Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 162404, 12 pages
doi:10.1155/2012/162404

Research Article

HwPMI: An Extensible Performance Monitoring Infrastructure for Improving Hardware Design and Productivity on FPGAs

Andrew G. Schmidt,[1] Neil Steiner,[1] Matthew French,[1] and Ron Sass[2]

[1] *Information Sciences Institute, University of Southern California, 3811 North Fairfax Drive, Suite 200, Arlington, VA 22203, USA*
[2] *Reconfigurable Computing Systems Lab, ECE Department, UNC Charlotte, 9201 University City Boulevard, Charlotte, NC 28223, USA*

Correspondence should be addressed to Andrew G. Schmidt, aschmidt@isi.edu

Received 4 May 2012; Revised 21 September 2012; Accepted 3 October 2012

Academic Editor: René Cumplido

Designing hardware cores for FPGAs can quickly become a complicated task, difficult even for experienced engineers. With the addition of more sophisticated development tools and maturing high-level language-to-gates techniques, designs can be rapidly assembled; however, when the design is evaluated on the FPGA, the performance may not be what was expected. Therefore, an engineer may need to augment the design to include performance monitors to better understand the bottlenecks in the system or to aid in the debugging of the design. Unfortunately, identifying what to monitor and adding the infrastructure to retrieve the monitored data can be a challenging and time-consuming task. Our work alleviates this effort. We present the Hardware Performance Monitoring Infrastructure (HwPMI), which includes a collection of software tools and hardware cores that can be used to profile the current design, recommend and insert performance monitors directly into the HDL or netlist, and retrieve the monitored data with minimal invasiveness to the design. Three applications are used to demonstrate and evaluate HwPMI's capabilities. The results are highly encouraging as the infrastructure adds numerous capabilities while requiring minimal effort by the designer and low resource overhead to the existing design.

1. Introduction

As hardware designers develop custom cores and assemble Systems-on-Chip (SoCs) targeting FPGAs, the challenge of the design meeting timing, fitting within the resource constraints, and balancing bandwidth and latency can lead to significant increases in development time. When a design does not meet a specific performance requirement, the designer typically must go back and manually add more custom logic to monitor the behavior of several components in the design. While this performance information can be used to better understand the inner workings of the system, as well as the interfaces between the subcomponents of the system, identifying and inserting infrastructure can quickly become a daunting task. Furthermore, the addition of the monitors may change the original behavior of the system, potentially obfuscating the identified performance bottleneck or design bug.

In this work, we focus on an extensible set of tools and hardware cores to enable a hardware designer to insert a minimally invasive performance monitoring infrastructure into an existing design, with little effort. The monitors are used in an introspective capacity, providing feedback about the design's performance under real workloads, while running on real devices. This paper presents our Hardware Performance Monitoring Infrastructure (HwPMI), which is designed to ease the identification, insertion, and retrieval of performance monitors and their associated data in developed systems.

The motivation for the creation and evaluation of this infrastructure stems from the inherent need to insert monitors into existing designs and to retrieve the data with minimal invasiveness to the system. Over the last several years we have been assembling a repository of performance monitors as new designs are built and tested. To increase a designer's productivity, we have put together a suite of software tools aimed at profiling existing designs and recommending and/or inserting performance monitors and the necessary hardware and software infrastructure. This

work also leverages existing open-source work by including Torc (Tools for Open Reconfigurable Computing) to provide an efficient backend for reading, writing, and manipulating designs in EDIF format [1].

Included in HwPMI are the specific monitors, such as timers, state-machine trackers, component utilization, and so forth, along with a sophisticated monitoring network to retrieve each monitor's data all while requiring little user intervention. To evaluate HwPMI, three use cases show how existing designs can utilize HwPMI to quickly integrate and retrieve monitoring data on running systems. Moreover, HwPMI is flexible enough to support both high performance reconfigurable computing (HPRC), running on the Spirit cluster as part of the Reconfigurable Computing Cluster (RCC) [2] project at the University of North Carolina at Charlotte, and embedded reconfigurable systems running on a single board with limited compute resources. Several interesting results are discussed which support the importance of such a monitoring infrastructure. Finally, the tools and hardware cores presented here are being prepared for an open-source release in the hope of increasing and diversifying the types of monitors and the systems that will be able to utilize HwPMI.

