Using Reconfigurability to Achieve Real-Time Profiling for Hardware/Software Codesign

Lesley Shannon and Paul Chow
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada M5S 3G4
{lesley, pc}@eecg.toronto.edu

ABSTRACT

Embedded systems combine a processor with dedicated logic to meet design specifications at a reasonable cost. The attempt to amalgamate two distinct design environments introduces many problems, one being how to partition a single design for the two platforms to achieve the best performance with the least effort. Since the latest FPGA technology allows the integration of soft or hard CPU cores with dedicated logic on a single chip, this presents new opportunities for addressing hardware/software codesign issues in the FPGA design process by utilizing the reconfigurable environment.

This paper introduces SnoopP, a non-intrusive, real time, profiling tool. The user is able to obtain a clock cycle accurate profile of the real time performance of a software program running on a soft-core processor instantiated on an FPGA. SnoopP is an essential tool for hardware/software codesign on a reconfigurable platform. It allows the user to quickly obtain accurate profiling information that may greatly influence the partitioning of the design.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*optimization*;
C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement Techniques*

General Terms

Design, Measurement, Performance

Keywords

FPGA, embedded processor, hardware/software codesign, performance measurement, profiling, soft processor

FPGA'04 February 22-24, 2004, Monterey, California, USA

1. INTRODUCTION

In recent years, FPGA companies [1, 2] introduced products that enable a user to implement a complete embedded computing system on a single chip. Embedded system designs are often complicated by strict performance, area, and power constraints. This has led to the study of hardware/software codesign issues such as cospecification, cosynthesis, and cosimulation. These issues traditionally arise from partitioning portions of a design into hardware and software components.

The newest FPGA technology allows a designer to use a single reconfigurable platform to instantiate both the processor and the required dedicated logic. To implement an embedded system with ASIC technology, the designer needs to perform initial studies, simulations, and prototyping to ensure that a working design is sent for fabrication the first time. Obviously the need to be completely right the first time is not as much of a concern in the reconfigurable domain as the user can easily redesign the embedded system avoiding further non-recurring costs.

This raises the possibility of a different codesign approach that can be more incremental, such that system development can occur on the final implementation technology. This concept is familiar to software developers who typically design an application on a workstation that is representative of the platform for the final product, not a simulator that models the implementation technology.

Designing on the target technology is equally attractive for FPGA designs. Their potential size and complexity makes simulation a very time consuming process that takes orders of magnitude longer than on-chip execution. Hemmert et al. [3] introduced a debugger for hardware designs capable of running on an FPGA for the benefit of accelerated speed of execution during the debugging process. This upholds the approach of an on-chip design methodology for FPGAs. What is required is a means of understanding the tradeoffs in embedded system designs. Thus, it is interesting to develop tools to facilitate the efficient design of hardware/software codesign applications implemented on FPGA technology.

This paper presents SnoopP, a real-time, non-intrusive *Snoop*ing Profiler (SnoopP) for software applications running on a soft-core processor instantiated on an FPGA. SnoopP allows a designer to precisely measure the number of clock cycles spent executing specified code regions. The objective is to quickly provide the designer with accurate profiling information that helps to identify the portions of

a design that are too slow to be implemented in software when timing constraints are critical. This enables the user to take less time making partitioning choices thus speeding up the design process. Hence, SnoopP is the basis of a suite of tools for designing embedded systems on FPGAs that allows the user to evaluate the design implementation and how to adapt it to meet the given constraints.

The remaining sections of this paper are structured as follows. Section 2 summarizes the previous work done on hardware/software codesign using reconfigurable platforms and cosimulation. It also describes *gprof*, a GNU software profiling tool, and the current tools available for designing embedded systems on FPGAs. Section 3 introduces SnoopP and Section 4 focuses on an implementation of SnoopP, outlining the interface to a MicroBlaze system, the user parameters, and the actual hardware circuitry. A methodology for using SnoopP to obtain profiling results is given in Section 5. Finally, the conclusions and future work for this project are summarized in Section 6.

2. BACKGROUND

Hardware/software codesign implementations arise from applications where there are fixed constraints that cannot be met in software but do not warrant a fully hardware solution. The basic design flow starts with a description of the application that is *partitioned* into hardware and software components. The processes running on each component are *scheduled* to provide the necessary communication between modules. The behaviour of the two environments and their interface can be approximated using *cosimulation* techniques. If this model of the system does not meet the necessary specifications, the designer may need to return to the first phase of the process and re-partition the design. However, if the design constraints appear to be met, the design can be *cosynthesized* and implemented on the target platform. Hardware/software codesign research aspires to address the challenges resulting from each of these complex problems.

This section describes the previous research that uses reconfigurable technology in hardware/software codesigns. It continues with a discussion of how these designs are modelled to provide the designer with the necessary feedback for making appropriate design decisions. Subsequently, there is a description of the GNU tool *gprof* [4], an intrusive software profiler used to obtain statistics on application performance, followed by an outline of some of the commercial tools available for implementing hardware/software codesigns on FPGAs. The section concludes with a summary of the problems arising from these techniques.

2.1 Using FPGAs in Hardware/Software Codesign

As FPGAs grow in size, they are able to implement an increasing number of large applications. Most of the previous work on hardware/software codesign uses FPGAs to speed up the portions of an application that fail to meet required specifications. These systems use one or more FPGAs that are configured once per application [5], or dynamically reconfigured on Dynamically Programmable Gate Arrays (DPGAs) at runtime [6], to implement different functions. Programmable Active Memories (PAMs) are also used as dynamically reconfigurable coprocessors [7]. These systems benefit from the lower redesign costs of reconfigurable technology, but they do not utilize the technology to obtain feedback as to the actual performance of their design.

