Maximizing System Performance: Using Reconfigurability to Monitor System Communications

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

Abstract

Commercial FPGA companies now provide tools that allow users to implement designs comprising soft-core processors and modules of dedicated logic. If a designer chooses to partition a system into multiple processors and hardware modules, tools and techniques for design analysis are necessary to understand system performance.

This paper introduces WOoDSTOCK, a tool that profiles system performance by adding monitors to the circuit running in real time on the chip. The user is able to generate a system specific profiler tailored to monitor the communication links between the different computing elements. This provides a macroscopic picture of system performance, which highlights the computing elements that cause bottlenecks in the design.

1. Introduction

Commercial Field Programmable Gate Array (FPGA) companies provide design tools that support the design of large systems that include multiple processors. This allows a complete embedded computing system to be implemented on a single chip. To create circuits of this size and complexity while minimizing design time requires the reuse of previously designed modules, known as Intellectual Property (IP) cores. The concept of module reuse behind the popularity of IP cores is analogous to the reuse of software library functions in different applications. However, the actual use of hardware IP has never been as easy as in software due to the increased difficulty of abstracting low-level information from the hardware designer.

To simplify the integration of IP cores into different designs, the VSI Alliance has created a set of Virtual Component Interface Standards that allow users to treat IP hardware modules the same as components on a printed circuit board [1]. However, even with these standards, there are still many low-level timing and interface issues that must be considered to connect IP blocks together. If the communication between hardware modules is viewed from a higher level of abstraction, it is possible to uncouple these low-level issues from the data transfers and create a single intra-chip communication format between IP blocks in SoC designs.

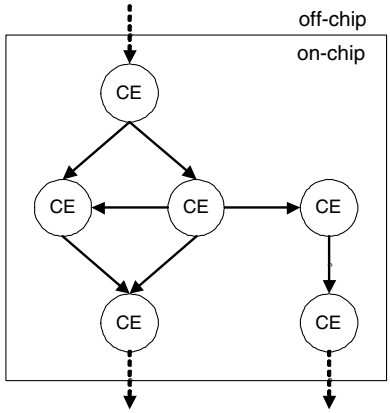

Figure 1. A generic processing system described using the SIMPPL model.

Work is currently being done to investigate the effectiveness of modelling SoCs as Systems Interfacing Modules with Predefined Physical Links (SIMPPL). The SIMPPL model represents an SoC as a combination of different Computing Elements (CEs) that are connected via communication links. A previously proposed model for the future of SoC design using many interacting heterogeneous processors [8] can have this structure, however, the SIMPPL model is more general, allowing CEs to depict either processors or dedicated logic modules.

Figure 1 illustrates a possible embedded system processing architecture described using the SIMPPL model, where the solid lines indicate *internal links* and the dotted lines indicate *I/O communication links*. I/O communication links may require different protocols to interface with off-chip hardware peripherals, but the internal links are standardized physical links to make the actual interconnection of CEs a trivial problem and to create a framework for embedded systems design. The information passed between CEs is abstracted from the links themselves and instead, the data transfers are adapted to the specific requirements of each CE. This format of communicating data between modules is akin to software design, where the stack provides the physical interface between software functions, similar to the proposed internal links. However, the information passed on the stack, such as the number of parameters, is determined by the individual function calls. In the SIMPPL model, the size

MicroBlaze and XPS are registered trademarks of Xilinx Incorporated.

Nios is a registered trademark of the Altera Corporation.

and nature of the data in the packet communicated between the IP modules performs this task. Each module must have internal protocols capable of properly creating and interpreting the information in a packet.

To study the viability of the design model shown in Figure 1, requires the building, testing, and analysis of numerous benchmark systems. However, before creating any systems, an important first step is to develop tools that provide feedback on the designs. To autogenerate tools that provide system-specific runtime information, a standardized format for linking CEs is desirable. The first proposed internal link format is a FIFO, which has the added advantage of enabling communication of data packets between asynchronous clock domains. Furthermore, the size of the FIFO can be set to support the size and nature of the data transmitted on each internal link. Based on this system model definition, tools can be produced to obtain feedback on the performance of a system at runtime, thus improving the design process.

This paper presents a real-time, on-chip system profiler that Watches Over Data STreaming On Computing element linKs (WOoDSTOCK) in designs similar to Figure 1. WOoDSTOCK is a profiling tool that is generated automatically for a particular target system. Its purpose is to monitor the communication links between CEs and gauge their utilization. This tool will be used in future work to evaluate the performance of designs implemented using the SIMPPL model.

A tool for autogenerating system benchmarks, called the System Generator, creates synthetic benchmarks that are used to demonstrate the functionality of WOoDSTOCK. These benchmarks are designed to mimic real systems that have CEs with varied numbers of inputs and outputs. This early version of the System Generator models all CEs as soft-core processors running software. It is a preliminary step that will eventually allow the user to create systems with a variable number of links and CEs, be they processors or hardware modules. The current version of the system Generator can generate sufficient benchmarks to verify WOoDSTOCK, which will then be used to study embedded computing systems built using the proposed SIMPPL architecture.

