extra[0] == 1) reg[rC] ← reg[rA] - rB else reg[rC] ← reg[rA] - reg[rB]
LOGIC	10 (b'1010)	reg[rC] ← reg[rA] *op* reg[rB] where *op* is one of OR, AND, XOR, NOR, NAND, XNOR
Reserved	11 (b'1011)	
Reserved	12 (b'1100)	

(Continued on next page)

[1]SUBTRACT currently does not implement borrow in the RMs since the application does not use it.

Table B.2. Opcodes (continued)

Opcode	*Encoding*	*Operation*
Reserved	13 (b'1101)	
Reserved	14 (b'1110)	
DONE	15 (b'1111)	*Wait for next cycle*

B.3 Assembler Grammar

```
file:
       statement TOK_NEWLINE
     | error TOK_NEWLINE
     | file statement TOK_NEWLINE
     | file error TOK_NEWLINE
     ;

statement:
       /* empty */
     | label_statement
     | load_statement
     | store_statement
     | move_statement
     | compare_statement
     | shift_statement
     | add_statement
     | sub_statement
     | or_statement
     | and_statement
     | xor_statement
     | branch_statement
     | done_statement
     | data_statement
     | input_statement
     | output_statement
     ;

input_statement:
       TOK_INPUT TOK_ID TOK_RANGE TOK_COMMA TOK_REG TOK_DOT
         TOK_RANGE
```

```
31      ;
32
33  output_statement:
34          TOK_OUTPUT TOK_ID TOK_RANGE TOK_COMMA TOK_REG TOK_DOT
35            TOK_RANGE
36      ;
37
38  data_statement:
39          TOK_DATA literal
40      | TOK_DATA literal TOK_COMMA literal
41
42  load_statement:
43          TOK_LOAD reg_spec TOK_COMMA TOK_LEFT_SQUARE literal
44            TOK_RIGHT_SQUARE
45      ;
46
47  store_statement:
48          TOK_STORE reg_spec TOK_COMMA TOK_LEFT_SQUARE literal
49            TOK_RIGHT_SQUARE
50      ;
51
52  move_statement:
53          TOK_MOVE reg_spec TOK_COMMA literal
54      | TOK_MOVE reg_spec TOK_COMMA reg_spec
55      ;
56
57  compare_statement:
58          TOK_CMP reg_spec TOK_COMMA literal
59      | TOK_CMP reg_spec TOK_COMMA reg_spec
60      ;
61
62  shift_statement:
63          TOK_SHIFTL reg_spec TOK_COMMA reg_spec TOK_COMMA literal
64      | TOK_SHIFTR  reg_spec TOK_COMMA reg_spec TOK_COMMA literal
65      | TOK_SHIFTLC reg_spec TOK_COMMA reg_spec TOK_COMMA literal
66      | TOK_SHIFTRC reg_spec TOK_COMMA reg_spec TOK_COMMA literal
67      ;
68
69  add_statement:
70          TOK_ADD  reg_spec TOK_COMMA reg_spec TOK_COMMA
71            add_statement_tail
72      | TOK_ADDC reg_spec TOK_COMMA reg_spec TOK_COMMA
73            add_statement_tail
74      ;
75
76  sub_statement:
77          TOK_SUB  reg_spec TOK_COMMA reg_spec TOK_COMMA
```

```
78          add_statement_tail
79      | TOK_SUBC reg_spec TOK_COMMA reg_spec TOK_COMMA
80          add_statement_tail
81      ;
82
83  xor_statement:
84        TOK_XOR reg_spec TOK_COMMA reg_spec TOK_COMMA reg_spec
85      ;
86
87  and_statement:
88        TOK_AND reg_spec TOK_COMMA reg_spec TOK_COMMA reg_spec
89      ;
90
91  or_statement:
92        TOK_OR reg_spec TOK_COMMA reg_spec TOK_COMMA reg_spec
93      ;
94
95  add_statement_tail:
96        reg_spec
97      | literal
98      ;
99
100
101 branch_statement:
102       TOK_B    literal_or_label
103     | TOK_BZ   literal_or_label
104     | TOK_BNZ  literal_or_label
105     | TOK_BC   literal_or_label
106     | TOK_BNC  literal_or_label
107     ;
108
109 reg_spec:
110       TOK_REG
111     ;
112
113 done_statement:
114       TOK_DONE
115     ;
116
117 literal:
118       TOK_DECIMAL_LITERAL
119     | TOK_HEXADECIMAL_LITERAL
120     ;
121
122 literal_or_label:
123       literal
124     | TOK_BACK_LABEL
```

```
125       | TOK_FRWD_LABEL
126       ;
127
128   label_statement:
129         TOK_DECIMAL_LITERAL TOK_COLON
130       ;
```