The recent advent of soft processor cores for FPGAs enables the customization of a processor core as an application specific embedded processor [8]. Hebert et al. use soft-cores to simplify the creation of application-tailored processors by reducing the design time and removing the need for processor specific emulators. In this case, the designers use reconfigurable technology for prototyping [8].

Tools developed for embedded systems with reconfigurable hardware include a partitioner for dynamically reconfigurable systems that minimizes the energy-delay cost due to computation and configuration by Rakhmatov et al. [9]. Noguera and Badia [10] also present a dynamic scheduling methodology for runtime reconfigurable architectures in hardware/software codesign. To obtain a schedule that minimizes runtime reconfiguration overhead, the scheduler relies on a partitioner to create a good mapping of the algorithm to hardware and software. The partitioner's choices are based on delay and area estimates and data from software profiling tools.

A common problem in hardware/software codesign is that the quality of the design is dependent on the partitioner's allocation of resources. However, the partitioner must make its choices based on estimates or models. Since partitioning occurs at the very beginning of the design process, there is no precise feedback available to the partitioner. The next section contains a brief discussion of some codesign simulation tools and how they partition a design.

2.2 Simulating and Partitioning Hardware/Software Codesigns

Conventional cosimulation environments emulate systems that combine a microcontroller with dedicated hardware to implement an embedded system [11, 12]. However, their simulation techniques result in only an approximation of the actual system performance. Some of the more recent hardware/software codesign research uses reconfigurable processors as the platform for implementation [13, 14]. These architectures combine a reconfigurable functional unit with the microprocessor, but do not use the reconfigurability to provide runtime performance information to benefit the partitioning process. Although the precise details of how applications are profiled for all of these projects are not provided, they attempt to simulate system performance to obtain this data. Mentor Graphics offers Seamless [15], a hardware/software co-verification simulation tool that enables a designer to interface Instruction Set Simulators (ISS) with memory and dedicated logic to detect scheduling problems. However, the cost of simulating both hardware and software causes simulations to run at 1000 to 5000 instructions/second.

Profiling and simulation are integral methods for acquiring performance information about an application, varying in both speed and accuracy. The most accurate method is simulating a system's performance on a cycle-accurate simulator. However, this accuracy incurs significant overhead, and consequently, may be too slow for large systems. Instruction-level simulators provide a coarser level of granularity of simulation, sacrificing accuracy for faster execution.

Designers can also profile an application during execution to measure aspects of performance. Most modern microprocessor's include a limited number of hardware performance

counters that can be used to count "Events". A Performance Application Programming Interface (PAPI) [16] provides users with a high-level interface for the usage of these counters. By annotating the application with calls to PAPI functions, the user can count numerous different kinds of Events [17]. The accuracy of PAPI's results is dependent on a large enough code space such that the overhead of the PAPI sampling code doesn't dominate the counter values [18]. Intel provides a commercial performance analyzer, *VTune*, that allows users to utilize the hardware counters on their processors to profile performance [19]. Another popular profiling tool for measuring performance from GNU [4] is described in the following section. Unlike PAPI, it is a statistical profiler that does not use the hardware counters. However, similar to PAPI, the results are of limited precision.

2.3 GNU's gprof

When creating an embedded system, one approach is to implement the entire design in software and then profile it. Using this information, a designer moves components to hardware to meet the necessary performance constraints. Another approach is to assume that the entire design is in hardware, and then move as many components into software as possible such that the specified constraints are still met. Assuming the former approach, tools such as *gprof* can measure the performance of a software implementation as well as determine the characteristics of an application's execution [20]. This profiling may be done on a different system from the actual target system, which leads to inaccuracies in the results. Changing the platform affects the Instruction Set Architecture (ISA), the microarchitecture, the compiler and potential optimizations, resulting in variances in the executable that is profiled.

To use *gprof*, the designer must compile and link the application with the profiling options enabled. Unlike PAPI, where the user manually inserts the profiling routines into the application, the compiler automatically generates the extra code necessary for generating the profile information used by *gprof*. It inserts this code into the application to count the function calls and to generate an interrupt that samples the Program Counter (PC). While this method allows a precise tabulation of the number of times each function is called, the timing information it obtains from the execution is not as accurate.

At specific intervals, normally every 10 ms, *gprof* samples the PC [20]. Depending on the value of the PC, it increments the execution time of the appropriate function by the sample time. This means that unless the total runtime of the application is significantly larger than the sampling period, the measured execution time for each function may be misrepresentative of the actual execution time. Therefore, for smaller executables, applications are run numerous times so that the profiling information accumulates for a substantial runtime.

Obviously, there is a measurable cost to using a runtime profiler on software executing on a processor. The values are imprecise and there is overhead to running the profiling software. However, the runtime profiling overhead is negligible compared with the time required to provide cycle accurate information by simulation. In other words, while *gprof* may add additional seconds or minutes to a software application's execution, cycle-accurate simulation requires seconds to minutes to simulate each cycle of a hardware system, depending on its complexity. Both Altera and Xilinx provide versions of *gprof* that are able to run locally on their soft processors. They also supply other design tools, some of which are described in the subsequent section.

2.4 Designing Embedded Systems on FPGAs

Xilinx and Altera both support the design of systems combining a processor with dedicated hardware. Altera provides designers with the System On a Programmable Chip (SOPC) Builder [21], which hooks into the Quartus II tools [22]. The user specifies a complete system from IP and user designed components and then the SOPC Builder generates the system.

Having created a design, the user can both debug and simulate its performance. ModelSim [23] can simulate a Nios system design, including the peripherals. This is done by simulating the entire system, including the processor, at the RTL or gate-level. While it provides cycle-accurate information, it is extremely slow making the simulation of larger applications prohibitive. To facilitate on-board debugging of the software, Altera provides multiple options. A simple monitor program called GERMS allows basic debugging operations, and for more complex options, there is GNU's *gdb*, but it can only run on a processor instantiated on an FPGA. Finally, Altera has partnered with First Silicon Solutions [24] to provide a core that connects to the Nios processor and acts as a system analyzer.

