Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 369040, 10 pages
doi:10.1155/2008/369040

Research Article

Exploiting Process Locality of Reference in RTL Simulation Acceleration

Aric D. Blumer and Cameron D. Patterson

Virgina Polytechnic Institute and State University, Blacksburg, VA 24061, USA

Correspondence should be addressed to Aric D. Blumer, aric@vt.edu

Received 1 June 2007; Revised 5 October 2007; Accepted 4 December 2007

Recommended by Toomas Plaks

With the increased size and complexity of digital designs, the time required to simulate them has also increased. Traditional simulation accelerators utilize FPGAs in a static configuration, but this paper presents an analysis of six register transfer level (RTL) code bases showing that only a subset of the simulation processes is executing at any given time, a quality called *executive locality of reference*. The efficiency of acceleration hardware can be improved when it is used as a process cache. Run-time adaptations are made to ensure that acceleration resources are not wasted on idle processes, and these adaptations may be affected through process migration between software and hardware. An implementation of an embedded, FPGA-based migration system is described, and empirical data are obtained for use in mathematical and algorithmic modeling of more complex acceleration systems.

1. INTRODUCTION

The capacity of integrated circuits has increased significantly in the last decades, following a trend known as "Moore's Law" [1]. The code bases describing these circuits have likewise become larger, and design engineers are stressed by the opposing requirements of short 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 are the large, nonrecurring expenses of producing or correcting an integrated circuit, 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" [4, 5].

As in the software industry, the efforts to mitigate the effect of growing hardware complexity have resulted in the use of greater levels of abstraction in integrated circuit descriptions. The most common level of abstraction used for circuit descriptions is the register transfer level (RTL). RTL code represents the behavior of an integrated circuit through the use of high-level expressions and assignments that infer logic gates and flip-flops in a fabricated chip. While the synthesis of descriptions above RTL is improving, it is not yet as efficient as human-coded RTL [6, 7], so RTL still forms a necessary part of current design flows.

While simulating circuit descriptions at higher levels of abstraction is faster than RTL simulation, another method of improving verification times is the hardware acceleration of RTL code. The traditional approach to hardware-based simulation acceleration is to use an array of field programmable gate arrays (FPGAs) to emulate the device under test [8]. The RTL code of the device is mapped to the FPGAs causing a significant increase in execution speed. There are, however, two drawbacks to this approach. First, the RTL code must be mapped to the hardware before the simulation can begin. Second, once the mapping is complete, it is static. Because it is static, the speedup is directly related to the amount of the design that fits into the acceleration hardware. For example, if only one half of the design can be accelerated, Amdahl's law states that the maximum theoretical speedup is two. Enough hardware must be purchased then to contain most if not all of the device under test, but hardware acceleration systems are expensive [8].

FPGA smay be configured many times with different functionality, but they are designed to be statically configured.

That is, the power-up or reset configuration of the FPGA can be changed, but the configuration remains static while the device is running. This is the reason that existing accelerators statically map the design to the FPGA array. It is possible, however, to change the configuration of an FPGA at run time, and this process is aptly termed *run-time reconfiguration* (RTR). 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. The system can be tuned by migrating busy processes into and idle processes out of the FPGA and establishing communication routes between processes 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, certain time-consuming or often-executed processes can be synthesized directly to hardware for further speed and area efficiency.

This paper presents the research done in the acceleration of RTL simulations using process migration between hardware and software. Section 2 gives an overview of existing acceleration methods and systems. Section 3 develops the properties that RTL code must exhibit in order for run-time mapping to acceleration hardware to be beneficial. Section 4 presents the results of profiling six sets of RTL code to determine if they have exploitable properties. Section 5 describes an implementation of an embedded simulator that utilizes process migration. Section 6 develops an algorithmic model based on empirical measurements of the implementation of Section 5. Section 7 concludes the paper.

2. EXISTING ACCELERATION TECHNOLOGY

Much progress has been made in the area of hardware acceleration with several commercial offerings in existence. Cadence Design Systems (Calif, USA) offers the Palladium emulators [9] and the Xtreme Server [10], 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 and hardware 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 [11], while not specifically mentioning Palladium, describes a system to accelerate event-driven simulations by providing low-overhead communication 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. The behavioral code is compiled for a local microprocessor run alongside the 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 [12]. 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.

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 [13]. Each processing element (PE) is a two-input lookup table (LUT) 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. The PEs are compiled into an FPGA which is controlled by a personal computer host through a PCI bus. The host schedules VLIWs to the FPGA, and each instruction word contains opcodes and addresses for all the PEs. On every VLIW execution, 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, resulting in a gate-level simulation rather than RTL-level. An EDA startup company, Liga Systems, Inc., (Calif, USA) has been founded to take advantage of this technology [14].

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. The next section presents a property of RTL code that can be exploited to reduce the amount of hardware that is required.

3. EXECUTIVE LOCALITY OF REFERENCE

An RTL simulation consists of a group of processes that mimic the behavior of parallel circuitry. Typical RTL simulation methods view processes as a sequence of statements that calculate the process outputs from the process inputs when a trigger occurs. The inputs are generally expressed as signal values in the right-hand side of expressions or in branching conditionals. The trigger, which does not necessarily contain the inputs, is expressed as a *sensitivity list*, which in synchronous designs is often the rising edge of a clock, but it may be a list of any number of *events*. These events determine when a process is to be run, and thus the simulation method is called *event-driven simulation* [15]. When all the events at the current simulation time are serviced, the simulator advances to the time of the next

event. Every advance marks the end of a *simulation cycle* (abbreviated as "simcycle"). In synchronous designs, the outputs of synchronous process can only change on a clock edge, so *cycle-based simulation* can improve simulation times by only executing processes on clock edges [16].