Embedded systems typically combine a processor with some hardware logic to meet specified performance constraints. To analyze the performance of the system, including the interactions between hardware and software, designers have used complex modelling techniques to simulate system performance. These results are able to approximate the operation of the design to provide some understanding of the system at run-time.

If the designer uses a reconfigurable design platform to implement the actual system, system development can also be done on the platform to study the real time performance. This leverages the main advantage of reconfigurability — the user may redesign the system while avoiding non-recurring costs. Then hardware design begins to resemble software design, where debugging and profiling instrumentation are easily added to the software to obtain run-time information. Tools, such as WOoDSTOCK, will play an important role in this type of environment.

The idea of using the implementation platform during the design process is presented as part of previous work that investigated the on-chip profiling of a single processor on a reconfigurable platform [10]. In this paper, it is extended by focusing on profiling the

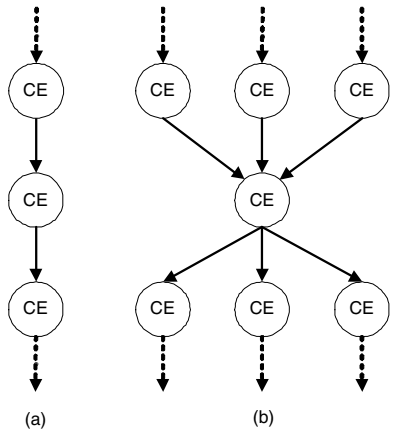

Figure 2. Different application types described using the SIMPPL model: (a) a pipelined data streaming application (b) a router.

communication between the different CEs in a complete reconfigurable system. By analyzing the system performance on-chip, an accurate real-time understanding of the design is obtained to guide the redesign of the system and maximize performance. Currently, Hemmert et al.'s hardware debugger for hardware designs, which can run on-chip to speed up execution during the debugging process [6], is one of the few existing tools to use an on-chip design approach. Ultimately, the goal is to provide a suite of on-chip design tools for designers of reconfigurable systems.

The remainder of this paper is structured as follows. Section 2 further discusses the proposed SoC model and outlines some of the previous work on simulating embedded systems and measuring the performance of multiprocessor systems. Section 3 discusses how the system models are generated and how WOoDSTOCK monitors a system, while Section 4 describes the platform-specific implementation details of WOoDSTOCK and the system models. A set of case studies demonstrating how WOoDSTOCK works is given in Section 5 and the paper closes with conclusions and possible future work for this project in Section 6.

2. Background

Recently researchers have begun investigating simple designs that use multiple Nios [11] or multiple MicroBlaze [7] soft-core processors. As these designs become more complex, the need to assess inter-computing element effects increases. Furthermore, complex multiprocessor designs present the possibility of system models that differ from typical hardware designs.

2.1 SoC Modelling

A common hardware design approach is illustrated in Figure 2(a). This design is a data streaming application that is pipelined to increase the system throughput. Since these applications have a linear data flow, it is easier to balance the computations and communications performed by each CE to prevent the stalling or starving of any CE. Networking applications, such as the simplified router model shown in Figure 2(b) [9], however, provide more interesting system models, due to their decision making components. The division of computations among the CEs

may not be intuitive to ensure the continuous operation of the system. More general structures, such as Figure 1, must then be considered, where it is difficult to statically predict timing information, and thus, properly divide system tasks among the CEs. For all of these application structures, understanding the behaviour of the system at run-time is key to maximizing performance and throughput.

2.2 Measuring Embedded System Performance

Most embedded system designs have restrictions that arise from strict performance, area, and power constraints. If a solution implemented completely in software fails to meet performance requirements, portions of the algorithm must be moved to hardware. The division of the system into hardware and software components is called *partitioning*. The complications arising from designing a system as a combination of hardware and software, also known as hardware/software codesign, make simulators a popular method for measuring embedded system performance.

For example, COSYMA [5] allows a user to cosimulate a design and automatically partitions it into hardware and software components. Partitioning is done at the basic block level using profiling and software timing information. The user provides a worst case data set, which COSYMA profiles to obtain the worst case performance. A newer version of COSYMA replaces the profiling information with SYMbolic hybrid Timing Analysis (SYMTA), which combines simulation and formal analysis, allowing COSYMA to obtain both upper and lower timing bounds to approximate system performance [12]. Other examples of available simulators include Polis [4] and Seamless [2] from Mentor Graphics.

2.3 Multiprocessor System Profiling

The model shown in Figure 1 resembles applications run on multiprocessor systems, where software designers are able to obtain some run-time statistics about an application's behaviour on their system. Of particular interest is their ability to determine the stall time of individual processors in the system. Typically, a scheduler monitors when a processor is waiting for another processing task, but as the scheduler is unaware of the nature of the actual tasks, it only provides system-level information.