Xilinx provides users with a similar tool set, the Embedded Development Kit (EDK). It is available as a separate environment for designing embedded systems on FPGAs. Similar to the Altera SOPC Builder, it generates the necessary hardware and software interfaces to facilitate the design of an embedded system.

To simplify the debugging of designs run on a MicroBlaze processor, Xilinx provides an Instruction Set Simulator that may be run in a cycle-accurate mode on a host computer. Unfortunately, this cycle-accurate simulation does not support peripherals at present, which would allow for faster simulation of the processor when embedded with the rest of the hardware. As with Nios, the complete design, including the processor, must be simulated at the HDL/gate level to obtain a complete simulation.

The users can also insert a Xilinx command stub (*xmdstub*) into their design, which attaches a monitor program to the design so that the user is able to debug the executable on the board. They access their executable via the XMD command window or the *gdb* interface on the host. As the XMD window is a TCL shell, users can add their own commands to interface with a design implemented on an FPGA. Finally, Xilinx supports an IP core, the Microprocessor Debug Module (MDM) that enables the user to perform JTAG based debugging on a configurable number of MicroBlaze processors.

Both companies provide numerous tools for debugging application software. However, neither supply tools capable of providing cycle accurate performance information for an application running in real time on a soft processor core instantiated on an FPGA. This is a factor in embedded system design. The importance of obtaining precise performance measurements for quality design implementation necessitates performance-based tools for designs that are prohibitively large for proper simulation.

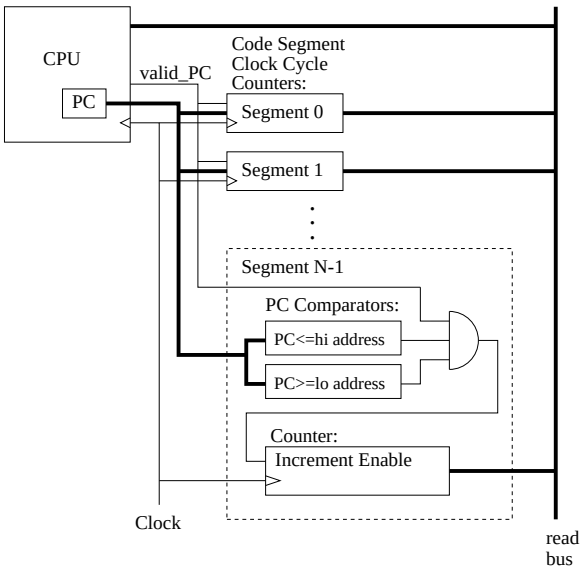

Figure 1: The Generic SnoopP Architecture.

2.5 Summary

To create a good partition of hardware and software components, the partitioner requires accurate feedback as to the performance of the design. Currently, the partition decisions rely on simulated performance and profile data. As described in Section 2.3, depending on how the profiler works, its data may not be very accurate. Furthermore, simulators must estimate and model performance, which means that they can only approximate final system performance.

3. ON-CHIP PROFILING: A NEW APPROACH

SnoopP is an approach to profiling that is targeted specifically at reconfigurable technology. The goal is to be able to precisely profile an application running in software. The premise is that simulating the cycle-accurate performance of a reconfigurable circuit is extremely computationally intensive and should only be used to determine preliminary functionality and not performance. Since the design platform is reconfigurable, measuring performance is possible as long as the user is able to obtain direct feedback on the design's actual behavior. By using the reconfigurable environment to test the system, instead of a simulator, design time can be reduced. The subsequent sections describe the generic architecture of SnoopP and how it benefits the design process.

3.1 General Architecture

Figure 1 illustrates the general structure of SnoopP. As seen in the diagram, the hardware module monitors the PC to see if it accesses specified code segments. Another signal, such as *valid_PC* may be required to indicate when the value in the PC is valid. The user can vary the number of code segment counters to suit the particular application being profiled.

Segment N-1 is magnified to illustrate the internal workings of a code segment's cycle counter. To determine if the PC is in range, comparators check to see if the present PC value is between the specified low and high addresses. If the PC is valid and is presently accessing an address within these bounds, then the counter value is incremented. The user reads and resets the counters in software through their connection to an external read bus. Thus, SnoopP allows the designer to measure the exact number of clock cycles the program spends executing each specified code segment at runtime.

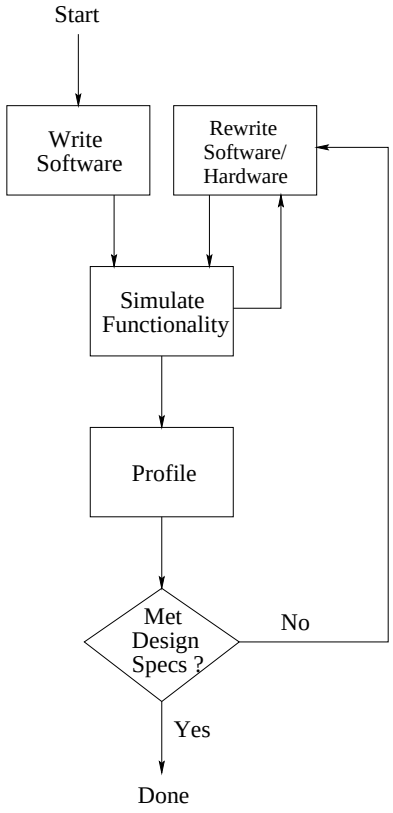

Figure 2: The Proposed Reconfigurable Hardware/Software Design Flow.