B.4 RM Memory Constraints

Table B.3. RM Memory Constraints

RM	***FPGA Column*** ● = data memory, ▲ = program memory, ■ = register file											
	18	***20***	***24***	***42***	***44***	***48***	***60***	***62***	***66***	***78***	***80***	***84***
0	●	▲	■									
1	●	▲	■									
2	●	▲	■									
3	●	▲	■									
4	●	▲	■									
5	●	▲	■									
6	●	▲	■									
7	●	▲	■									
8				●	▲	■						
9				●	▲	■						
10				●	▲	■						
11				●	▲	■						
12				●	▲	■						
13				●	▲	■						
14				●	▲	■						
15				●	▲	■						
16							●	▲	■			
17							●	▲	■			
18							●	▲	■			
19							●	▲	■			
20							●	▲	■			
21							●	▲	■			
22							●	▲	■			
23							●	▲	■			
24										●	▲	■
25										●	▲	■
26										●	▲	■
27										●	▲	■
28										●	▲	■
29										●	▲	■
30										●	▲	■
31										●	▲	■
32				●	▲	■						
33				●	▲	■						
34										●	▲	■

Appendix C

Algorithmic Modeling Code

```
#include <stdio.h>
#include <string.h>  /* for memset */
#include <stdlib.h>  /* for malloc */

#ifndef DEBUG
#   define DEBUG 0
#endif

/*
 * Parameters:
 *   t_ro_swswitching = RM Overhead time in the simulator for SW Switching
 *   t_ro_hwswitching = RM Overhead time in the simulator for HW Switching
 *   (this is zero)
 *   t_re = RM Execution Time
 *   t_ve = VM Execution Time (no overhead)
 *   t_vo = VM Overhead Time (no execution time)
 */
#ifndef t_vm
#   define t_vm          83.1273396084784 /* Microseconds */
#endif
#define t_ro_swswitching 17.0290382431469 /* Microseconds */
#define t_ro_hwswitching 0   /* No simulator overhead in HW switching case */
#define t_o              2.95859          /* Microseconds */
#ifndef t_ve
#   define t_ve          1.0 /* Assume optimistic value for execution time. */
#endif
#ifndef t_vo
#   define t_vo          15.0/* Assume it's a little better than RM overhead */
#endif

/*
 * Mapping to Mathematical Empirical Model:
 *
 *      Algorithmic               Math
 *      t_ro_swswitching        = t_r
 *      t_ro_hwswitching        = n/a (it's always zero)
 *      t_ve + t_vo             = t_v
 *      migration_function()    = t_m
 *      t_R                     = t_R
 *
 *      t_re is not in that model since it is always less than t_v
 *
 *
```

```
44  * Compile-time parameters
45  *    SOFTWARE_SWITCHING        Set to 1 if you want to simulate no HW
46  *                              switching
47  *
48  *    MIGRATION_OVERHEAD        Set to 1 if you want the migration overhead
49  *                              included in the calculations.  This uses a
50  *                              function gleaned from the actual measurements,
51  *                              but is highly specific to the EOT application.
52  *
53  *    MIGRATION_OVERHEAD_WORST_CASE
54  *                              Set to 1 if you want to use the worst-case
55  *                              migration overhead time.
56  *
57  *    MIGRATION_OVERHEAD_AVERAGE
58  *                              Set to 1 if you want to use the average
59  *                              migration overhead time.
60  *
61  *
62  *    C                         The number of simcycles to model.
63  *
64  *    P                         The number of processes to model.
65  *
66  *    R_max                     The maximum number of RMs to use.
67  *
68  *    CODE_SIZE                 A multiplier to t_ve to simulate processes
69  *                              that take longer to execute.
70  *
71  *    PARALLEL_ROUTE_CALC       Set to 1 to model parallel calculation of
72  *                              routes while software switching is used.  The
73  *                              algorithm uses sw switching until t_R expires
74  *                              and then uses hw switching.
75  *
76  *
77  *    t_R                       The modeled routing time.
78  *
79  *    USE_SWITCHING_FUNCTION    Allows scripts to put in a function to
80  *                              determine switching delays.  Used, for
81  *                              example, to implement the switching ratio.
82  *
83  */
84
85  /* Compile-time parameters and defaults if not specified */
86  #ifndef SOFTWARE_SWITCHING
87  #   define SOFTWARE_SWITCHING      1
88  #endif
89  #ifndef MIGRATION_OVERHEAD
90  #   define MIGRATION_OVERHEAD      1
91  #endif
92  #ifndef MIGRATION_OVERHEAD_WORST_CASE
93  #   define MIGRATION_OVERHEAD_WORST_CASE 0
94  #endif
95  #ifndef MIGRATION_OVERHEAD_AVERAGE
96  #   define MIGRATION_OVERHEAD_AVERAGE       0
97  #endif
98  #ifndef C
99  #   define C                       35
100 #endif
101 #ifndef P
102 #   define P                       35
103 #endif
104 #ifndef R_max
105 #   define R_max                   35
106 #endif
107
108 #ifndef CODE_SIZE
109 #   define CODE_SIZE               1
110 #endif
111
```

```
112 #ifndef PARALLEL_ROUTE_CALC
113 #   define PARALLEL_ROUTE_CALC     0
114 #endif
115
116 #ifndef t_R
117 #    define t_R                    0
118 #endif
119
120 #ifndef USE_SWITCHING_FUNCTION
121 #   define USE_SWITCHING_FUNCTION  0
122 #endif
123
124 /*
125  * The active flag is used to arrive at the numerator for the speedup
126  * calculation.  It ensures that all processes are active, so we can see the
127  * difference of different activity ratios in the speedup.
128  */
129 static int active_flag;
130
131 /*
132  * Structure used to maintain the state of each process.