This granularity of measurement is analogous to the information that WOoDSTOCK provides FPGA SoC designers. It highlights problems arising from inter-CE communication and indicates to the user when a particular CE acts as a system bottleneck. Like the scheduler, WOoDSTOCK is similarly unaware of the actual computation performed on a CE. Therefore, the precise cause of a system bottleneck is determined using a combination of the system performance results along with user's knowledge of the design. Subsequent work will include developing tools that help the user look in more detail at the problem areas identified at the system-level.

3. General Architecture

This section provides details on the architecture of the autogenerated systems and WOoDSTOCK.

3.1 Autogenerated Systems

The autogenerated systems are of the format illustrated in Figure 1, where each CE has the generic structure shown in Figure 3. Each CE has N input links and M output links. Internal links connect a CE to other CEs, where input links connect to *parent* CEs and output links connect to *child* CEs.

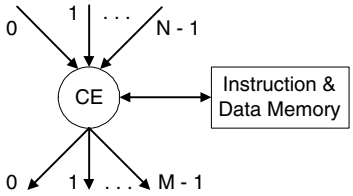

Figure 3. The system generator's generic computing element.

The current System Generator tool represents all CEs as soft-core processors. Figure 3 shows each CE as having its own local instruction and data memory. The main reason for this is that sharing memory creates possible data hazards. Even if two processors share a block of memory but have two distinct address spaces, there will be bus contentions causing interference in the execution results. Here, it is assumed that each of these modules should have the same performance independent of the number of other CEs in the system and that there is no need to share data between two CEs unless it is sent via a link. Each CE's autogenerated source program file is stored on its local memory and provides functions for receiving and transmitting over the links and constants representing the processing time required to generate output data for the CE.

The generality of the current system autogenerator limits its ability to accurately model all types of systems. It is only able to model sequential consumption and generation of data. While modelling pipelining and other forms of parallelism within a CE would be easier using a hardware model, the current focus is to test and study the usefulness of WOoDSTOCK and System Generator creates adequate benchmarks for this purpose.

Since I/O communication links cannot be automatically generated, off-chip peripherals that produce/consume system data are modelled as part of the CE to which they are connected. If there are no internal input links (N=0) to a CE, then it generates output data by modelling input received from an off-chip hardware peripheral that must be processed before generating an output. Similarly, if a CE has no output links (M=0), all data is consumed to model output generated for an off-chip hardware peripheral.

Figure 4 illustrates the connections between WOoDSTOCK and a multi-CE system. Each diamond represents a monitor that is associated with a specific CE. A monitor is a piece of hardware that records the behaviour of the traffic on all the internal input and output links connected to its CE through internal counters. These counters are used to measure the total possible stalling/starving time for a CE during the profiling period, which is set based on the program execution of a specially selected *base* processor.

Each of the CEs and their monitors in Figure 4 are labelled for the purpose of differentiating the base processor (0) from the remaining processors (1,2,3). Although the current benchmarks are all implemented as networks of processors, WOoDSTOCK is able to monitor any system that has at least one processor and uses FIFOs as communication links. The base processor may be any processor in the system as long as its execution can be set for a finite inter-

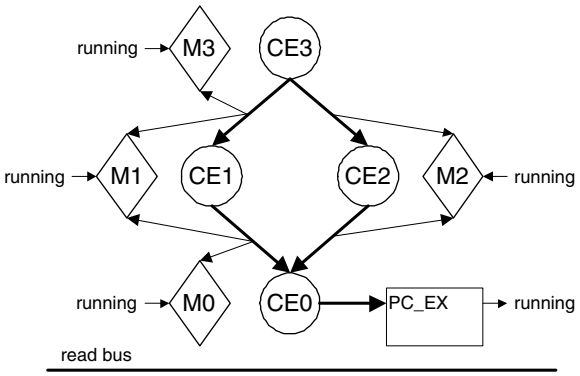

Figure 4. The WOoDSTOCK architecture.

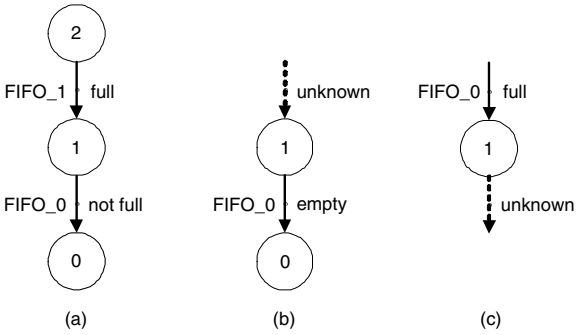

Figure 5. Examples of the different types of bottlenecks detectable by WOoDSTOCK: (a) interior bottleneck, (b) input bottleneck, and (c) output bottleneck.

val. This processor will determine the run-time for the monitors based on its executing Program Counter (PC_EX).