The two most common languages used for RTL coding are Verilog and VHDL, and in both cases, during a simulation cycle, the processes do not communicate while they are executing. Rather, the simulator propagates outputs to inputs between process executions [17, 18]. Furthermore, the language standards specify that the active processes may be executed in any order [17, 18], including simultaneous. It is, Through this simultaneous execution of processes, hardware accelerators provide a simulation speedup. Traditional hardware accelerators place both idle and active processes in the hardware with a static mapping, and the use of the accelerator is less efficient as a result.

A simulator that seeks to map only active processes to an accelerator at run time relies fundamentally on the ability to delineate between active and idle processes. Locality of reference is a term used to describe properties of data that can be exploited by caches [19]. Temporal locality indicates that an accessed data item will be accessed again in the near future. Spatial locality indicates that items near an accessed item will likely be accessed as well. These properties are exploited by keeping often-accessed data items in a cache along with their nearest neighbors. For executing processes, there is *executive locality of reference*. Temporal executive locality is the property that an executing process will be executed again in the near future. Spatial executive locality is the property that processes receiving input from an executing process will also be executed. These properties can be exploited by keeping active processes in a parallel execution cache.

Spatial executive locality is easily determined in RTL code using the "fanout" or dependency list of a process's output, a structure a simulator already maintains. Temporal executive locality, however, is not quite so obvious. Take, for instance, a synchronous design in which every process is triggered by the rising edge of a clock. An unoptimized cycle-based simulator will run every process at every rising edge of the clock, so every process is always active. If, however, no inputs to a process have changed from the previous cycle, then the output of the process will not change. Therefore, it does not need to be run again until the inputs do change. During this period, the process is idle. This is the fundamental characteristic on which temporal locality depends, and detecting changes on the inputs can be accomplished with an *input monitor*.

Input monitoring has been used to accelerate the software execution of gate simulations (a technique often called "clock suppression") [20], but it has not been used to determine how hardware acceleration resources should be used. Whether the inputs change or not is readily determinable by a simulator, but the question remains whether activity of the processes varies enough to be exploitable. Memory caches are inefficient if all the cachable memory is always accessed. Similarly, an execution cache is inefficient if all the processes are always active. Even though the processes exhibit good temporal locality, the effect of process caching is diminished in this case. But are RTL processes always active during a simulation? The next section addresses that question.

```
(1) always   @B   $check_sig(21, 0, B);
(2) always   @C   $check_sig(21, 1, C);
(3) always   @ (posedge  CLK);
(4)        if ($shall_run(21));
(5)             A <= B + C;
```

Algorithm 1: Input monitor instrumentation.

4. PROFILING AND RESULTS

The RTL code of six Verilog designs from OpenCores [21] was instrumented and profiled using Cadence's LDV 5.1. Since the source code for LDV's ncsim is not available, every input of the `always` blocks is monitored with a PLI function call. The previous value of the inputs are kept, and the process is run only if at least one of the inputs changes from the previous cycle. A sequence of code demonstrating the method is shown in Algorithm 1. Lines (1), (2), and (4) were added for this study, but RTL code will require no modifications if this functionality is part of the simulator itself. The arguments to `$check_sig`() are the process number, the signal number for this process, and the signal's value. Each process is assigned a unique number, and multiple instantiations of the same process are also delineated. If there are any changes detected by `$check_sig`() for a process, then the next call of `$shall_run`() returns 1. Otherwise, it returns 0, and the code is not run. With this instrumentation, the test benches of the designs were run to verify correct functionality.

The code that was analyzed included a PCI bridge, Ethernet MAC, memory controller, AC97 controller, ATA disk controller, and a serial peripheral interface (SPI). They represent a range of functionality and code sizes (shown in Figure 1, excluding the test bench). Activity data were recorded on a per-test basis, and the metrics used for evaluation are the activity ratio (AR) of each process and the number of lines of code in each process. The AR, expressed as a percentage, is the ratio of the number of times a process is executed to the number of times it was triggered. In other words, it is the ratio of the number of times `$shall_run`() returns 1 to the number of times it is called. The AR, however, does not show the entire picture. If the inactive processes are short sequences of code, then the bulk of the design is still active. Figure 2 shows a representative example of the activity ratios and the line counts of the PCI controller for a single test. These graphs show that a large number of the processes are inactive for the test, a demonstration of temporal executive locality. The graph of the line counts per process shows that the idle processes do comprise a significant part of the design. Note that this graph's Y-axis is limited to show the detail of the majority of the processes;

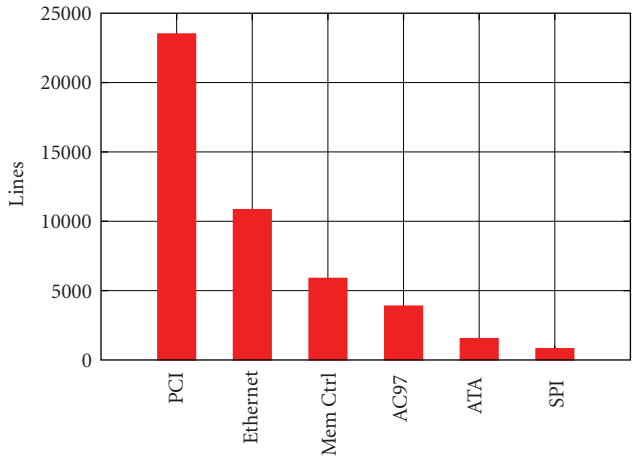

Figure 1: Sizes of profiled code.