The remainder of this paper is organized into the following sections: in Section 2 the background and related works are discussed. Section 3 details HwPMI's design and implementation, while the results and experiences of integrating the system into three applications are discussed in Section 4. Finally, in Section 5 the conclusion and future work are presented.

2. Background and Related Work

As FPGA resources increase in number and diversity with each device generation, researchers are exploring architectures to outperform previous implementations and to investigate new designs that were previously limited by the technology. Unfortunately, a designer trying to exploit the inherent parallelism of FPGAs is often faced with the nontrivial task of identifying system bottlenecks, performance drains, and design bugs in the system. The result is often the inclusion of custom hardware cores tasked with monitoring key locations in the system, such as interfaces to the network, memory, finite-state machines, and other resources such as FIFOs or custom pipelines.

This is especially true in the rapidly growing field of High Performance Reconfigurable Computing (HPRC). There are several research projects underway that investigate the use of multiple FPGAs in a high-performance computing context: RAMP, Maxwell, Axel, and Novo-G [3–6]. These projects and others, like the RCC Spirit cluster, seek to use many networked FPGAs to exploit the FPGA's potential on-chip parallelism, in order to solve complex problems faster than before.

2.1. Productivity and Monitoring Tools. Tools of some form are needed to help the designer manage the complexities associated with hardware design, such as timing requirements, resource limitations, routing, and so forth. FPGA vendors provide many tools beyond synthesis and implementation that reduce development time, including component generators [7] that build mildly complex hardware cores, that is, single-precision floating point units and FIFOs. System generator tools like the Xilinx Base System Builder (BSB) Wizard [8] and the Altera System-on-Programmable-Chip (SoPC) Builder [9] help the designer construct a customizable base system with processors, memory interfaces, buses, and even some peripherals like UARTs and interrupt controllers. There are even tools that can help a designer debug running hardware in the same manner as with logic analyzers in a microelectronics lab [10, 11]. However, these virtual logic analyzers do not provide runtime feedback on the system's performance, and furthermore require specialized external equipment. In the case of ChipScope, each FPGA must be connected to a computer through JTAG. This limits its use in large scale designs, where debugging hundreds to thousands of FPGAs is a daunting task. While JTAG can support multiple devices in a single chain, there is additional latency as the number of devices in the chain increases. Of course, if JTAG were the only option, HwPMI could be inserted into the JTAG chain; at this time no such integration has been performed.

There are also several projects investigating ways to monitor FPGA systems. The Owl system monitoring framework presented in [12] uses hardware monitors in the FPGA fabric to snoop system transactions on memory, cache, buses, and so forth. This is done to avoid the performance penalty and intrusiveness of software-based monitoring schemes. Along similar lines [13] presents a performance analysis framework for FPGA-based systems that automates application-specific run-time measurements to provide a more complete view of the application core's performance to the designer. Source level (HDL) instrumentation is used to parse code and insert logic to extract desired data at runtime. The University of Florida proposed CARMA [14] as a framework to integrate hardware monitoring probes in designs at both the hardware and software layer. TimeTrial [15, 16] explores performance monitoring for streaming applications at the block level, which keeps it language agnostic and suitable for dealing with different platforms and clocks. FPGAs have also been used to emulate and speed up netlist level fault injection and fault monitoring frameworks to build resilient SoCs [17].