3.2 WOoDSTOCK

The user chooses the region of the base processor's source code where the monitors should be running. Addresses of the instructions bounding this code region are provided to WOoDSTOCK as start and stop points. The *running* signal, shown in Figure 4, is enabled and disabled when the addresses of the defined start and stop instructions, respectively, are seen as valid addresses on the PC_EX bus. This signal is used to enable or disable the system's monitors. The monitors could also be turned on and off by inserting instructions into the software that enable/disable the counters by writing to a memory location, but observing the PC_EX bus is less intrusive to the system's normal run-time behaviour.

WOoDSTOCK assumes that the only signals a monitor can connect to are the full and empty status signals of the FIFOs implementing the internal input and output links of its respective CE. These signals are used to generate enable signals for the counters used to profile the system. The counters are used to measure the number of clock cycles where a CE is potentially starving or stalling the system. A more naive approach would be to assign individual counters to the full and empty signals of each link in the system. However, this provides less useful information to the designer as the relationship between these status signals is required to determine if a CE is a system bottleneck as shown in the following paragraph.

Table 1. Example output equations for the systems in Figure 5.

Bottleneck Example	Output Equation
Interior Bottleneck	FIFO_1_full and (not FIFO_0_full)
Input Bottleneck	FIFO_0_empty
Output Bottleneck	FIFO_0_full

Figure 5 illustrates examples of the three types of system bottlenecks that WOoDSTOCK can be used to detect. Figure 5(a) shows an *interior bottleneck*, where CE 1 has both internal input and output links and is stalling the system. To understand how WOoDSTOCK determines there is a bottleneck, consider when FIFO_1 becomes full. CE 1 may not be consuming the data produced by CE 2 fast enough. However, CE 1 may also be stalled because it cannot write to FIFO_0 if it too is full, in which case a child CE is the bottleneck and not CE 1. To differentiate between these situations, a CE is defined to be an interior bottleneck when all the input links that provide data to generate a specific output are full and the link at the output is not full as depicted in Figure 5(a). The specification of the output link as "not full", as opposed to empty, delineates an important aspect of the system monitoring tool. WOoDSTOCK is unaware of the nature of the data being transferred between CEs, so if CE 1 produces a data packet that CE 0 requires in its entirety to continue processing, then the link should normally be empty when the system is balanced. However, if CE 1 produces a data packet that is consumed as multiple individual data packets by CE 0, then there will normally be data in this FIFO even when the system is balanced. Therefore, the output link must be only "not full" instead of "empty" to produce a bottleneck.

A CE that has internal output links and no internal input links may cause an *input bottleneck*. This occurs when either the off-chip hardware peripheral supplying input to the CE is too slow or the processing time of the CE is too slow. In either case, the system is starved for data. To detect this situation, WOoDSTOCK monitors the empty status signal of the output link. Figure 5(b) shows CE 1 as the potential cause of an input bottleneck. The status of the I/O communication link is unknown and FIFO_0 is empty. However, CE 1 may not be a bottleneck if CE 0 consumes data at the same rate as CE 1 produces it. This situation would also cause FIFO_0 to be empty for the majority of the system's run-time. Since the results from these measurements are not conclusive on their own, the designer needs to see how this information fits in with the results obtained from monitoring the rest of the system.

Output bottlenecks arise in CEs that have internal input links and no internal output links. They occur due to the slow processing rate of either the CE or an off-chip peripheral. Both cases result in the input links to the CE becoming full as illustrated in Figure 5(c). While the state of the I/O communication links is unknown, FIFO_0 becomes full stalling the system. In situations where a CE stalls or starves because of an off-chip peripheral's slow data rate, this is still measured as being caused by the CE implementation. Therefore, the user must be sufficiently familiar with the CE's processing to determine the precise cause of the bottleneck.

To generate a system-specific monitoring system, the user writes a description of the system that states the required combination of data on internal input

links used to produce an output for a given output link. WOoDSTOCK uses this information to create an *output equation* for each CE output described in terms of link empty and full status signals. Table 1 shows the appropriate output equations for CE 1 in each of the systems in Figure 5. These equations generate counter specific enable signals that are combined with the *running* signal to enable all the appropriate counters during each sampling clock cycle.

The frequency of WOoDSTOCK's sampling clock can be set to any rate, depending on the desired profiling accuracy. If sampling is done using the fastest system clock, then the measured results are precise. However, a slower clock may be used to do the sampling and obtain a statistical measurement of system performance. This information can still help detect system bottlenecks, but the system may need to be profiled for longer run-times to observe the problem.

4. Implementation Details

This section describes the implementation of WOoDSTOCK and the multi-processor systems using a Xilinx design platform.