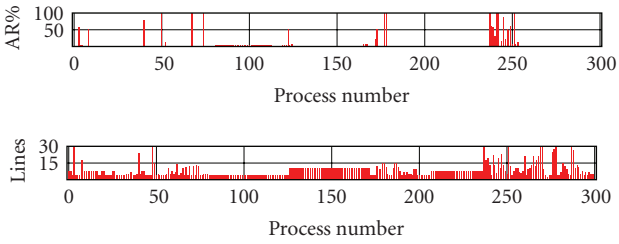

Figure 2: Process activity ratios and line counts for the PCI controller.

several are larger than 30 lines. It is inefficient to place these idle processes into acceleration resources.

The number of lines of HDL code is not a precise indication of the actual run-time size of a process, but it does provide a good estimate. A simulation system would likely use metrics such as compiled code size or actual process run times. Nevertheless, using data such as that shown in Figure 2, it can be determined what percentage of the total lines of code are active during a test. Whether a process is considered active or not is determined by an AR threshold. For example, one can determine how many lines of code there are in all the processes that have an AR above 10%. This is called the *activity footprint*. The smaller the activity footprint, the less hardware is required to accelerate it. The activity footprints for a number of tests are shown in Figure 3 with AR thresholds of 1%, 10%, 20%, and 30%. A system can vary the AR threshold until the activity footprint fits within the acceleration hardware.

The PCI code shows a small activity footprint for the seventeen tests shown. Tests 8 and 9 are PCI target tests, and they show the smallest footprints of 17.9% with a 1% AR threshold. Therefore, for these tests, only enough acceleration hardware is required to hold 17.9% of the process code. Test 16, a bus transaction ordering test, shows the largest footprint, exceeding 50% in all cases. For almost all the tests, the 10% AR threshold yields a footprint less than 50%. The Ethernet code has larger activity footprints in general, but the footprint varies from approximately 29%

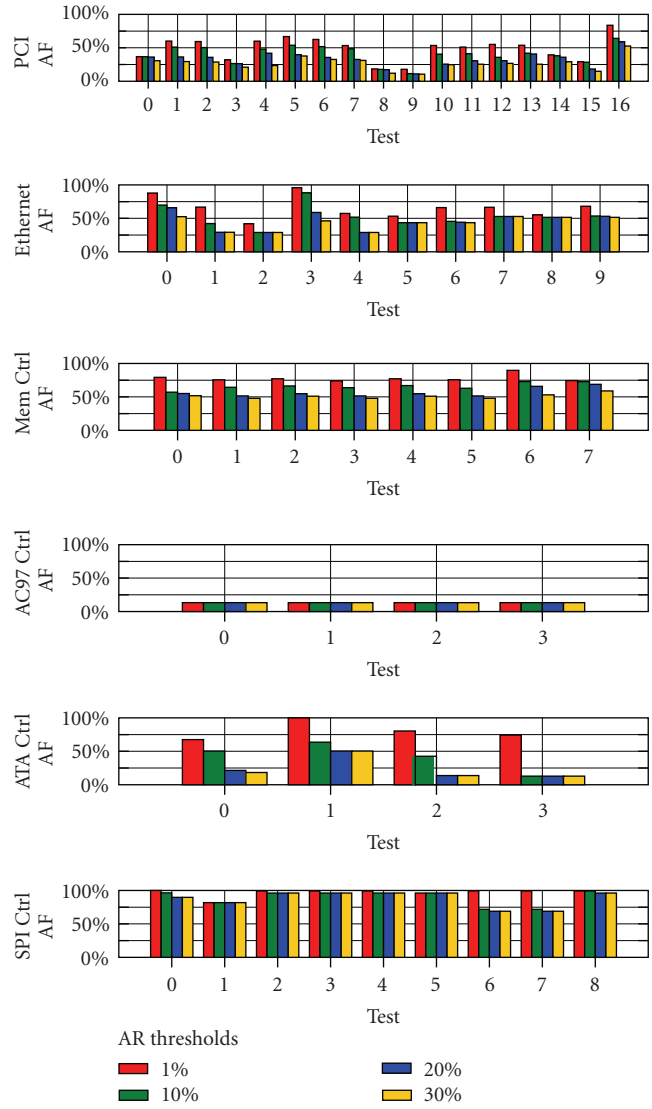

Figure 3: Activity footprints for each design.

for Test 2 (a walking-ones register test) to 88% for Test 3 (a register reset test) when the AR threshold is 10%. The higher values for Test 3 are partly attributable to the fact that the test is relatively short. Even so, the Ethernet code exhibits footprints of approximately 50% or less for a large number of the tests when the AR threshold is 10%.

The behavior of the memory controller is comparable with the Ethernet controller. Its characteristics are largely attributable to a single-state machine process that is 789 lines of code and is always active. In contrast, the AC97 controller has a small activity footprint of just over 13% regardless of the AR threshold. A small percentage of its code handles data flow which the majority of each test exercises.