133  */
134 typedef struct proc_state_s {
135     int in_rm;
136     int active;
137     double route_time_remaining;       /* Only used when PARALLEL_ROUTE_CALC */
138 } proc_state_t;
139
140
141 static inline double
142 t_ve_function(unsigned long p)
143 {
144     /* Could be used to implement different execution times per process. */
145     return t_ve; /* microseconds */
146 }
147
148 static inline double
149 t_vo_function(unsigned long p)
150 {
151     /* Could be used to implement different overhead times per process. */
152     return t_vo; /* microseconds */
153 }
154
155 static double
156 t_ro_function(unsigned long p, unsigned long RMs_avail, unsigned long RMs_used,
157     unsigned long c, unsigned long route_time_remaining)
158 {
159 #   if USE_SWITCHING_FUNCTION
160         /*
161          * This is a mechanism for scripts to include their own algorithm to
162          * determine switching type.
163          */
164 #      include "switching_code.c"
165 #   else
166         /*
167          * This code mimics the behavior of the HW switching.  When the last
168          * processes is moved into the RM array, all switching is in HW.  THIS
169          * IS SPECIFIC TO THE EOT ALGORITHM.
170          */
171         if(RMs_used == P) {
172             /* All of the processes are in RMs, so switching is completely in
173              * HW. */
174             return t_ro_hwswitching;
175         } else {
176             /*
177              * Not all processes are in RMs, so we need to return HW switching
178              * for all but the last that is in hardware
179              */
```

```
180             if(p == (RMs_used - 1)) {
181                 /* This is the rightmost, so we return software switching for
182                  * that one. */
183                 return t_ro_swswitching;
184             } else {
185                 return t_ro_hwswitching;
186             }
187         }
188 #   endif
189 }
190
191 static inline double
192 t_re_function(unsigned long p)
193 {
194     /*
195      * We estimate that the execution time of each RM is less than 1 us, so we
196      * assume the worst case here.
197      */
198     return 1.0; /* microseconds */
199 }
200
201 static int
202 is_active(unsigned long p, unsigned long c, unsigned long RMs_avail)
203 {
204     if(active_flag) {
205         return 1;
206     } else {
207         /*
208          * This is a mechanism for scripts to include their own algorithm to
209          * determine process activity.
210          */
211 #       include "activity_code.c"
212     }
213 }
214
215 static inline double
216 migration_function(unsigned long migrations)
217 {
218     if(MIGRATION_OVERHEAD) {
219         if(MIGRATION_OVERHEAD_WORST_CASE) {
220             return 5680650.0/300000000.0 * 1000000;
221         } else {
222             if(MIGRATION_OVERHEAD_AVERAGE) {
223                 return
224                     (
225                         (5680650.0/300000000.0 * 1000000)
226                       + (1802878.0/300000000.0 * 1000000)
227                     ) / 2.0;
228             } else {
229                 switch(migrations) {
230                     /*
231                      * These values are from column 2 of
232                      * ../fpl07/results_20070315_hw35cycles.txt_35.txt
233                      */
234                     case  0: return 0; break;
235                     case  1: return 5680650.0/300000000.0 * 1000000; break;
236                     case  2: return 3489271.0/300000000.0 * 1000000; break;
237                     case  3: return 2759364.0/300000000.0 * 1000000; break;
238                     case  4: return 2394251.0/300000000.0 * 1000000; break;
239                     case  5: return 2175460.0/300000000.0 * 1000000; break;
240                     case  6: return 2029207.0/300000000.0 * 1000000; break;
241                     case  7: return 1925102.0/300000000.0 * 1000000; break;
242                     case  8: return 1846736.0/300000000.0 * 1000000; break;
243                     case  9: return 2271994.0/300000000.0 * 1000000; break;
244                     case 10: return 2174928.0/300000000.0 * 1000000; break;
245                     case 11: return 2095514.0/300000000.0 * 1000000; break;
246                     case 12: return 2029232.0/300000000.0 * 1000000; break;
247                     case 13: return 1973415.0/300000000.0 * 1000000; break;
```

```
248                     case 14: return 1925286.0/300000000.0 * 1000000; break;
249                     case 15: return 1883651.0/300000000.0 * 1000000; break;
250                     case 16: return 1847303.0/300000000.0 * 1000000; break;
251                     case 17: return 2072587.0/300000000.0 * 1000000; break;
252                     case 18: return 2029871.0/300000000.0 * 1000000; break;
253                     case 19: return 1991766.0/300000000.0 * 1000000; break;
254                     case 20: return 1957328.0/300000000.0 * 1000000; break;
255                     case 21: return 1926253.0/300000000.0 * 1000000; break;
256                     case 22: return 1897953.0/300000000.0 * 1000000; break;
257                     case 23: return 1872136.0/300000000.0 * 1000000; break;
258                     case 24: return 1848459.0/300000000.0 * 1000000; break;