4.1 Experimental Platform

The Xilinx Multimedia Board with a Virtex II 2000 is used to implement MicroBlaze designs. To obtain a performance profile of the design, WOoDSTOCK monitors the communication links between computing elements. The system links are implemented using the Fast Simplex Links (FSLs) created by Xilinx [3] to allow streaming and buffering of data between computing elements. The FSLs are FIFOs that support slave read and master write protocols by MicroBlaze processors.

Benchmarks are autogenerated using the default version of the MicroBlaze soft-core processor to represent each of the system CEs. The code running on each of the processors is stored in local on-chip memories and accessed via Local Memory Buses. The maximum number of MicroBlaze processors is limited by the tools to eight. Each MicroBlaze has eight built-in FSL receive and transmit ports and the autogenerated send and receive functions are based on macros provided by Xilinx to read and write from these ports.

4.2 System Generator and User Interface

The System Generator creates all the necessary source files to describe a unique project for the Xilinx Platform Studios (XPS) software. These files are generated based on an input description file of the system provided by the user. The input file describes how each CE generates its outputs as a function of its inputs. CE 0 represents the base processor for the system. Its source file is generated with a *for* loop encompassing the region of code repeated by the processor, thus creating a finite execution interval. Source files for the remaining processors run continuously, using *while(1)* loops to enclose the data consumption/production loop.

The project file is designed to generate a download file that includes all of the executable source files for the processors. After the bitstream is downloaded onto the FPGA, all the processors startup and begin running their program. A print statement is included in the source code of the base processor, to indicate when WOoDSTOCK is finished monitoring the system. The user then connects to the base processor through the MicroBlaze Debug Module (MDM) to halt its execution and read the counter values from the FPGA via the *xmd* control window.

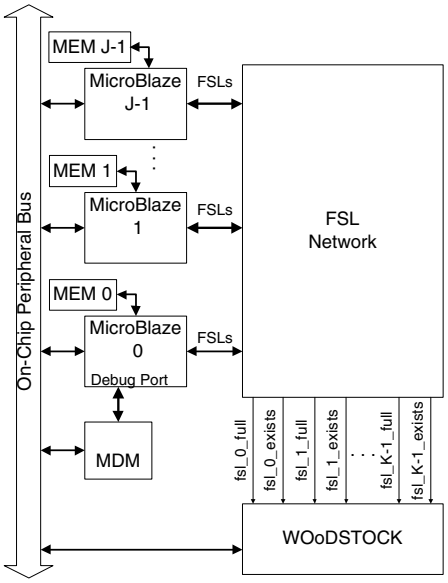

Figure 6. The interface of WOoDSTOCK with a multi-MicroBlaze system.

4.3 WOoDSTOCK Implementation

WOoDSTOCK generates the necessary system dependent VHDL files to implement the monitoring system, along with the files required by XPS to interface WOoDSTOCK into a MicroBlaze system as shown in Figure 6. WOoDSTOCK connects to the On-Chip Peripheral Bus (OPB) as a slave device and to FSL status signals that indicate when there is data in the FSL to read and when the FSL is full. For the purpose of this paper, these signals will be referred to as *fsl_empty* and *fsl_full* respectively. By monitoring their runtime values, WOoDSTOCK enables the appropriate counters based on the user-defined output equations.

The internal structure of WOoDSTOCK is subdivided into two components — the system monitors and the OPB interface. The former profiles the system links based on the user-provided system profile (recall Figure 4) while the latter provides software access to their values. The counters are memory mapped to an OPB interface for reading and resetting their values.

4.4 Design Decisions

The objective is to make the WOoDSTOCK circuit as small and as fast as possible so that it does not impact the embedded system design. However, to be a useful system profiler, it must allow the user flexibility to assign the appropriate number of counters for the system. The decisions outlined below are an attempt to balance these considerations.

The size of the overall circuit depends mostly on the number of counters required to store the system profiling data. The counter size is set to 46-bits to allow a maximum system profile period of eight days at 100 MHz, however, the maximum clock speed for the monitoring system is dependent on the complexity of the system being monitored. Additional logic resources are used to generate the running signal for the counters from two 32-bit comparators for the start and stop addresses, which are hardwired for the system to reduce the required logic.

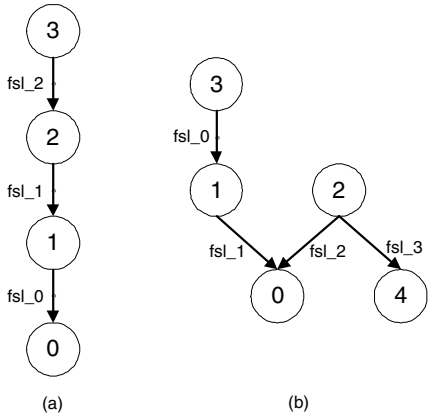

Figure 7. Two application architectures described with the SIMPPL model: (a) a pipelined system (b) a system with branching.