3.2 Benefits to the Design Process

Presently SnoopP is only able to profile software, however, the long term objective is to create a tool that allows the user to measure the real time performance of both the software and the hardware on a reconfigurable platform. Ideally, SnoopP allows the designer to assess which aspects of the design, in software or hardware, fail to meet the required specifications. This leads to the introduction of the design flow shown in Figure 2.

The process begins by describing the entire application in software. Once the software is functionally verified, SnoopP profiles it on-chip. If a software implementation fails to meet the design specifications, the profiling information is used by a partitioner or the designer to select which components should be moved to hardware. The design is then re-written to comprise hardware and software components. Again, the user performs functional verification to ensure that the design is correct.

When SnoopP reprofiles the system, the designer again determines if the required design specifications have been met. If not, the designer iterates between retuning the hardware/software codesign and profiling its performance until all specifications are met. In this fashion, SnoopP allows the user to quickly obtain feedback about design performance.

Although, SnoopP currently only profiles program execution, it can easily be adapted to profile data accesses in specified memory regions. Yet, unlike most software profilers that must insert extra code into an application to obtain profiling data, SnoopP is non-intrusive in that it does not alter the user's software executable. It is an independent hardware module, a *Snoop*ing Profiler (SnoopP), that observes the executed PC to determine if any of the counters should be incremented. While the additional hardware circuitry that monitors the processor may be viewed as intrusive to the hardware system, it does not affect software execution.

This technique not only ensures that software performance is unchanged by the addition of the profiler, it also enables a much more precise measure of execution time. Where profilers such as *gprof* sample the program counter to approximate the portion of time spent executing each function, SnoopP counts actual clock cycles and does it very quickly.

The most important benefits to designing hardware/software codesigns on an FPGA are that there is no need to finalize the partitioning of the design at the beginning of the design process or to create a complex cosimulation environment to model communication between hardware and software. The system can be run on the reconfigurable fabric where the precise interaction between hardware and software can be tested and it is easy to iterate between partitioning and profiling the design. Furthermore, by profiling the system on-chip it is possible to obtain exact measures of performance and thus provide better feedback for the partitioning process.

4. TOOL DESCRIPTION

This section describes an implementation of SnoopP for the Xilinx MicroBlaze processor.

4.1 Experimental Platform

For this study, the Xilinx Multimedia Board with a Virtex II 2000 was used to implement MicroBlaze designs. SnoopP is designed to complement the existing Xilinx tools, using the *xmdstub* and *xmd* monitor program as the user interface. Although this implementation of SnoopP is intended to run with MicroBlaze processors, all it requires to profile the performance of a software application is access to the executing PC of the processor and an indication of when the PC is valid. To port SnoopP to a different design platform, it may also be necessary to adapt the present I/O interface to new system bus protocols to make the counter values accessible.

The Xilinx MicroBlaze processor is a soft-core that can be configured to best support the application that is being run on the system. However, since the objective of this study was to develop a real-time, non-intrusive, on-chip profiler for software running on a soft core processor, the default values for the MicroBlaze core were adequate. This includes using the software implementation of the multiply instruction, even though the hardware version is supported on the Virtex II architecture.

4.2 The SnoopP Circuit

Figure 3 illustrates how SnoopP interfaces with a MicroBlaze system. It connects to the On-Chip Peripheral Bus (OPB) as a slave device. The counters are memory mapped, which enables the contents to be read and reset from the *xmd* monitor interface. SnoopP also connects to the Mi-

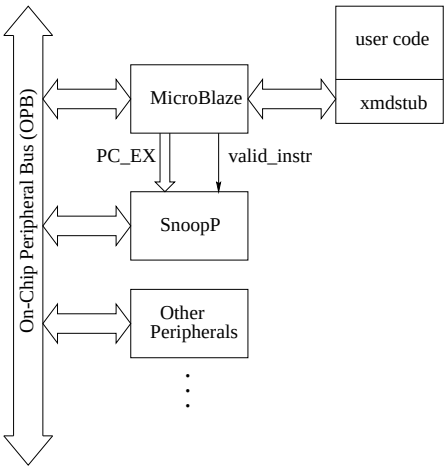

Figure 3: The Interface of SnoopP with a MicroBlaze System.

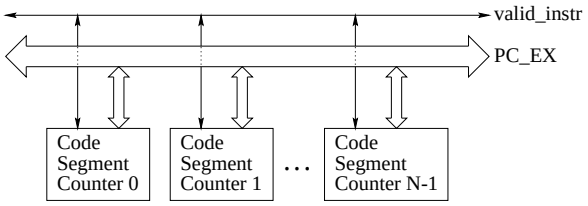

Figure 4: The Architecture of the Code Segment Counting Component.

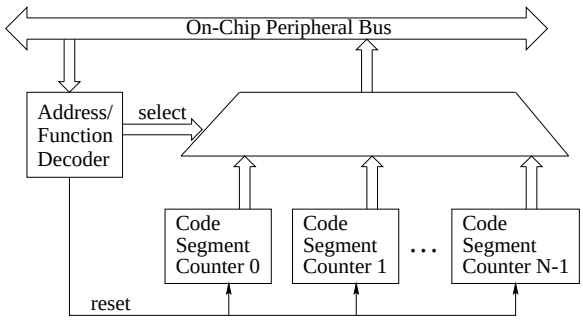

Figure 5: The Architecture of the OPB Interface Component.

croBlaze processor to determine the addresses of executed instructions during runtime.

The internal structure of SnoopP subdivides into two components – the instruction counters and the OPB interface. The former profiles the system with the user-specified number of counters while the latter provides software access to their values. To profile the system, SnoopP connects to the MicroBlaze processor, as seen in Figure 4. The MicroBlaze processor has a *PC_EX* bus that displays the executing program counter and a *valid_instr* signal that is high when the value on the *PC_EX* bus is valid. Thus a counter increments every time the value of the *PC_EX* bus is both valid and in range. This results in an accurate clock-cycle count of the time spent in a code segment. For a cycle-accurate profile of data region accesses, SnoopP simply needs to read the address bus and look for a "valid data address" signal. The OPB interface, illustrated in Figure 5, is responsible for resetting the counters and writing their values to the OPB.