259                     case 25: return 2001703.0/300000000.0 * 1000000; break;
260                     case 26: return 1974962.0/300000000.0 * 1000000; break;
261                     case 27: return 1950226.0/300000000.0 * 1000000; break;
262                     case 28: return 1927251.0/300000000.0 * 1000000; break;
263                     case 29: return 1905892.0/300000000.0 * 1000000; break;
264                     case 30: return 1885897.0/300000000.0 * 1000000; break;
265                     case 31: return 1867204.0/300000000.0 * 1000000; break;
266                     case 32: return 1849734.0/300000000.0 * 1000000; break;
267                     case 33: return 1833144.0/300000000.0 * 1000000; break;
268                     case 34: return 1817459.0/300000000.0 * 1000000; break;
269                     case 35: return 1802878.0/300000000.0 * 1000000; break;
270                 }
271                 /* Assume worst-case migration if outside empirical range. */
272                 return 5680650.0/300000000.0 * 1000000;
273             }
274         }
275     } else {
276         return 0;
277     }
278 }
279
280 int
281 main(void)
282 {
283     double total_sim_time, time_delta;
284     double basis;       /* Numerator of speedup calc. */
285     double rm_time;
286     double rm_overhead;
287     double vm_overhead;
288     double vm_time;
289     double migration_time;
290     double routing_time;
291
292     unsigned long RMs_used;             /* Number of RMs in use. */
293     unsigned long RMs_avail;            /* Number of available RMs */
294     unsigned long p;                    /* Process counter. 0 <= p < P */
295     unsigned long c;                    /* Cycle counter.   0 <= c < C */
296     unsigned long migrations;           /* Counts migrations per cycle */
297     unsigned long total_migrations;     /* Stat: total migrations per run */
298     unsigned long idle_RMs;
299     unsigned long active_VMs;
300
301
302     proc_state_t *pstate;               /* Keeps process state */
303     pstate = malloc(P * sizeof(*pstate));
304     if(!pstate) {
305         perror("malloc");
306         return 1;
307     }
308
309     /* Data for input to Gnuplot is written to stdout */
310     /* This is a header containing input params and column labels */
311     printf("# P = %d, R_max = %d, C = %d\n", P, R_max, C);
312     printf("# RMs Speedup        TotalSimTime      Migrations\n");
313
314     active_flag = 1;
315
```

```
316     for(RMs_avail = 0; RMs_avail <= R_max; RMs_avail++) {
317         /*
318          * If this is the second iterations (RMs_avail == 1), and the
319          * active_flag is set, then that means we now want to run again with
320          * RMs_avail = 0, with without forcing all processes to be active.
321          * This allows us to measure the speedup of having idle processes.
322          */
323         if((RMs_avail == 1) && active_flag) {
324             active_flag = 0;
325             RMs_avail = 0;
326         }
327 
328         /* Clear all process state for a new run */
329         memset(pstate, 0, P * sizeof(*pstate));
330 
331         total_sim_time   = 0;
332         total_migrations = 0;
333         RMs_used         = 0;  /* Begin in VMs only */
334 
335         /* Iterate through the cycles */
336         for(c = 0; c < C; c++) {
337             rm_time        = 0;
338             vm_time        = 0;
339             migration_time = 0;
340             routing_time   = 0;
341             rm_overhead    = 0;
342             vm_overhead    = 0;
343             migrations     = 0;
344             idle_RMs       = RMs_avail - RMs_used;
345             active_VMs     = 0;
346 
347             /*
348              * First, calculate how many idle RMs there are and how many
349              * active VMs there are.
350              */
351             for(p = 0; p < P; p++) {
352                 pstate[p].active = is_active(p, c, RMs_avail);
353                 /* Check to see if this process is in an RM already or not. */
354                 if(pstate[p].in_rm) {
355                     /* Check if this process is active or not. */
356                     if(!pstate[p].active) {
357                         /* Process is not active */
358                         idle_RMs++;
359                     }
360                 } else {
361                     /* Process is not in an RM */
362                     if(pstate[p].active) {
363                         /* It's active in a VM */
364                         active_VMs++;
365                     }
366                 }
367             }
368 
369             if(active_VMs < idle_RMs) {
370                 migrations = active_VMs;
371             } else {
372                 migrations = idle_RMs;
373             }
374 
375             {
376                 int mc = migrations;
377 
378                 for(p = 0; p < P; p++) {
379                     if(!mc) break;
380                     if(!pstate[p].in_rm && pstate[p].active) {
381                         if(RMs_used < RMs_avail) {
382                             RMs_used++;
383                         } else {
```

```
                        /*
                         * In this case, we're swapping a used RM (having
                         * an idle process) with a VM, so RMs_used does
                         * not increase.