Similarly, the OPB interface needs two comparators to determine if the user has accessed WOoDSTOCK's address space plus logic to multiplex the selected counter onto the OPB. Finally, the logic required to generate the counter specific enable signals depends on the complexity of each output equation. However, the logic resources used are negligible relative to the size of a counter.

5. Case Studies

This section uses WOoDSTOCK in two different case studies to show that the measurement approach works and to demonstrate how the information it provides can help to refine a design. It details the issues encountered while profiling each system and concludes with a discussion of the advantages of on-chip system profiling.

5.1 Methodology

The two benchmarks illustrated in Figure 7 are the case studies used to demonstrate the functionality of WOoDSTOCK. The first is a simple pipelined design that is quite common to hardware design. The second is an imaginary system used to highlight the increasing difficulties of analyzing a design that is less intuitive. For both benchmarks, WOoDSTOCK uses the global system clock as its sampling clock. Different configurations of each system are created by assigning delays to model CE processing times. These processing delays are used to create system imbalances that WOoDSTOCK should report as well as balanced systems to determine how this affects the results obtained by WOoDSTOCK.

Each system configuration is profiled for varying lengths of time to determine the initial effects of system start up on the results. Recall from Section 4.2 that the main processing loop of the base processor uses a *for* loop to determine how many data packets to consume before exiting. The length of the profiling period is varied by changing the number of packets the base processors consume before exiting.

5.2 Pipelined System Example

The pipelined system in Figure 7(a) requires 5 counters to monitor the system. The first three columns of Table 2 list the counters, the output equations used to generate their respective enables, and the possible meaning of these conditions. A question mark in column 3 indicates that this CE may actually not be the source of any system performance problems as it only represents a possible input bottleneck. Counters 0, 1, and 2 monitor FSL full signals to determine if CEs 0, 1 , or 2, respectively, are stalling the system. Counter 3 counts the number of clock cycles for which fsl_2 is empty and Counter 4 stores the total run-time of the monitoring system. Configuration *A* has equal processing delays for all the CEs except for CE 3. Its delay is twice as long as the rest to create a situation where CE 2 should be starved for data. The *for* loop of CE 0, the base processor, is then set to consume 20, 100, and 200 data packets, respectively to create profiles of varying length. While WOoDSTOCK is able to obtain more accurate information about system performance as the profiling time is increased, the consumption of 200 data packets is sufficient to demonstrate WOoDSTOCK's functionality for the different configurations of the pipelined and branching systems.

The results for Configuration A are found in Table 2 in the subcolumns labelled *Con A*. All values in the configuration columns are reported as the percentage of the monitor run-time, which is given to the nearest million clock cycles in the final row of the table (Counter 4). Anytime a counter's value was actually zero, the percentage is reported as *0*, whereas if the value is simply negligible relative to the profiling period, the percentage is reported as *0.0*. This same method is applied to counters that run for almost the entire profiling period, *100.0* versus *100*. As illustrated by the table, the only counter to be incremented monitors when there is no data in fsl_2. Without any knowledge of individual CE behaviour, it is impossible to determine if the system is balanced or if CE 3 is too slow and starving the system. In either case, the processing time of CE 3 needs to be reduced if the designer wishes to try and improve system performance.

The second system configuration reduces CE 3's processing delay by 50% so that all the CEs have equal delays for processing time. This should balance the pipelined system, yet, the results in Table 2, in the subcolumns labelled *Con B*, show that the 3rd counter is still enabled for almost 100% of the monitor run-time even though the system should now be balanced. This is because CE 2 consumes data at the same rate as CE 3 produces, thus fsl_2 remains empty most of the time. Instead, the decreased total run-time for the system in Configuration B proves that CE 3 is an input bottleneck in Configuration A. The overall run-time is reduced by approximately 50%, mirroring the decrease in CE 3's processing delay. Since no other bottlenecks have appeared in the system, the designer may not realize that the system's performance has been maximized if they are insufficiently aware of the individual processing requirements of each CE.

If the user assumes that CE 3 is still an input bottleneck to the system, they may choose to further reduce its processing delay. Configuration C allows CE 3 to run faster by decreasing the processing delay to 90% of the processing delay used by the rest of the system. As can be seen from the data in the subcolumns labelled *Con C*, for smaller run-times, no bottlenecks are introduced into system communications. In fact, the percentage of time for which there is no data in fsl_2 decreases to 23.6% when the base processor consumes only 20 data packets, compared to the other two configurations where there is no data in fsl_2 for almost 100% of the run-time. However, as the system continues to run, fsl_2 becomes full as

Table 2. Table for pipelined system counter results describing the counter enables, what the counters represent, and reporting the measured results as percentages of the total monitor run time given in Counter 4 to the nearest million clock cycles.