2.2. Tools for Open Reconfigurable Computing. Torc (Tools for Open Reconfigurable Computing) is a C++ open-source framework designed to simplify custom tool development and enable new research [18]. Torc's capabilities are built upon industry standard file formats, including EDIF and XDL, and come with support for a broad range of Xilinx devices. Research into reconfiguration or CAD tools with Torc can be validated in hardware or compared to mainstream tool performance. Developing and debugging these capabilities from scratch would take a significant amount of effort, but Torc provides them for free. Furthermore, the user can pick and choose the parts of Torc that are of interest to them, and ignore the rest. In fact, Torc was developed for precisely the kinds of research purposes that HwPMI is

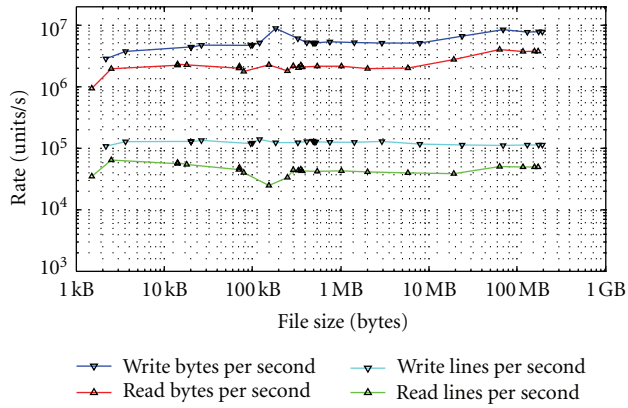

Figure 1: Torc's Generic Netlist API I/O Performance. This log-log plot shows reasonably linear I/O performance for EDIF file sizes ranging from 1 KB to 175 MB. On a 2.8 GHz quad-core Xeon, the API reads 45-thousand lines per second, and 2.3 Megabytes per second on average. The largest of these files contains over 150,000 instances and over 200,000 nets. Shape differences between the two curves are most likely due to different name lengths in the EDIF files.

aimed at addressing—situations in which the mainstream tools could not provide the required functionality.

Torc consists of four core Application Programming Interfaces (APIs) and a collection of CAD tools built upon them. The APIs are the Generic Netlist API, the Physical Netlist API, the Device Architecture API, and the Bitstream Frames API. The main CAD tools are the router and placer. For HwPMI, the appeal of Torc's generic netlist is that it allows us to insert performance monitors without having to modify the design's HDL source. The Generic Netlist API supports netlists that are not mapped to physical hardware and provides full EDIF 2.0.0 reading, writing, and editing capability. The internal object model is flexible enough to support other netlist formats, if parsers and exporters are provided. Because the Generic Netlist API supports generic EDIF, it is usable for Xilinx FPGAs, non-Xilinx FPGAs, ASICs, or even circuit boards.

Early versions of HwPMI interacted with VHDL source to identify and insert performance monitors and the necessary infrastructure into existing designs. Torc is being added to expand beyond VHDL and to ensure that the original source remains unmodified after it has been profiled and evaluated—only the resulting synthesized netlists are modified. Another reason for migrating to Torc is its efficiency and scalability: a plot of EDIF read and write performance is provided in Figure 1. Using Torc to interact with EDIF has been shown to be far more efficient than using VHDL parsing and insertion tools.

Another well-established CAD tool for reconfigurable computing is VPR [19, 20], a place-and-route tool that remains widely used more than a decade after its inception, and now forms the base of the broader VTR [21]. However, VPR has traditionally not supported EDIF, XDL, or actual device architectures. Some of those pieces—EDIF in particular—are available from Brigham Young University (BYU), albeit in Java rather than C++. More recently, BYU developed an open-source Java tool named RapidSmith that fully supports XDL and Xilinx device databases [22].

3. Design

The performance monitoring infrastructure assembled as part of this work builds upon our previous research in the area of resilient high-performance reconfigurable computing. In [23] a System Monitoring Infrastructure was developed and a proof-of-concept was presented to address the question: "how do we know when a node has failed?" In [24] the System Monitor functionality was significantly improved, adding a Context Interface (CIF) along with dedicated hardware cores for checkpoint/restart capability of both the processor and hardware core's state. In addition, a framework for performance monitoring was presented in [25], which this work extends to provide a designer with the capability of inserting specific performance monitors into an existing hardware design. Specifically, this article extends our previous works with a more thorough design and evaluation of HwPMI and further discusses how Torc is incorporated for efficient netlist manipulations.