The ATA and SPI code bases are relatively small with less than 25 processes each. Despite the small code size, they do demonstrate some exploitable locality. In the case of the ATA controller, an AR threshold of 10% gives results that are comparable to the other designs. However, the SPI device,

which is the smallest, has rather large footprints; but these footprints are a percentage of a small code base. It is intuitive that as designs get smaller, they exhibit less exploitable locality because the design is not large enough to have independent code that is idle during a test. These smaller designs would generally not be candidates for hardware acceleration, but acceleration may still prove useful for a large number of small devices such as those found in a system on a chip (SoC) design.

The amount of data gathered from the profiling is significant, and only a portion is shown here. These results, however, show that executive locality of reference does exist in these OpenCores designs, whose code demonstrates various functionality and code size. RTL code is normally proprietary and unavailable to researchers for analysis, but by inference, one would be expected executive locality of reference to be a general rule rather than an exception demonstrated by these particular designs. The remainder of this paper shows how this locality can be exploited.

5. IMPLEMENTING PROCESS MIGRATION

A process migration system consists of communicating processes, processors, and a communication infrastructure. If the processes are to be run in either software or hardware contexts, they must be in a form that is portable. A common instruction set (CIS) serves that purpose. Processes compiled to a CIS can then be executed in a system composed of simple virtual machines (VMs) and real machines (RMs) as shown in Figure 4. A set of processes begins running entirely in VMs, one per process. The use of VMs allows the system to begin execution immediately, and it monitors the activity of the processes to determine which processes should be migrated to the RMs or synthesized directly to the hardware. The algorithms used to determine when and where to migrate processes is left to future work. If there are more active processes that do not fit in the hardware, they can be compiled to native code in the VMs. Idle processes in VMs will not be run and can therefore be left in CIS form until they become active. Also note that the state of any process is available at any time for debugging.

5.1. Implementation details

The infrastructure required to implement a complete simulator would likely 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.

To this end, a process migration system was implemented on an XUPV2P board developed for the Xilinx University Program [22]. The board is populated with a Virtex-II Pro FPGA (XC2VP30) and 512 megabytes of DDR memory. The XC2VP30 contains two PowerPC 405 processors along

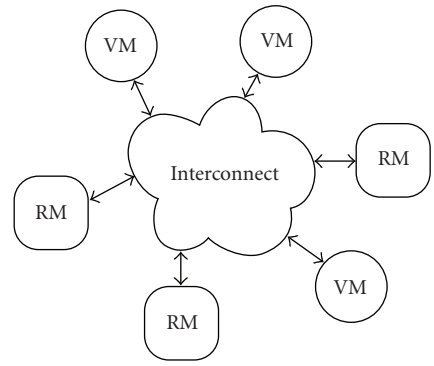

Figure 4: The VMs and RMs execute processes collaboratively. The RMs execute in parallel.

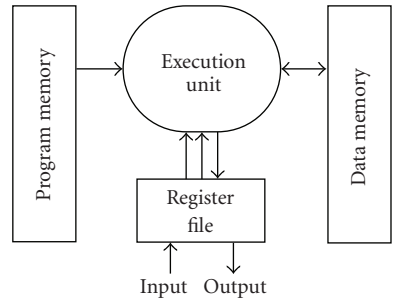

Figure 5: VM and RM block diagram.

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 processors. A VM-based cycle simulator runs the RTL simulation on one PowerPC, and it migrates the state of the VMs to and from the RMs as needed.

The internal structure of both the VMs and RMs is shown in Figure 5. Migration is the transfer of the state in program memory, data memory, and the 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 as follows. (1) The use of only RMs is the first step to a complete migration system that also uses runtime synthesis of processes directly to hardware. It is more efficient to measure migration overheads first with a simpler RM-only system before committing to further work. (2) The additional logic required for migration without RTR would increase the size of the infrastructure, reducing the number of RMs and making timing closure more difficult.

The original RMs were fully pipelined using flip-flops for the register file, but Virtex-IIs do not provide a way to update flip-flop state on a per-RM basis. The `GRESTORE` configuration command updates all flip-flops in the FPGA including critical infrastructure [23]. A solution is to use LUT-distributed RAM instead. LUT-based RAMs can be updated through the ICAP on a per-machine basis, but they cannot provide the single-in-double-out interface required

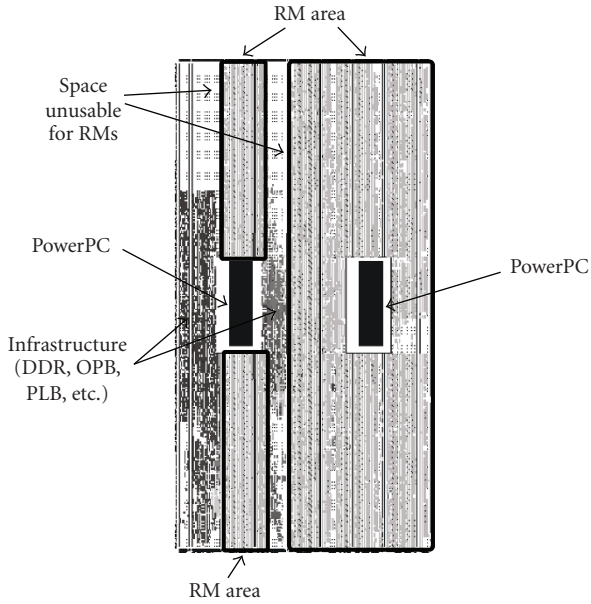

Figure 6: Device floor plan of the migration system.

by full pipelining without doubling the memories. We elected to execute an instruction every two clock cycles in a nonpipelined fashion, and the absence of data hazards makes the RMs smaller.