                         */
                        /* Find an IDLE RM */
                        unsigned long q;
                        for(q = 0; q < P; q++) {
                            if(pstate[q].in_rm == 1 && !pstate[q].active) {
                                /* Found it */
                                pstate[q].in_rm = 0;
                                break;
                            }
                        }
                        if(q == P) {
                            fprintf(stderr, "Did not find one!!\n");
                            exit(1);
                        }
                    }
                    pstate[p].in_rm = 1;
                    mc--;
                    migration_time += migration_function(migrations);
                    if(!PARALLEL_ROUTE_CALC) {
                        if(!SOFTWARE_SWITCHING) {
                            routing_time += t_R;
                            pstate[p].route_time_remaining = 0;
                        }
                    } else {
                        /*
                         * For parallel route calc, we don't add any route
                         * calculation here.  The route is calculated in
                         * the "background," and in the meantime, we do
                         * software switching.
                         */
                        routing_time += 0;
                        pstate[p].route_time_remaining = t_R;
                    }
                }
            }
        }

        /* Now accumulate run times */
        for(p = 0; p < P; p++) {
            if(pstate[p].in_rm) {
                rm_overhead += t_ro_function(p, RMs_avail, RMs_used, c,
                    pstate[p].route_time_remaining);
                if((t_re_function(p) * CODE_SIZE) > rm_time) {
                    rm_time = t_re_function(p) * CODE_SIZE;
                }
            } else {
                if(pstate[p].active) {
                    vm_time     += t_ve_function(p) * CODE_SIZE;
                    vm_overhead += t_vo_function(p);
                } else {
                }
            }
        }


        /*
         * This cycle takes up the maximum of the rm_time or the vm_time,
         * whichever is greater.
         */
        time_delta = 0;
        if(rm_time > vm_time) {
            time_delta += rm_time;
        } else {
            time_delta += vm_time;
```

```
452             }
453             total_migrations += migrations;
454             time_delta       += migration_time;
455             if(!PARALLEL_ROUTE_CALC) {
456                 /*
457                  * If we are calculating routes in parallel, the simulator
458                  * does not take a routing hit here.
459                  */
460                 time_delta += routing_time;
461             }
462             time_delta += rm_overhead;
463             time_delta += vm_overhead;
464             time_delta += t_o;
465             total_sim_time += time_delta;
466 
467             if(PARALLEL_ROUTE_CALC) {
468                 /*
469                  * For parallel route calculations, we have to determine if
470                  * the route time has expired.  If it has, we switch over from
471                  * SW switching to HW switching.
472                  */
473                 for(p = 0; p < P; p++) {
474                     if(pstate[p].route_time_remaining) {
475                         if(time_delta > pstate[p].route_time_remaining) {
476                             time_delta -= pstate[p].route_time_remaining;
477                             pstate[p].route_time_remaining = 0;
478                         } else {
479                             pstate[p].route_time_remaining -= time_delta;
480                             /* We break here because multiple routes cannot be
481                              * calculated at the same time. */
482                             break;
483                         }
484                     }
485                 }
486             }
487 
488             if(DEBUG) {
489                 /* For debugging purposes */
490                 fprintf(stderr, "----------------------------------------\n");
491                 fprintf(stderr, "Cycle␣%lu\n", c);
492                 fprintf(stderr, "␣␣␣␣␣␣␣␣␣rm_time:␣%f\n", rm_time);
493                 fprintf(stderr, "␣␣␣␣␣␣␣␣␣vm_time:␣%f\n", vm_time);
494                 fprintf(stderr, "␣␣␣␣␣rm_overhead:␣%f\n", rm_overhead);
495                 fprintf(stderr, "␣␣␣␣␣vm_overhead:␣%f\n", vm_overhead);
496                 fprintf(stderr, "␣␣␣␣routing_time:␣%f\n", routing_time);
497                 fprintf(stderr, "␣␣migration_time:␣%f\n", migration_time);
498                 fprintf(stderr, "␣␣total_sim_time:␣%f\n", total_sim_time);
499             }
500         }
501 
502         /*
503          * If this is the first iteration and the active flag is set, we want
504          * to record the basis, which is the numerator of the speedup
505          * calculation.
506          */
507         if(RMs_avail == 0 && active_flag) {
508             basis = total_sim_time;
509         } else {
510             printf("%2lu␣%10.6f␣%15.6f␣%lu\n",
511                 RMs_avail, basis / total_sim_time,
512                 total_sim_time, total_migrations);
513         }
514     }
515     return 0;
516 }
```