Unlike conventional approaches where a designer must manually create and insert monitors into their design, including the mechanisms to extract the monitored results, this work analyzes existing designs and generates the necessary infrastructure automatically. The result is a large repository of predesigned monitors, interfaces, and tools to aid the designer in rapidly integrating the monitoring infrastructure into existing designs. This work also provides the designer with the necessary software infrastructure to retrieve the performance data at user defined intervals during the execution of the system.

3.1. Hardware Performance Monitor Infrastructure. The HwPMI tool flow, which will be discussed within this section, consists of several stages, as depicted in Figure 2. In addition to the tools, the monitoring infrastructure consists of several hardware cores that, for the high-performance reconfigurable computing (HPRC) system, spans a variety of elements both in the system and across the cluster. The infrastructure to support the HPRC monitoring system is

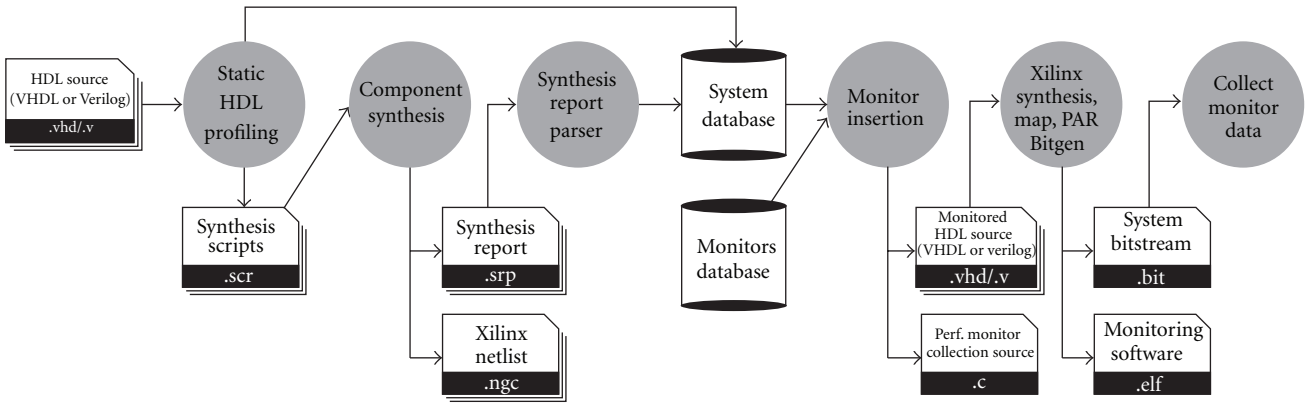

Figure 2: Hardware Performance Monitoring Infrastructure's Tool Flow.

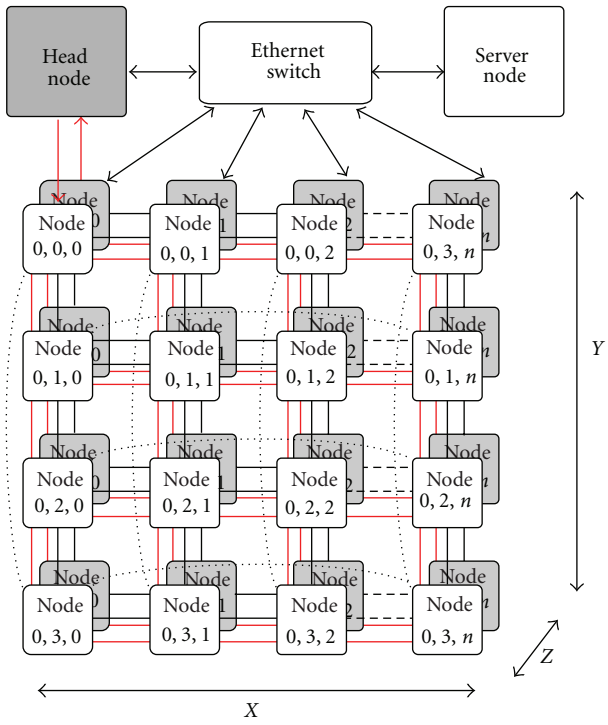

Figure 3: Block diagram of Spirit cluster's HwPMI.