There is another limitation of LUT RAMs, however, that must be considered. A frame is the minimum configuration unit in an FPGA. In Virtex-II FPGAs, frames span the entire height of the device [23, page 337], and some of the logic used to load the frames is used 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 that use LUT RAMs: just reading the frames used by them during run time causes a malfunction. Careful floorplanning is required to ensure that the RMs do not share any frames with this infrastructure. This issue does not affect the RMs themselves, however, because migration to and from RMs only occurs between simulation cycles when the RMs are idle. Each RM is constrained to occupy a 16×16 slice area, and they are floorplanned into columns with other frames reserved entirely for infrastructure. 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 6.

5.2. The simulator application

The even-odd transposition (EOT) sort [24] is a good application to measure the performance of the simulator because it allows the number of RMs and the number of simulation cycles to be varied orthogonally while still giving correct results. Each iteration of the EOT sort is a swapping—or transposition—of the numbers to place the greater number on the right. When executed in parallel, it requires a maximum of n cycles to sort n numbers when there are $n/2$ transpositions per 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 cycles. In an RTL simulator, this code would be compiled into one process for each instance (barring optimization), each running the same code.

The VHDL was translated into the CIS, which is executable in either VMs or RMs. In this implementation, the CIS is much like typical RISC assembly languages, but the CIS is an abstraction layer that may vary across migration system implementations depending on the possible migration destinations. To make the RMs as small as possible, they have 16-bit instructions and 16-bit data. Each RM also has eight 16-bit registers with some aliased to inputs and outputs.

The PowerPC in the Virtex-II Pro operates at 300 MHz while the remaining logic operates at 100 MHz. The PowerPC executes the VMs and the simulation infrastructure with the "standalone" software package provided in Xilinx's EDK version 7.1i. Inputs and outputs of the RMs are connected to the on-chip peripheral bus (OPB) through a mailbox. Between simulation cycles, the simulator reads the outputs of the RMs and writes their inputs, a process called *software connectivity*. On the other hand, *hardware connectivity* is the joining of RM outputs to RM inputs directly in the FPGA. To measure the difference in overhead between the two types, a number of tests were run with hardware connectivity written into RTL code. A complete system would instead implement these connections at run time, and that behavior is modeled in Section 6.2.

5.3. Results for empirical modeling

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 were also done for performance measurements.

To evaluate pure parallel performance, the speedups with no migration were measured for both software and hardware connectivity. Figure 7 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 speedup is the run time without RMs (T_0) divided by the run time with RMs (T_r) [19]. One result of executing code in VMs without native just-in-time (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 1,005 when using hardware connectivity with 35 RMs. To illustrate how this occurs, consider a serial program that executes 10 tasks, each taking 1 second. If those tasks are parallelized on 10 processors, they all execute in 1 second for a speedup of 10. If, however, the parallel processors are ten times faster than the serial processor, the tasks complete in 0.1 second for a speedup of 100. Note also that the relatively low speedups on the left side of the plots demonstrate

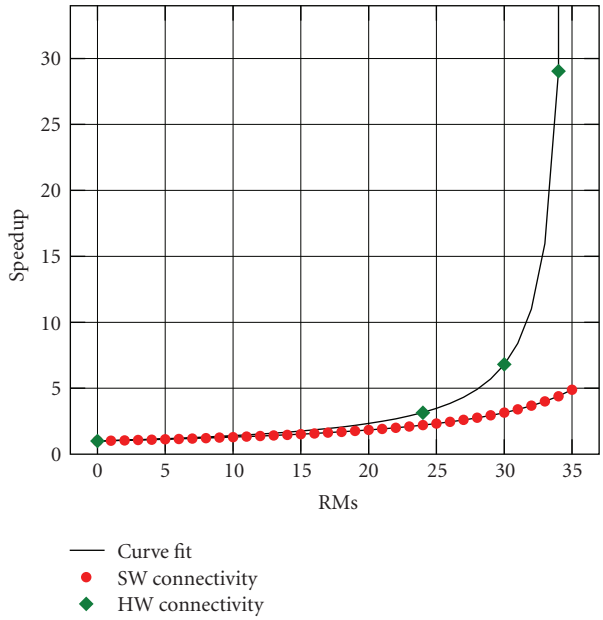

Figure 7: Speedup of HW and SW connectivity (35 simcycles).

Table 1: Modeling parameters.

Param.	Description	Value
C	Simcycles	*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 $\frac{\text{ms}}{\text{migration}}$

Amdahl's law, a behavior exhibited by all parallel systems, and is not unique to this system. Figure 7 shows clearly that hardware connectivity is a necessary improvement. These direct connections could be accomplished with a reconfigurable crossbar switch, network on a chip, or run-time routing.

The time required to execute the sorting simulation with software connectivity is

$$T(r) = C(t_o + rt_r + [P - r]t_v) + rMt_m, \quad (1)$$

where t_o, t_r, and so forth are described in Table 1. The sort tests use $C = P = 35$, while r varies from 0 to 35; and 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 microsecond. 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.