Appendix D

OpenCores Code Statistics

"Max. Hits" is the maximum number of hits on a single process.

Table D.1. AC97 Tests

Test	*Max. Hits*	*Test Description*
Test 0	580,145	Basic AC97 I/O Test & Reg Wr
Test 1	947,467	Basic AC97 I/O Test & Reg Rd
Test 2	546,088	VSR AC97 I/O Test
Test 3	26,340,560	VSR AC97 I/O Test (INT ctrl)

Table D.2. ATA Tests

Test	*Max. Hits*	*Test Description*
Test 0	1,176,191	IO Test 1 Testing WISHBONE wait state insertion, and iordy assertion.
Test 1	2,480,165	IO Test 2 Testing PIO Modes, iordy assertion and iordy delays.
Test 2	154	INT Test
Test 3	614	RST Test

Table D.3. Ethernet Tests

Test	*Max. Hits*	*Test Description*
Test 0	207	BYTE SELECTS ON 3 32-BIT READ-WRITE REGISTERS (VARIOUS BUS DELAYS)
Test 1	7,808	'WALKING ONE' WITH SINGLE CYCLES ACROSS MAC REGISTERS (VARIOUS BUS DELAYS)
Test 2	169,352	'WALKING ONE' WITH SINGLE CYCLES ACROSS MAC BUFFER DESC. (VARIOUS BUS DELAYS)
Test 3	544	MAX REG. VALUES AND REG. VALUES AFTER WRITING INVERSE RESET VALUES AND HARD RESET OF THE MAC
Test 4	8,206	BUFFER DESC. RAM PRESERVING VALUES AFTER HARD RESET OF THE MAC AND RESETING THE LOGIC
Test 5	102,912	TRANSMIT PACKETS (NO PADs) FROM 0 TO (MINFL - 1) SIZES AT 8 TX BD (10Mbps)
Test 6	15,829	TRANSMIT PACKETS (NO PADs) FROM 0 TO (MINFL - 1) SIZES AT 8 TX BD (100Mbps)
Test 7	147,944	RECEIVE PACKETS FROM MINFL TO MAXFL SIZES AT MAX RX BDs (100Mbps)
Test 8	114,290	RECEIVE PACKETS FROM 0 TO (MINFL + 12) SIZES AT 8 RX BD (10Mbps)
Test 9	15,675	RECEIVE PACKETS FROM 0 TO (MINFL + 12) SIZES AT 8 RX BD (100Mbps)

Table D.4. Memory Controller Tests

Test	*Max. Hits*	*Test Description*
Test 0	1886	SDRAM PARITY TEST
Test 1	30,180	SDRAM SEQUENTIAL ACCESS TEST
Test 2	12,695	SDRAM RANDOM ACCESS TEST
Test 3	24,830	SDRAM SEQUENTIAL ACCESS READ-MODIFY-WRITE TEST
Test 4	25,106	SDRAM RANDOM ACCESS READ-MODIFY-WRITE TEST