4.3 User Parameters

SnoopP allows the user to choose from a limited number of counters to define independent instruction ranges. The user can obtain the addresses for the upper and lower bound parameters by assembling the code or reading the symbol table. The HDL wrapper for SnoopP includes these values, along with the number of counters, as parameter definitions. Furthermore, parameters in the HDL wrapper enable the user to memory map the counters to any address space that is available in their design. Since all of these parameters are hardwired during synthesis, the number of counters, their individual address ranges and their location in the memory map can only be set before circuit synthesis.

When using SnoopP, it is important to remember that only instructions executed in contiguous regions of memory are counted. For example, to accurately profile how long a function *A* with subfunction calls *X*, *Y*, and *Z* takes to execute, the user must assign a counter to the function as well as to each of the subfunctions called during it's execution (i.e. *A*, *X*, *Y*, and *Z*). Furthermore, if another function *B* calls any of these subfunctions, for instance *Y*, it may not be possible to distinguish which portion of the subfunction *Y*'s execution time is due to function *A* versus function *B*.

4.4 SnoopP interface

The standard EDK design flow can incorporate the SnoopP logic block into the design as a separate module. Once the design is downloaded onto the board, the *xmd* control window provides a software interface to the system design. The commands available in *xmd* allow the user to control and examine the SnoopP counters. This includes writing TCL scripts that can initialize the circuit and analyze the results.

4.5 Design Decisions

The objective is to make the SnoopP circuit as small and as fast as possible so that it doesn't impact the embedded system design. However, to be a useful software profiler, it must allow the user flexibility in specifying address ranges and the number of counters for the system. The decisions outlined below are an attempt to balance these issues.

The maximum number of profiling counters is set to 16 to limit circuit size. Each counter requires two comparators to determine if the 32-bit address is in a valid range. To provide complete flexibility in specifying the code segments, there is no minimum or maximum code segment size. This means that a code segment could be anywhere from a single instruction to an entire program. However, this flexibility has potential performance costs as the comparators must be large enough to differentiate between individual instruction addresses. Thus, the number of counters and code segment boundaries are hard-wired for each design. While it is desirable to be able to reload the boundaries between application runs, it is potentially more important that the circuit not limit the system's clock speed. However, if speed is not a concern, then the design can easily be changed to using programmable address ranges.

Since most software programs require many cycles for completion, 64-bit counters are used to store the clock cycle counts. The decision to count clock cycles, as opposed to the number of executed instructions, is based on the desire to be precise as to the actual time spent executing each code segment. Given that most code segments will include a branch and/or a memory fetch, there will likely be pipeline stalls that could significantly increase the time spent executing a segment. This stall time is not accounted for if only the number of executed instructions are counted.

While this architecture provides the user with significant flexibility for profiling software, the hardware required to implement SnoopP with the maximum 16 counters is undesirably large. The architectural decisions translate into a maximum circuit size that utilizes 1129 flipflops and 1719 LUTs for logic. The 16 64-bit counters require 1024 flipflops, accounting for 91% of the flipflops utilized by SnoopP. The remaining flipflops latch internal control signals to prevent the system's critical path from being in SnoopP for this system.

In the present MicroBlaze system, SnoopP does not limit the maximum clock speed and, ideally, the profiling circuit will never be on the system's critical path. However, if the design is approaching the capacity of the FPGA, it may be unavoidable. If necessary, SnoopP can be pipelined to reduce the delay path in faster systems. This includes latching the *PC_EX* and *OPB_ABus* buses and the *valid_addr* and *OPB_select* signals. These additions have not been incorporated into the present version of SnoopP as they are unnecessary and increase the size of the circuit.

To implement the 32 32-bit comparators used to determine if an executing PC is within a code segment requires 1024 LUTs. This encompasses only 60% of the LUTs employed in the SnoopP, but does not include the logic required to interface SnoopP to the OPB. The OPB interface must use two comparators to resolve that the user has accessed the SnoopP memory space. More logic is required to select the counter operation and to implement a 16-to-1 multiplexer that drives the appropriate value onto the OPB. Thus, the resources necessary to implement SnoopP using 16 counters is actually larger than what is required to implement a MicroBlaze processor, and an area for future study is possible methods of minimizing the size of SnoopP. These include reducing the maximum number of counters, the counter size, or the resolution of the address ranges.

5. PROFILING BENCHMARKS

This section illustrates how SnoopP is used to profile two different benchmarks. It details the issues encountered when profiling each application and concludes with a discussion of the effectiveness of the SnoopP approach.

Function Name	Total Calls
Func_1	300
Proc_7	300
Func_2	100
Func_3	100
Proc_1	100
Proc_2	100
Proc_3	100
Proc_4	100
Proc_5	100
Proc_6	100
Proc_8	100
main	1

Table 1: gprof Statistics on Functions comprising the Dhrystone Benchmark after One Hundred Passes.

5.1 Methodology

The first benchmark, Dhrystone [25], is a relatively small application whereas the second, a cipher block chaining implementation of the Rijndael algorithm (AES) [26], is significantly larger. There are two possible methods of using SnoopP to profile software performance. The first is to use *gprof* to obtain an initial profile of executable performance. This information can be used to try and assign the counters to what *gprof* determines are the important regions of the executable.

The other method is to perform all the profiling using SnoopP. To do this, the user divides the software executable into groups of functions forming continuous address blocks and obtains an initial profile. The regions that require the largest percentage of execution time can be subdivided further to determine which specific functions take the most execution time. Depending on the size of these regions, the number of functions and the division of execution time, the user may have to iterate through this process until a suitable performance profile has been obtained.