Migration overhead, t_m, is affected by 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 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 [25, page 4]. Multiple frames must be read and written to migrate state to a single RM, and the unoptimized migration time per machine was measured to exceed 220 milliseconds. However, the full-height span of frames can be exploited. The RMs were floorplanned into columns of 8, except for the rightmost column which contains 11. For each RM, an RLOC constraint places the program, data, and register file memories into the same frames. This optimization reduces the number of frames containing memories from 65 to 16. 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 reduced the average migration time from 220 milliseconds to 6.0 milliseconds, and the maximum time for a single migration—which frame caching does not help—is 18.9 milliseconds.

Speedup is defined as $S = T_0/T(r)$, which can be modeled as follows for hardware connectivity, where T_0 is the measured time without RMs:

$$S = \begin{cases} \dfrac{T_0}{C(t_o + Pt_v)} & \text{for } r = 0, \\ \dfrac{T_0}{C(t_o + t_r + [P - r]t_v) + rMt_m} & \text{for } 0 < r < P, \\ \dfrac{T_0}{Ct_o + rMt_m} & \text{for } r = P. \end{cases} \quad (2)$$

The hardware connectivity execution time is calculated differently than software (1) because processes in RMs with direct connections cause no overhead in the simulator. Equation (2) is piecewise because there are no RM overheads t_r and t_m when zero RMs are in use. 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 and migration overhead.

As mentioned previously, the EOT sort gives correct results for runs longer than 35 simulation cycles. Figure 8 shows speedup plots for 35, 1024, and 10240 simulation cycles, including all migration overheads for one migration per RM ($M = 1$), along with plots of (2). These simulation lengths are reasonable based on the test benches of the OpenCores code, which contain simulations ranging from dozens to millions of simulation cycles per test. Larger simcycle-to-migration ratios (SMRs) amortize the migration overhead over a longer period of time, improving the speedup. For small SMRs (e.g., 35 : 1), the migration time exceeds the simulation run time, and the system exhibits a

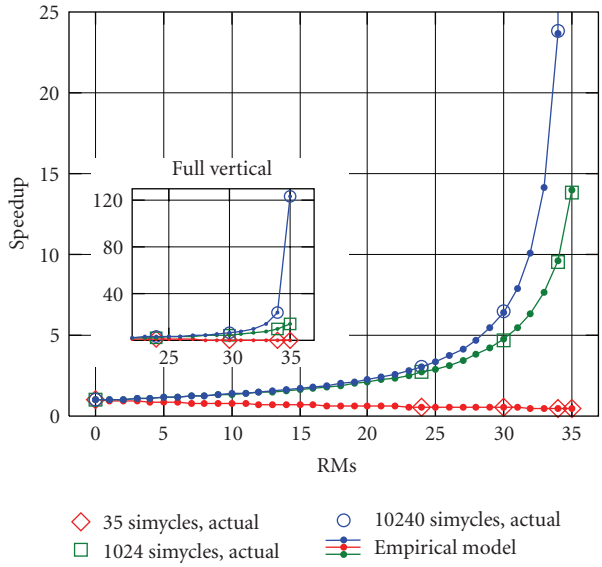

Figure 8: HW connectivity speedups compared to the empirical model.

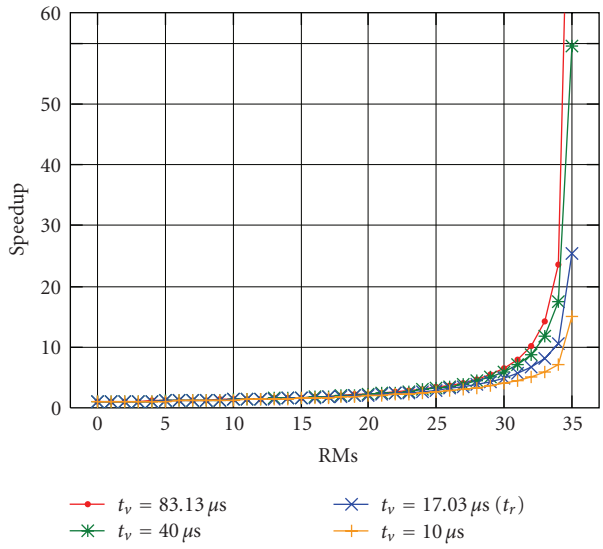

Figure 9: Modeled effect of t_v on speedup (10 k simcycles).

speedup less than one. Using the measured speedups with (2), the Marquardt-Levenberg curve fitting algorithm solves for t_r, t_v, and t_m as shown in Table 1. t_o was calculated directly from the third case of (2) when $M = 0$.

No complete system would run CIS code in VMs without compiling it to native code, so (2) is used to explore the effect of VM-related run times (t_v). Figure 9 shows that speedups of 15 are possible even if VM execution is faster than RM execution ($t_v < t_r$). Since t_v includes the time to execute a process and to update process inputs and outputs, it is unlikely to be less than t_r.