(Continued on next page)

Table D.4. Memory Controller Tests (continued)

Test	*Max. Hits*	*Test Description*
Test 5	214,424	SDRAM block copy TEST-1
Test 6	97,557	Asynchronous memory back-2-back test
Test 7	94,732	SSRAM SEQUENTIAL ACCESS TEST

Table D.5. Serial Peripheral Interface (SPI) Tests

Test	*Max. Hits*	*Test Description*
Test 0	10	Program Registers
Test 1	8	Verify Registers
Test 2	29	Transfer: 8 bit, MSB first, TX negedge, RX posedge
Test 3	44	Transfer: 16 bit, LSB first, TX negedge, RX posedge
Test 4	145	Transfer: 64 bit, LSB first, TX posedge, RX negedge
Test 5	284	Transfer: 128 bit, MSB first, TX posedge, RX negedge
Test 6	73	Transfer: 32 bit, MSB first, TX negedge, RX posedge, Interrupt Enabled
Test 7	73	Transfer: 32 bit, MSB first, TX posedge, RX negedge, Interrupt Enabled, Automatic Slave Select
Test 8	14	Transfer: 1 bit, MSB first, TX posedge, RX negedge, Interrupt Enabled, Automatic Slave Select

Table D.6. PCI/Wishbone Bridge Tests

Test	*Max. Hits*	*Test Description*
Test 0	738	Test Initial All Conf Values
Test 1	20073	Test Wishbone Image 1
Test 2	20073	Test Wishbone Image 2
Test 3	54	Wishbone Slave Errors
Test 4	882	Wishbone to PCI Error Handling
Test 5	900	Parity Checking

(Continued on next page)

Table D.6. PCI/WB Bridge Tests (continued)

Test	*Max. Hits*	*Test Description*
Test 6	1868	Wishbone to PCI Transactions
Test 7	16240	Test Master Overload
Test 8	27	Configure Bridge Target Base Addresses
Test 9	26	Test Conf Cycle Type1 Reference
Test 10	1198	Test PCI Image
Test 11	108	Test Wishbone Error Read
Test 12	112	Target Fast Back-to-Back
Test 13	211	Target Disconnects
Test 14	22506	Test Target Overload
Test 15	184	Test Target Abort
Test 16	572	Transaction Ordering

Acronyms

AC97	Audio Codec '97, 5, 63, 66, 78, 87
APR	Active Process Ratio, 118–120
ARM	Advanced RISC Machines, 11
ASIC	Application-Specific Integrated Circuit, 21–24, 144
ATA	Advanced Technology Attachment, 5, 63, 66, 78, 88
BEE	Berkeley Emulation Engine, 19
BEE2	Berkeley Emulation Engine II, 20
BSP	Board Support Package, 95
CAD	Computer Aided Design, 2
CIS	Common Instruction Set, 23, 104, 108, 112, 134
CLB	Configurable Logic Block, 21
CPU	Central Processing Unit, 67
CRP	Canonical RTL Process, 57–65, 68
DDR	Double Data Rate, 20, 93, 94, 98, 99
DIP	Dual Inline Package, 93
DISC	Dynamic Instruction Set Computer, 11, 12
DLL	Delay-Locked Loop, 9
DSP	Digital Signal Processor, 136
EDA	Electronic Design Automation, 21
EDK	Embedded Development Kit, 91, 93, 100
EOT	Even-Odd Transposition, 103, 106, 111, 128, 132
ESL	Electronic System Level, 13
FIFO	First-In, First-Out, 77, 98

FPGA Field Programmable Gate Array, 3, 4, 6–12, 14, 18–24, 29, 30, 44–48, 91–93, 97, 98, 100, 102, 106, 109, 124, 125, 129, 131, 133, 135–136
FSM Finite State Machine, 5, 29–39, 43–45, 47, 48, 133

GHz Gigahertz, 10^9 cycles per second, 128

HDL Hardware Description Language, 8, 14, 15, 18, 19, 21, 51–53, 55–60, 63, 66, 144–146

ICAP Internal Configuration Access Port, 11, 92–94, 97, 98, 100, 101, 109, 110, 126, 127
IEEE Institute of Electrical and Electronics Engineers, 14, 16, 52, 146
IP Intellectual Property, 6
ISS Instruction Set Simulator, 1

JIT Just In Time, 3, 65, 92, 106, 118, 135
JTAG Joint Test Action Group, 11

LED Light Emitting Diode, 93
LRM Language Reference Manual, 16, 62
LUT Look-Up Table, 9, 10, 98, 100, 106, 109