Cntr	Enable Condition	Possible Meaning	20 Data Packets			100 Data Packets			200 Data Packets		
			Con A	Con B	Con C	Con A	Con B	Con C	Con A	Con B	Con C
0	fsl_0_full	CE 0 slow	0	0	0	0	0	0	0	0	0
1	fsl_1_full and (not fsl_0_full)	CE 1 slow	0	0	0	0	0	0	0	0	0
2	fsl_2_full and (not fsl_1_full)	CE 2 slow	0	0	0	0	0	0	0	0	30.8
3	fsl_2_empty	CE 3 slow?	100.0	100.0	23.6	100.0	100.0	5.2	100.0	100.0	2.7
4	running	monitors on	688	368	366	3248	1648	1646	6448	3248	3246

Table 3. Table for branching system counter results describing the counter enables, what the counters represent, and reporting the measured results as percentages of the total monitor run time given in Counter 7 to the nearest million clock cycles.

Cntr	Enable Condition	Possible Meaning	20 Data Packets			100 Data Packets			200 Data Packets		
			Con A	Con B	Con C	Con A	Con B	Con C	Con A	Con B	Con C
0	fsl_1_full	CE 0 slow	21.4	10.1	0	83.7	82.2	0	91.8	91.1	0
1	fsl_2_full	CE 0 slow	0	0	0	0	0	0	0	0	0
2	fsl_0_full and (not fsl_1_full)	CE 1 slow	0	0	0	0.0	0.0	0	0.0	0.0	0
3	fsl_2_empty	CE 2 slow?	2.4	94.9	2.3	0.50	99.0	0.5	0.2	100.0	0.2
4	fsl_3_empty	CE 2 slow?	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
5	fsl_0_empty	CE 3 slow?	83.3	94.9	100.0	17.3	18.8	100.0	8.7	9.4	100.0
6	fsl_3_full	CE 4 slow	0	0	0	0	0	0	0	0	0
7	running	monitors on	672	632	352	3232	3192	1632	6432	6392	3232

CE 2 acts as an interior bottleneck because it cannot consume data as quickly as CE 3 produces it. This is reflected in the value of Counter 2 when the base processor consumes 200 data packets and highlights the importance of running systems for long periods of time to achieve a more steady-state view of the system. Furthermore, if CE 3 had still been an input bottleneck to the system, the decrease in the processing delay of CE 3 should have been mirrored in the total run-time of the system, which remained almost unchanged from Configuration B to Configuration C.

5.3 Branching System Example

Figure 7(b)'s branching system requires 8 counters that are enabled based on the functions described in column 2 of Table 3 when the monitors are running. The data in the table is presented following the same format as Table 2. Counters 0 and 1 monitor fsl_1 and fsl_2 to determine if CE 0 is stalling the system. Similarly, counters 2 and 6 measure when CE 1 and CE 4, respectively, stall the system. Counters 3 and 4 count the number of clock cycles for which fsl_2 and fsl_3 are empty as does counter 5 for fsl_0. This information can help to determine if either CE 2 or CE 3 are producing output data too slowly, and thus starving their respective children CEs. The possible interpretations for the counter values are summarized in column 3.

In this system, each data packet to and from each link is processed independently. For example, in CE 2 an output is generated for fsl_2 after a processing delay and an output is generated for fsl_3 after a separate processing delay. Therefore, for CE 2, the time between generating outputs for fsl_2 is the sum of these two delays. Similarly, in CE 0, data is read from fsl_1 followed by a processing delay before data is read from fsl_2 followed by an independent processing delay. In this case, for CE 0, the time between reading inputs from fsl_1 is the sum of these two delays. The first configuration of this system has all of the processing delays for each link set to the same value. This creates an imbalanced system as CE 0 and CE 2 have an effective per link processing delay that is twice that of the other CEs. Again, the base processor's *for* loop is set to consume 20, 100, and 200 data packets, which is sufficient to demonstrate the system imbalances for the following configurations.

Table 3 summarizes the results for Configuration A in the subcolumns labelled *Con A* where all values are in terms of the percentage of the monitor's run-time for which the counter was enabled. The total profiling period is reported to the nearest million clock cycles in Counter 7's row. The importance of running the system for a significant period of time is highlighted by the results for counter 0, which vary from 21.4% to 91.8%. The larger value from the long run-time clearly indicates that CE 0 is stalling the system by not consuming data quickly enough.

To try and remove this bottleneck, CE 0's processing delays for input data read from fsl_1 and fsl_2 are reduced to 50% of the delays for the rest of the system. This means that the combined effective per link processing delays for fsl_1 and fsl_2 are now the same as the rest of the system, with the exception of CE 2's processing delays, which are left unchanged. The results for this configuration are found in Table 3 in the subcolumns labelled *Con B*. From these results, it appears that CE 0 is still stalling the system, however closer inspection disproves this theory. While fsl_1 is still becoming blocked as the run-time increases, the period for which fsl_2 is empty has increased dramatically (see Counter 3). This may indicate that CE 2

cannot keep up with its child nodes. If this is the case, CE 0 is now starved for data on fsl_2 and still not able to keep up with its parent node CE 1. This also is reflected in the overall run-time that remains basically unchanged between Configuration A and Configuration B as the profiling period increases. If CE 0 were the only bottleneck in the system, the system's performance should have increased noticeably. Therefore, CE 2 must also be a system bottleneck, failing to provide data at the necessary production rate.