comprised of three types of nodes and two networks, as illustrated in Figure 3. Node types include a server node, a head node, and many worker nodes. With the exception of the server, all nodes are Xilinx ML410 development boards with Virtex4 FX60 devices. 64 FPGAs are connected via six direct-connect links to configure the Spirit cluster's primary network as a custom high-speed 3-dimensional torus [26]. Two more links on the custom network board are used to form the sideband network, which is used by the HwPMI to send and receive performance monitoring commands and data. More details regarding the networks are presented in [24].

Each worker node is running an application-specific SoC which includes the HwPMI hardware cores. These cores can be seen in Figure 4 as the System Monitor Hub, HwPMI Interfaces, Context Interface, Performance Monitor Hub, and Performance Monitor Cores. The System Monitor Hub acts as an intermediary to decode incoming requests for performance data. Each hardware core connects to the System Monitor Hub via a software-generated Context Interface (CIF). The CIF connects to the Performance Monitor Hub which in turn aggregates all of the performance monitor core data within the hardware core.

Initial HwPMI development was targeted to support high-performance reconfigurable computing systems, such as Spirit. However, the tools and techniques are also easily adapted to support more traditional embedded system development with FPGAs. In fact, the sideband network can be replaced with a bus interface to give an embedded system access to its own performance monitoring data. While this introspective monitoring does add to the runtime overhead of the system, designers can now specify the interface mechanism to HwPMI. PowerPC 405, PowerPC 440, and MicroBlaze systems are supported through interfaces with the Processor Local Bus. An embedded system example is shown in Figure 5, where a separate SoC provides independent monitoring of hardware cores. In this case, resources are required for the extra soft-processor, buses, and peripheral IP cores, in addition to the monitoring infrastructure. The benefit of this approach is that no modifications are necessary in the original Device Under Test (DUT)—no changes to the software running on the DUT's processor—so the performance overhead is minimized.

3.2. Static HDL Profiling. The process of identifying performance monitors to be inserted into an existing design begins with Static HDL Profiling, shown in Figure 2. To start, HwPMI parses the existing hardware design's HDL files in order to collect information pertaining to the construction of the system. This includes not only the design's modular hierarchy, but also the specific interfaces between components. For example, a hardware core may consist of several FIFOs, BRAMs, and finite-state machines, as well as a slave interface to the system bus. The Static HDL Profiler identifies these components and interfaces in order to assemble a list of recommended performance monitors to be inserted.