6. ALGORITHMIC MODELING

A mathematical model based upon the empirical data obtained in Section 5 is useful when simplifying assumptions are made about the behavior of the system. Since an implementation that can simulate larger code bases would require a significant investment in infrastructure such as JIT compilers and run-time routers, a more complex model is required to better understand the behavior of complete systems. Some behavior, such as the effect of frame caching, is difficult to model with mathematical formulas. This section explains an algorithmic model based on (2) that is executed as a C program, allowing fixed parameters to be replaced with functions. To ensure that the model reflects the behavior of the system, the graph of Figure 8 was reproduced using the model, and the average percent error for the 1024- and 10240-simcycle plots from the measured speedups is less than 1%. For the 35 simulation cycle case, the average percent error is 2.1%. The subsequent modeling scenarios assume the pessimistic value of $t_v = 16$ microseconds and a migration time of $t_m = 12.5$ milliseconds (the average of 6 milliseconds and 18.9 milliseconds from Table 1).

6.1. Active process ratios

In Section 4, ARs of six designs were measured to demonstrate exploitable executive locality of reference. It is the activity ratio of the processes that identifies the best opportunity for acceleration, and an AR threshold determines which processes are to be migrated to the acceleration hardware. The ARs were measured on a per-process basis in the six designs, but for the purposes of modeling, process activity is specified as two sets of processes, those with a 100% AR, and those with a 0% AR. The ratio of the active processes to the total number of processes is the active process ratio (APR), and if the processes are assumed to be the same size, it is equivalent to the activity footprint. For example, an APR of 60% means that 60% of the processes are always active, and 40% of the processes are never active. In the *C* model, process activity is determined through a call to a function which returns zero for inactive processes and non-zero for active processes for every simulation cycle.

APRs of 17 : 35 and 30 : 35 are shown in Figure 10 with an APR of 1 : 1 as a baseline. This single graph demonstrates the benefit to speedup by exploiting executive locality. It shows that the lower the APR, the less acceleration hardware is required. The maximum speedup for each plot also shows that overhead is reduced because inactive processes are not migrated to hardware.

6.2. Route calculation

Another consideration in a migration system is the effect of establishing interconnect. It is difficult to estimate the time required to build connections between RMs at run time without a full implementation, but the graph of Figure 11 shows that if the time is long enough, the speedup can be less than one. This graph shows the route times in terms of

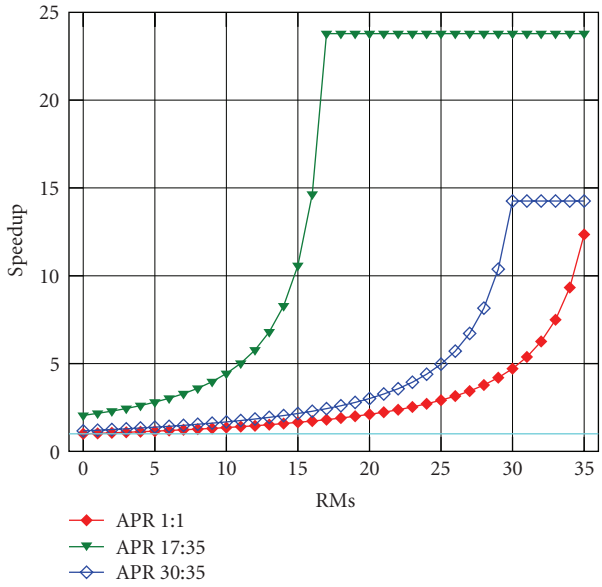

Figure 10: Modeled effect of APR on speedup.

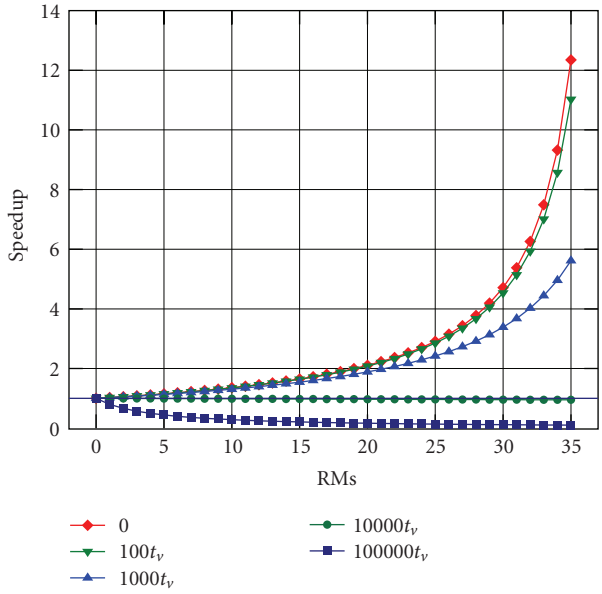

Figure 11: Serial route calculation (10 k simcycles).

t_v because the processor speed, which is reflected in t_v, has a direct effect on the route calculation time.