MAC Media Access Control(ler), 5, 63, 66–67, 70, 76, 77
MHz Megahertz, 10^6 cycles per second, 95, 101, 125–128

OPB On-chip Peripheral Bus, 93, 94, 98, 99, 102

PCI Peripheral Component Interconnect, 5, 19, 21, 66, 75, 77, 84–86
PE Processing Element, 20, 21
PLI Progamming Language Interface, 62, 66–68, 72–74
PLL Phase-Locked Loop, 9
PRC Parallel Route Calculation, 115
PRISM Processor Reconfiguration through Instruction-Set Metamorphosis, 11

RAM Random Access Memory, 93, 94, 98, 100, 106, 109

RAMP Research Accelerator for Multiple Processors, 20
RISC Reduced Instruction Set Computer, 11, 95
RM Real Machine, 23–26, 28, 50, 76, 92–100, 102, 104–132, 134, 135
RTL Register Transfer Level, 2–8, 12–15, 17–18, 20, 23, 24, 43, 51, 54–57, 59, 65, 68, 72, 73, 75, 78, 79, 91, 93, 103, 117, 133–137
RTR Run-Time Reconfiguration, 3, 4, 97, 98, 102

SMR Simulation-cycle-to-Migration Ratio, 108, 113, 129, 131
SoC System on Chip, 6, 13, 18, 72
SPI Serial Peripheral Interface, 5, 66, 79, 89, 90
SRAM Synchronous Static Random Access Memory, 9, 100, 126, 127

TBC Time Base Counter, 95
TLM Transaction-Level Model, 2, 68
TSC Time Stamp Counter, 67, 75, 128

UART Universal Asynchronous Receiver/Transmitter, 93, 94
USB Universal Serial Bus, 18

VHDL VHSIC Hardware Description Language, 5, 14–16, 19, 51–53, 56, 60–62, 92, 103, 104, 106, 144
VHSIC Very High Speed Integrated Circuit, 168
VLIW Very Long Instruction Word, 20, 21
VM Virtual Machine, 23, 24, 26, 91–93, 95, 97, 104–106, 108, 111, 112, 114, 115, 117, 118, 124–126, 128, 129, 131, 132, 135

XDL Xilinx Design Language, 92
XPS Xilinx Platform Studio, 93, 95, 100

Colophon

This document was typeset primarily in the "Nimbus Roman No9 L" and "Nimbus Sans L" fonts using the LaTeX document preparation system, specifically `pdflatex` of the teTeX distribution, version 3.0. LaTeX is a system that may be extended using packages, and the following packages were used within this document:

- fontenc
- textcomp
- url
- pslatex
- fancyhdr
- amsmath
- amssymb
- amsthm
- tocloft
- graphicx
- indentfirst
- hyphenat
- wasysym
- setspace
- caption
- geometry
- color
- hyperref
- lettrine
- enumerate
- listings
- calc
- bibunits
- multirow
- array
- hhline
- longtable
- glossary[1]
- bsheaders

Information about each of these packages can be found on the Comprehensive TeX Archive Network, www.ctan.org.

The `hyperref` package provides the most useful functionality by defining hyper links within the document. Each of the acronyms is a link to the Acronyms section which defines them, and each citation is a link to the Bibliography.

The illustrations were done with the Inkscape Vector Illustrator, version 0.45.1 (www.inkscape.org). The graphs were created with Gnuplot, version 4.2 (www.gnuplot.info).

[1] Because this package uses `\thepage` macro, it sometimes gives incorrect page numbers, despite the author's objections to the contrary. The newer `glossaries` package was not ready at the time of writing.

Vita

Aric David Blumer was born in 1970 in Flint, Michigan. He received a bachelor of science in engineering science/physics from Bob Jones University in Greenville, South Carolina. Continuing his education at Clemson University in Clemson, South Carolina, he was awarded a master of science in computer engineering in 1994. He accepted a position in Warrendale, Pennsylvania, at FORE Systems, Inc., in the ATM Adapter design group as a Diagnostic Engineer, where he gained experience in embedded programming and the post-production verification of ASICs and printed circuit boards. His responsibilities later transitioned to ASIC pre-production testing and ASIC design. FORE Systems was acquired by Marconi, plc, and after ten years of combined service at these companies, he resigned to pursue a Ph.D. in computer engineering from Virginia Tech as a College of Engineering Fellow and Bradley Fellow. He also helped to found SDG Systems, LLC, with his cousin Todd Blumer. SDG specializes in driver development and operating system porting, and Aric continues to assist SDG on a consultation basis.