Given that the design must be resynthesized every time the counter range values change, it is preferable to limit the number of times SnoopP must be reconfigured to profile the system. Therefore, for this study, the application is initially profiled with *gprof* on a Sun Blade 1000 running version 8 of the Solaris OS. The design is then run on a MicroBlaze processor instantiated on the FPGA and profiled with SnoopP for more precise performance information. Both applications are compiled using *gcc -O3* for both the Sun and MicroBlaze platforms. As the authors are unfamiliar with the executional behaviour of both benchmarks, it prevents the intentional assignment of the counters to known problem areas and ensures that the method is totally dependent on profiling information.

5.2 Dhrystone

Dhrystone is a synthetic benchmark to test a system's integer performance that Xilinx uses to measure MicroBlaze processor performance. The downloaded MicroBlaze version of Dhrystone makes only one hundred passes through the main loop. This means that *gprof* is unable to obtain any statistical timing information as it completes execution in less than 10ms on the workstation.

As can be seen from Table 1, *Func_1* and *Proc_7* are called three times more than the other procedures in the benchmark. However, this does not provide any indication as to which functions are the most costly to implement in software. By increasing the number of passes to a million, *gprof* is able to obtain the data in Table 2

Function Name	Percent Time	Self Seconds	Cumulative Seconds	Total Calls
internal_mcount	39.0	0.76	0.76	—
Proc_8	12.8	0.25	1.01	1000000
main	8.7	0.17	1.18	1
Func_1	5.6	0.11	1.29	3000000
_mcount	5.6	0.11	1.40	—
Proc_7	5.1	0.10	1.50	3000000
Proc_1	5.1	0.10	1.60	1000000
Proc_2	4.1	0.08	1.68	1000000
Proc_4	3.6	0.07	1.75	1000000
Func_2	3.1	0.06	1.81	1000000
Proc_3	2.1	0.04	1.85	1000000
Proc_6	2.1	0.04	1.89	1000000
Func_3	1.5	0.03	1.92	1000000
Proc_5	1.5	0.03	1.95	1000000

Table 2: gprof Statistics on Functions comprising the Dhrystone Benchmark after a Million Passes.

Counter Number	Function Name	Number of Instructions
0	main	368
1	Proc_1	65
2	Proc_2	13
3	Proc_3	17
4	Proc_4	17
5	Proc_5	7
6	Proc_6	41
7	Proc_7	5
8	Proc_8	65
9	Func_1	10
A	Func_2	47
B	Func_3	7
C	divsi3_proc	38
D	malloc	10
E	mulsi3_proc	22
F	strcmp	32

Table 3: Dhrystone SnoopP Counter Assignments.

The functions *internal_mcount* and *_mcount* are part of the profiler and count the number of times a function is called during execution. While *gprof* does not report the number of times these functions are called, their combined overhead accounts for 44.6% of the execution time calculated by *gprof*. Although this data dominates the results, the increased execution time provides a clearer picture of where most of the execution time is probably spent. It is also interesting to note that while *Func_1* and *Proc_7* have three times the number of function calls, the executable appears to spend more than twice their respective execution times in *Proc_8*.

Since this application only has a few functions, it is possible to assign the counters in SnoopP to almost every function. Table 3 outlines how the application is partitioned into profiling segments. It includes the number of instructions per code segment to indicate the static size of the code and to give better context to the profiling results. The table also illustrates that *main* accounts for 48% of the static code size whereas *Proc_8* is only 8.5%. When selecting which regions should be profiled, all the initialization and clean up portions of the executable were ignored as they add little overhead and cannot be moved to hardware.

Function Name	Percentage of Execution Time (100 Passes)	Percentage of Execution Time (A Million Passes)	Percentage of Execution Time (100 Passes) HW multiply
mulsi3_proc	29.31	29.34	—
divsi3_proc	14.93	14.95	22.41
main	12.81	12.75	18.40
strcmp	10.32	10.33	15.49
Proc_1	8.23	8.24	12.36
Proc_8	7.69	7.69	6.43
Func_2	4.06	4.07	6.10
Proc_6	3.07	3.08	4.45
Proc_3	1.87	1.87	2.80
Proc_4	1.87	1.87	2.80
Proc_7	1.65	1.65	2.47
Proc_2	1.43	1.43	2.14
Func_1	1.32	1.32	1.98
Proc_5	0.77	0.77	1.15
Func_3	0.66	0.66	0.99
malloc	0.02	0.00	0.03

Table 4: Cycle-Accurate Results using SnoopP to Profile Dhrystone on MicroBlaze systems that include and exclude the Hardware Multiplier.

Table 4 contains the results obtained from profiling the Dhrystone benchmark on the FPGA. The percentages are based on assuming that the total execution time can be approximated by summing the time spent executing the functions within the user written portion of the executable. Notice that there is a significant difference between results obtained by SnoopP versus *gprof*. Not only are the execution time percentages different, but *gprof* ranks *Proc_8*, *Func_1*, and *Proc_7* as the top three of the application's functions consuming processing time. In contrast, SnoopP proves that excluding the library functions, *Proc_1*, *Proc_8*, and *Func_2* actually consume the most processing time. Therefore, if the designer bases the partitioning of the design on the results from *gprof*, the designer does not select the appropriate functions to implement in hardware.

Comparing columns two and three illustrate the consistency of profiling information obtained using SnoopP. While *gprof* obtained different profiles by executing the Dhrystone main loop one hundred times versus a million times, SnoopP obtained results that vary by no more than 0.06%. The resulting variance is easily explained by the diminishing significance of initialization code with respect to the longer execution time of the main loop. Therefore, SnoopP is able to obtain more accurate and consistent results in only 0.01% of the execution time.