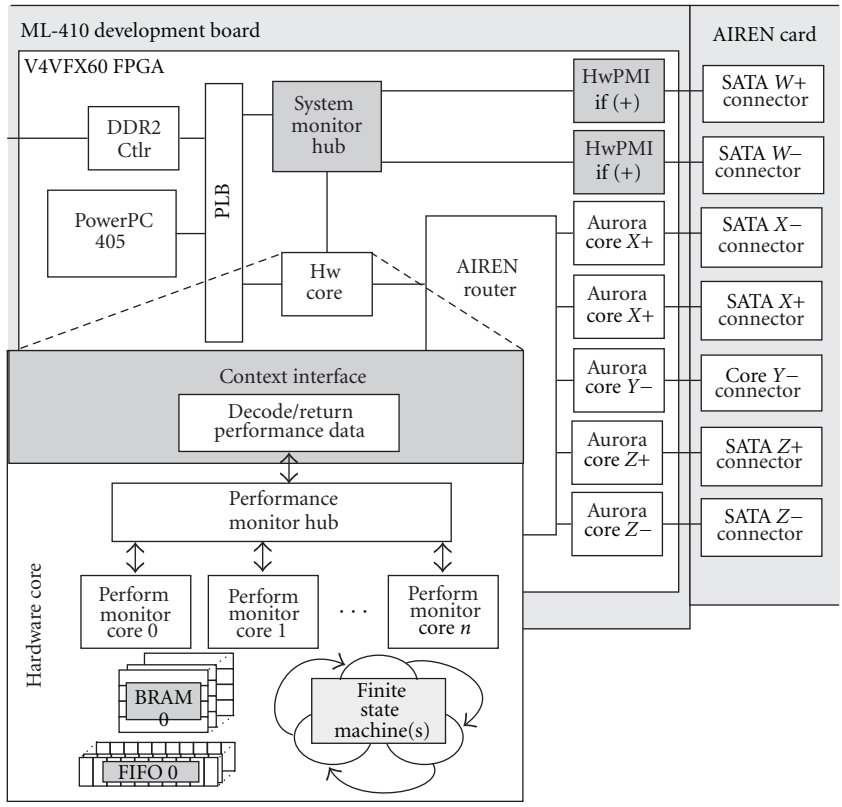

Figure 4: Block diagram of FPGA node's HwPMI.

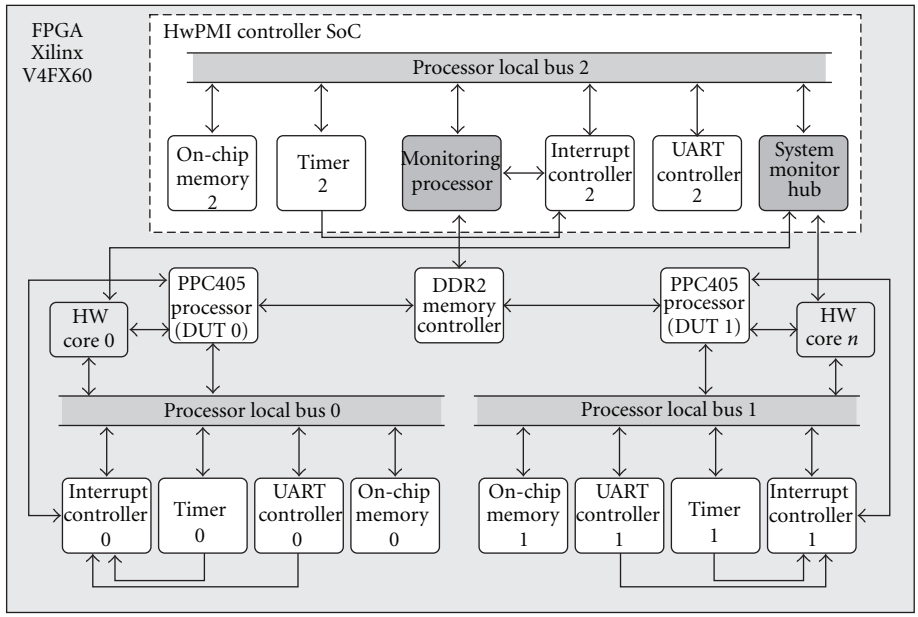

Figure 5: Block diagram of HwPMI System-on-Chip running on embedded device under test.

Static HDL profiling is similar in principle to software profiling (i.e., gprof) in that all of the critical information is collected in advance, and the system runtime performance information is captured during execution.

The Static HDL Profiler is comprised of three software tools written in Python, to more autonomously profile the original design. Presently, HwPMI supports the Xilinx Synthesis Tool (XST) and designs written in VHDL; however, work is underway to extend beyond XST and VHDL through the use of Torc. The first tool, HwPMI Core Parser, provides parsing capabilities for VHDL files. The designer invokes the parser on the specific design through the command

```
Collatz_core_0_wrapper:
  Interfaces:
    (1) plb_slave
  Components:
    (1) plb_slave_attachment
    (2) user_logic
    (3) collatz_kernel
          Registers (collatz_kernel):
            64-bit register for signal <n>
            32-bit register for signal <steps_i>
          FSMs:
            <FSM_0> for signal <fsm_cs>
  Signals:
    (1) PLB IPIC Signals
```