By reducing CE 2's processing delay for generating outputs for fsl_2 and fsl_3 to 50% of the original processing delay, the system should be balanced. This is designated as Configuration C and the results are found in the subcolumns labelled *Con C* in Table 3. In this case, none of the links become full so the system never stalls. This produces the expected increase in the overall system performance by decreasing the overall run-time by approximately 50% from the Configuration *A*.

5.4 Summary

WOoDSTOCK is able to detect bottlenecks in system performance and the removal of these bottlenecks dramatically improves the overall performance as demonstrated in the above examples. WOoDSTOCK required 579 LUTs and 331 flipflops to monitor the pipelined example and 928 LUTs and 478 flipflops to monitor the branching example. If these results are normalized in terms of the number of counters in each system, the pipelined example uses 115.8 LUTs and 66.2 flipflops per counter and the branching example uses 116 LUTs and 59.8 flipflops per counter. These results highlight that the increased size of WOodSTOCk is mainly due to the extra counters and that overhead logic needed to provide a user interface can be considered minimal.

The system must be run for a significant period of time to obtain accurate results using WOoDSTOCK. This may be on the order of minutes to hours depending on system complexity, and is necessary to account for the initial effects of starting up the system. If these results are to be found via simulation, the required time could be excessive. Although WOoDSTOCK obtains only a macroscopic view of system performance, combined with an understanding of the individual CEs, it provides greater insight into system behaviour that can guide the redesign of a system. Finally, while a designer should be sure that there are no CEs stalling the system, interpreting the meaning of the measured results for more complex systems requires that the Counter values not be viewed in isolation as demonstrated in the branching example.

6. Conclusions and Future Work

WOoDSTOCK is a real-time system profiler that runs on the reconfigurable design platform. It provides a macroscopic picture of system performance that can help guide the redesign of a system. It is important to run the system for a significant period to account for start up noise and to focus the results to the real problem areas. A method of autogenerating system models comprised of Computing Elements and standardized Communications Links is also presented. This generic computing model allows system analysis based on system communication. Both WOoDSTOCK and the autogenerated systems have been implemented on a Xilinx Virtex II FPGA using a variable number of MicroBlaze processors and FSLs.

The next phase of this research is to use WOoDSTOCK to investigate the proposed SIMPPL SoC model to determine the challenges of porting applications to this model and to assess the benefits and limitations of this interconnect structure. Tools capable of providing more localized information along with timing dependent information should also be developed to provide the user with a better understanding of the run-time system behaviour and further the investigation of the SIMPPL model.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council and an Ontario Government Scholarship for Science and Technology. The authors would like to thank Jason Anderson, Tomasz Czajkowski, and Peter Jamieson and the anonymous reviewers for their many helpful comments and suggestions.

7. References

[1] VSIA Home Page. Online: http://www.vsia.org.

[2] Mentor Graphic's Seamless Co-verification Simulator. Online: http://www.mentor.com/seamless.

[3] MicroBlaze Processor Reference Guide. Online: http://direct.xilinx.com/ise/embedded/edk_docs.htm.

[4] 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.

[5] R. Ernst, J. Henkel, and T. Benner. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, Sept. 1993.

[6] 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*, pages 228–237, Apr. 2003.

[7] P. James-Roxby, P. Schumacher, and C. Ross. A Single Program Multiple Data Parallel Processing Platform for FPGAs. In *IEEE Symposium on Field Programmable Custom Computing Machines*, Apr. 2004.

[8] P. Magarshack and P. G. Paulin. System-on-chip Beyond the Nanometer Wall. In *Proceedings of 40th Design Automation Conference*, pages 419–424, June 2003.

[9] N. Shah, W. Plishker, and K. Keutzer. NP-Click: A Programming Model for the Intel IXP1200. In *2nd Workshop on Network Processors, 9th International Symposium on High Performance Computer Architecture*, Feb. 2003.

[10] L. Shannon and P. Chow. Using Reconfigurability to Achieve Real-Time Profiling for Hardware/Software Codesign. In *ACM Int. Symposium on Field-Programmable Gate Arrays*, pages 190–199, Feb. 2004.

[11] X. Wang and S. G. Ziavras. Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines. *Concurrency and Computation: Practice and Experience*, 16(4):319–343, 2004.

[12] W. Ye and R. Ernst. Embedded Program Timing Analysis Based on Program and Architecture Classification. *Technical Report CY-96-3 from the Braunschweig University of Technology*, Oct. 1996.