The two functions requiring the largest percentage of the execution time are *mulsi3_proc* and *divsi3_proc*. These implement software versions of the integer multiply and divide functions respectively. The divide function is only called in *main* whereas the multiply function is called in both *main* and *Proc_8*. Column 4 illustrates how the inclusion of the hardware multiplier can affect the performance profile.

The removal of the software multiply instruction reduces the overall instruction count, which generally increases the percentage execution time of all of the functions except for *Proc_8*. Since *Proc_8* calls *mulsi3_proc*, these results are not intuitive. However, the reason for the decrease in execution time is mainly due to the fact that the number of instructions in the function dropped from 65 to 34 due to optimizations that were possible with the removal of the software multiply.

Function Name	Percent Time	Self Seconds	Cumulative Seconds	Total Calls
MixColumns	48.9	0.23	0.23	180000
rijndaelEncrypt	42.6	0.20	0.43	20000
internal_mcount	4.3	0.02	0.45	—
blockEncrypt	2.1	0.01	0.46	20000
cipherInit	2.1	0.01	0.47	20000
makeKey	0.0	0.00	0.47	2
rijndaelKeySched	0.0	0.00	0.47	2
main	0.0	0.00	0.47	1
rijndaelCBC_MCT	0.0	0.00	0.47	1

Table 5: gprof Statistics on Functions Comprising the AES Benchmark for 2 Different Keys with 10 Thousand Blocks Each.

In summary, the difference between the results obtained on the FPGA with SnoopP and those obtained using *gprof* highlight the inaccuracy of sampling the program counter to determine which functions require the longest execution time. Moreover, the on-chip results only required one hundred passes of the main loop in Dhrystone whereas a million passes were needed to obtain statistical timing information using *gprof*.

5.3 AES

AES is a more realistic benchmark for SnoopP to profile [27]. Like Dhrystone, it also uses only integer mathematical operations. However, it is a popular design for hardware, which can greatly increase the throughput rate of the encryption.

Originally, *gprof* profiled the application with two different keys encrypting ten thousand blocks each. However, since the *internal_mcount* function ranked third in terms of processing time, the number of keys used to encrypt the ten thousand blocks was increased to four hundred. The results from these different runs are found in Tables 5 and 6 respectively. As can be seen from these tables, the percentage of execution time for each function changes as the execution time is increased. Furthermore, the longer run of the executable resulted in *rijndaelCBC_MCT* accruing a larger percentage of the execution time than in the shorter pass. Another side effect of increasing the number of keys to 400 is that the timing interrupt resulted in *_mcount* being sampled.

The AES executable is significantly larger than that of Dhrystone, hence it is impossible to assign individual counters to each of the functions. Instead, one counter is assigned to count all the clock cycles used in the executable and the rest are assigned to functions deemed important in the profiling results from *gprof*. Remembering that the counters will only increment when the program is inside their respective code segments, counters are also assigned to the functions called by these main functions. Table 7 summarizes what functions are chosen for profiling and number of instructions comprising each.

Table 8 summarizes the results obtained using SnoopP to profile AES when only two keys are used and when four hundred keys are used. There is only one column of results as there is no change in any of the values when the number of keys is increased, reinforcing the fact that the on-chip profiling provides valuable information with significantly less

Function Name	Percent Time	Self Seconds	Cumulative Seconds	Total Calls
MixColumns	56.5	43.75	43.75	36000000
rijndaelEncrypt	32.8	25.34	69.09	4000000
blockEncrypt	3.5	2.67	71.76	4000000
internal_mcount	3.2	2.46	74.22	—
cipherInit	2.8	2.16	76.38	4000000
rijndaelCBC_MCT	1.2	0.90	77.28	1
_mcount	0.1	0.09	77.37	—
makeKey	0.0	0.00	77.37	400
rijndaelKeySched	0.0	0.00	77.37	400
main	0.0	0.00	77.37	1

Table 6: gprof Statistics on Functions Comprising the AES Benchmark 400 Different Keys with 10 Thousand Blocks Each.

Counter Number	Function Name	Number of Instructions
0	Entire Program	13840
1	MixColumns	139
2	rijndaelEncrypt	308
3	blockEncrypt	309
4	cipherInit	97
5	makeKey	149
6	rijndaelKeySched	301
7	main	14
8	rijndaelCBC_MCT	445
9	modsi3_proc	38
A	mulsi3_proc	22
B	divsi3_proc	38
C	strncpy	66
D	memcpy	11
E	sprintf	26
F	_vfprintf_r	1652

Table 7: AES SnoopP Counter Assignments.

loop iterations. The majority of the execution time is spent in the *modsi3_proc*. This function is called in *MixColumns*, *rijndaelEncrypt*, *rijndaelKeySched*, and *mul*, which explains its dominance. *modsi3_proc* is an internal function used to implement a software version of the modulus function, similar to *mulsi3_proc*, and *divsi3_proc*.

MixColumns and *rijndaelEncrypt*, ranked first and second by *gprof* in terms of sampled execution time, are expected to require significant execution time. Although their execution does not obviously dominate the results obtained using SnoopP, the reason is that the time spent in function calls is not counted in these percentages. This includes the time spent in *modsi3_proc*, which is called twice from both *MixColumns* and *rijndaelEncrypt*.

The percentage of the total execution time measured by the SnoopP counters equaled 96.72%. Since one counter is used to measure the application's total execution time and the remaining fifteen are assigned to look at specific functions, this is reasonably good coverage. It translates into monitoring only 3555 instructions, which is 26% of the total application to achieve over 95% of the executional time coverage. This coverage is partially due to the compiler inlining function calls so that the executable contains functions, such as *mul*, that are never called during program execution.