A possible optimization is to continue the simulation using software connectivity while the routes between RMs are calculated in parallel by another processor. (The implementation of Section 5.1 contains two PowerPC processors.) In the algorithmic model, the behavior is modeled by keeping track of the amount of time left until a route is complete. While the time remaining to complete a route is greater than zero, the model uses software connectivity. When the timer reaches zero, the model uses hardware connectivity. The results of such an algorithm are shown in Figure 12. The knees in the plots of the shorter simulations are due to

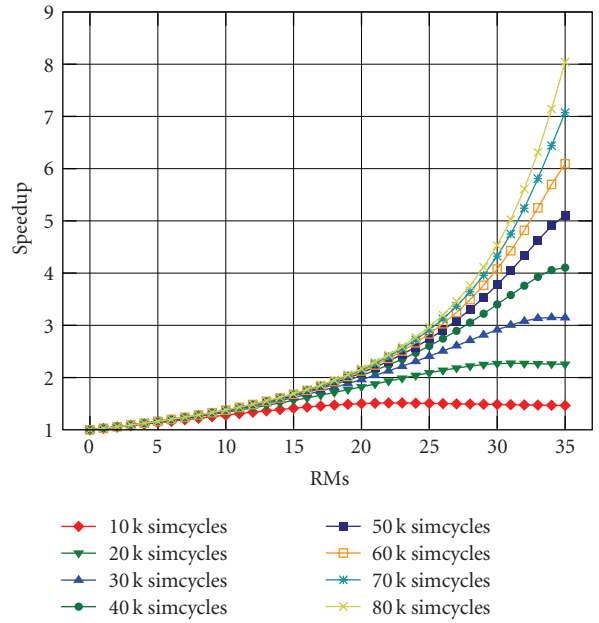

Figure 12: Parallel route calculation (10000 t_v).

insufficient time to complete the parallel route calculation. That is, the simulation ends before the route calculations are complete. Longer simulations allow the parallel route calculation to complete, and they benefit from the faster hardware connectivity.

7. CONCLUSION

We have shown that the RTL code of various designs demonstrates executive locality of reference which can be exploited by a process cache. The larger designs exhibit activity footprints below 50% in many cases and less than 75% in almost all cases. The smaller designs—SPI and ATA in particular—do not exhibit as much inactivity, but smaller designs are also less likely to require simulation acceleration. Traditional RTL accelerators require a mapping of the RTL code to the hardware prior to simulation, and that mapping remains static. Since many processes are idle, and since activity can change during simulation prior, static mapping to hardware does not maximize efficiency. Hardware/software process migration is one means of implementing a process cache to ensure that only active processes are accelerated.

Existing FPGAs, however, are not designed to be efficiently run-time reconfigurable; nevertheless an implementation using an existing FPGA allows us to determine and model the hurdles to efficient processes migration. The largest hurdles are the migration time due to reconfiguration and, potentially, the building of run-time routes. As run-time reconfiguration becomes more prevalent, and as the FPGA architectures are developed to improve reconfiguration performance, the migration time is expected to decrease. The potential bottleneck of run-time routing may also be overcome through the use of on-chip networks where communication occurs during simulator overheads.

Nevertheless, speedups of greater than ten are exhibited in many of the modeled scenarios. Further work is also warranted in comparing the trade-offs between a large number of small, communicating processors, and run-time synthesis of processes.

ACKNOWLEDGMENT

This work has been graciously funded through Virginia Tech College of Engineering and Bradley Fellowships.

REFERENCES

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

[2] S. Kakkar, "Proactive approach needed for verification crisis," EETimes, April 2004.

[3] A. Raynaud, "The new gate count: what is verification's real cost?" Electronic Design, October 2003.

[4] B. Bower, "The 'what and why' of TLM," EETimes, March 2006.

[5] P. Varhol, "Is software the new hardware?" EETimes, August 2006.

[6] R. Goering, "Tools ease transaction-level modeling," EETimes, January 2006.

[7] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: a high-end reconfigurable computing system," *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 114–125, 2005.

[8] G. D. Peterson, "Evaluating simulation acceleration techniques," in *Proceedings of the Enabling Technology for Simulation Science V*, vol. 4367 of *Proceedings of SPIE*, pp. 127–136, Orlando, Fla, USA, April 2001.

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

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

[11] J. Bauer, M. Bershteyn, I. Kaplan, and P. Vyedin, "A reconfigurable logic machine for fast event-driven simulation," in *Proceedings of 35th Design Automation Conference (DAC '98)*, pp. 668–671, San Francico, Calif, USA, June 1998.

[12] EVE Corporation, http://www.eve-team.com/.

[13] S. Cadambi, C. S. Mulpuri, and P. N. Ashar, "A fast, inexpensive and scalable hardware acceleration technique for functional simulation," in *Proceedings of 39th Design Automation Conference (DAC '02)*, pp. 570–575, ACM Press, New Orleans, La, USA, June 2002.

[14] R. Goering, "Startup liga promises to rev simulation," EETimes, 2006.

[15] R. D. Smith, *Simulation Article*, Encyclopedia of Computer Science, Nature Publishing, New York, NY, USA, 4th edition, 2000.

[16] D. Döhler, K. Hering, and W. G. Spruth, "Cycle-based simulation on loosely-coupled systems," in *Proceedings of the 11th Annual IEEE International ASIC Conference*, pp. 301–305, Rochester, NY, USA, September 1998.

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

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

[19] J. Hennesey and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 1990.

[20] R. Razdan, G. P. Bischoff, and E. G. Ulrich, "Clock suppression techniques for synchronous circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 10, pp. 1547–1556, 1993.

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

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

[23] XilinxInc, "Virtex-II Pro and Virtex-II Pro X FPGA User Guide," Xilinx, Inc., March 2005.

[24] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Addison Wesley, New York, NY, USA, 2nd edition, 2003.

[25] D. R. Curd, "Xapp660: dynamic reconfiguration of RocketIO MGT attributes," February 2004, http://www.xilinx.com/support/documentation/application_notes/xapp660.pdf.