Profiling AES demonstrates that the initial profile *gprof* provides is a valuable indicator of potential performance hot spots. The on-chip profiling results suggest that the first function to implement as hardware would be the modulus

Function Name	Percentage of Execution Time
modsi3_proc	65.24
MixColumns	14.13
rijndaelEncrypt	7.78
_vfprintf_r	5.71
blockEncrypt	1.16
cipherInit	0.93
mulsi3_proc	0.62
sprintf	0.46
rijndaelCBC_MCT	0.32
memcpy	0.19
divsi3_proc	0.17
rindaelKeySched	0.00
makeKey	0.00
strncpy	0.00
main	0.00

Table 8: The Results from Profiling AES on-chip with SnoopP for Both 2 and 400 Keys.

function. It has the largest execution time and is called from multiple functions. The other most obvious function to implement in hardware is *MixColumns*, which requires almost twice the execution time of the next most time consuming function and is at the core of the encryption algorithm.

5.4 Results of using SnoopP

Using SnoopP to profile a system produces consistent, fast, clock cycle accurate profiles of execution performance as demonstrated by the results of profiling Dhrystone and AES. While, *gprof* is able to obtain a basic overview of software performance, it needs numerous more loops of the main algorithm to obtain its percentage of execution time per function. Moreover, this data is statistical and does not match the exact results measured by SnoopP. However, the initial profile from *gprof* is very useful in determining which code segments likely require the most execution time. It greatly facilitates the assignment of the SnoopP counters to the appropriate code segments.

6. CONCLUSIONS AND FUTURE WORK

SnoopP is a real-time, non-intrusive, on-chip software profiler for soft core processors on a reconfigurable platform. It provides clock-cycle accurate results at speeds much faster than possible by simulation without altering the executable. It is proposed as part of an approach to a new system design flow where the design performance is not simulated but measured on-chip. An implementation of SnoopP on a Xilinx Virtex II FPGA with the MicroBlaze processor has been used to profile two benchmarks that are unfamiliar to both authors to demonstrate the facility of its use. Furthermore, the modular design of the circuit makes it easily adaptable to other platforms.

The future direction of this project is to adapt SnoopP to profile systems that comprise both software and hardware components. Research towards reducing the area and delay of SnoopP is currently underway. Finally, if the profiler runs fast enough, soft loadable boundaries would reduce the number of times a design might need to be resynthesized to profile the different desired code segments.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council and the Ontario Government. The authors would like to thank Tor Aamodt, Anish Alex, Jason Anderson, Tomasz Czajkowski, Ian Kuon, and the anonymous reviewers for their many helpful comments and suggestions. The authors would also like to acknowledge Goran Bilski, Satish Ganesan, Ram Subramanian, and Ralph Wittig from Xilinx for providing additional information on MicroBlaze and their embedded system design tools.

7. REFERENCES

[1] Altera's Home Page. Online: http://www.altera.com.

[2] Xilinx's Home Page. Online: http://www.xilinx.com.

[3] K. S. Hemmert, J. L. Tripp, B. Hutchings, and P. A. Jackson. Source Level Debugger for the Sea Cucumber Synthesizing Compiler. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[4] GNU's Not Unix! The GNU Project and Free Software Foundation (FSF). Online: http://www.gnu.org.

[5] S. Kimura, Y. Itou, and M. Hirao. A Hardware/Software Codesign Method for a General Purpose Reconfigurable Co-Processor. In *5th International Workshop on Hardware/Software Co-Design (Codes/CASHE '97)*, March 1997.

[6] J. Fleischmann, K. Buchenrieder, and R. Kress. Codesign of Embedded Systems Based on Java and Reconfigurable Hardware Components. In *Design Automation and Test in Europe*, March 1999.

[7] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1), 1996.

[8] O. Hebert, I. C. K., and Y. Savaria. A Method to Derive Application-Specific Embedded Processing Cores. In *Proceedings of the Eighth International Symposium on Hardware/Software Codesign*, May 2000.

[9] D. N. Rakhmatov and S. B. K. Vrudhula. Hardware/Software Bipartitioning for Dynamically Reconfigurable Systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, May 2002.

[10] J. Noguera and R. M. Badia. Dynamic Run-Time HW/SW Scheduling Techniques for Reconfigurable Architectures. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, May 2002.

[11] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, Dordrecht, The Netherlands, 1997.

[12] R. Ernst, J. Henkel, and T. Benner. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Design and Test of Computers*, 10(4), September 1993.

[13] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Design Automation Conference*, June 2000.

[14] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli. HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, May 2002.

[15] Mentor Graphics. Seamless Co-verification Simulator. Online: http://www.mentor.com/seamless.

[16] PAPI's Home Page. Online: http://icl.cs.utk.edu/projects/papi/.

[17] S. Browne, J.Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3), 2000.

[18] W. Korn, P.J. Teller, and G. Castillo. Just how accurate are performance counters? In *20th IEEE International Performance, Computing, and Communications Conference*, April 2001.

[19] B. Sprunt. Pentium 4 Performance-Monitoring Features. *IEEE Micro*, 22(4), 2002.

[20] GNU gprof Manual. Online: http://www.gnu.org/manual/gprof-2.9.1/gprof.html.

[21] Altera's SOPC Builder. Online: http://www.altera.com/products/software/system/products/sopc/sop-index.html.

[22] Altera's SOPC Builder. Online: http://www.altera.com/products/software/pld/products/q2/qts-index.html.

[23] Model Technology Home Page. Online: http://www.model.com.

[24] First Silicon Solutions Home Page. Online: http://www.fs2.com.

[25] R. P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10), 1984.

[26] J. Daemen and V. Rijmen. Rijndael, the Advanced Encryption Standard. *Dr. Dobb's Journal*, 26(3), 2001.

[27] Page: The Block Cipher Rijndael. Source Code Download: Reference code in ANSI C v2.2. Online: http://www.esat.kuleuven.ac.be/ rijmen/rijndael/.