Figure 6: Sample output of HwPMI System Analyzer tool on the Collatz Design, identifying the components, registers, statements, and interfaces to the core.

```
Recommended Performance Monitors:
Top-Level Entity: collatz_core
collatz_core:
  plb46_slave_single_i
    NONE
  user_logic
    Utilization Monitor
    Interrupt Timer Monitor
    PLB SLV IPIF Monitor
  collatz_kernel
    Finite State Machine Profiler
```

Figure 7: Sample output of performance monitor recommendation tool.

line and can specify a specific VHDL source to evaluate. The parser identifies the entity's structure in terms of ports, signals, finite-state machines, and instantiated components. The parser works by analyzing the VHDL source file and uses pattern matching to decompose the component into its basic blocks. The parser is only responsible for the identification of the VHDL component's structure. The results are then passed into a Python Pickle for rapid integration with the remaining tools.

Next, the HwPMI System Analyzer tool iteratively parses the design to identify the different interfaces, such as bus slaves, bus masters, direct memory access, and Xilinx LocalLink. This is done at a higher level than the HwPMI Core Parser which more specifically analyzes individual IP Cores. Figure 6 shows the output of the HwPMI System Analyzer for one of the systems evaluated in this work, the Collatz Design. More commonly a designer would use the System Analyzer because it can support iterating through an entire design, once given a list of all of the source files. On the command line the designer invokes the tool with a project file that lists all of the VHDL source file locations. The user also specifies the top-level entity for the design.

To support Xilinx Platform Studio (XPS) IP core development, the HwPMI Parse PCORE tool is used to parse Xilinx PCORE directory files: The Microprocessor Description (MPD), Peripheral Analysis Order (PAO), and Black Box Description (BBD) files, along with any Xilinx CoreGen (XCO) project files. This enables a designer to migrate profiled cores to other XPS systems with minimal effort. Furthermore, monitors can be written based on the Xilinx CoreGen project files to provide monitoring of components such as generated FIFOs, memory controllers, or floating point units, if so desired.

3.3. Component Synthesis. The next stage is Component Synthesis where the original hardware design is synthesized prior to any insertion of performance monitors. The purpose of synthesizing the design at this point is to retrieve additional design details from the synthesis reports including subcomponents, resource utilization, timing requirements, and behavior. This leverages the synthesis tool output to supplement the Static HDL Profiling stage by more readily identifying finite state machines and flip-flops in the design. All of the configuration information and synthesis results are available for performance monitor recommendation/insertion.

Three tools have been developed to specifically support the designer in the Component Synthesis stage. These tools automatically synthesize, parse, and aggregate the individual component utilization, resource utilization, and timing information data for the designer. The first tool is the Iterative Component Synthesis tool which runs the synthesis scripts for each of the components in the design. The second tool is the Parse Component Synthesis Reports tool: it runs after all of the system components have been synthesized, and collects a wealth of information about each component from the associated synthesis report file (SRP). This information includes the registers, FIFOs, Block RAMs, and finite-state machines (FSM), in addition to all subcomponents. The third tool is the Aggregate System Synthesis Data tool which is used to aggregate all of the data collected as part of the Parse Component Synthesis Reports tool. These tools collectively identify the system's interconnects, processors, memory controllers, and network interfaces, in addition to the designer's custom compute cores.

3.4. Insertion of Performance Monitors. At this point the design has been analyzed in order to recommend specific performance monitors for insertion. The designer can choose to accept any number of these recommendations, from a report like that shown in Figure 7 The monitors are stored in a central repository which can be augmented by the designer if a specific monitoring capability is not available. The monitors all are encapsulated by a Performance Monitor Interface, shown in Figure 8, which connects the monitor to the Performance Monitor Hub and includes a finite-state machine to retrieve the specific performance monitor data and forward it to the hub. To aid in the insertion of HwPMI into existing hardware designs, a software tool has been

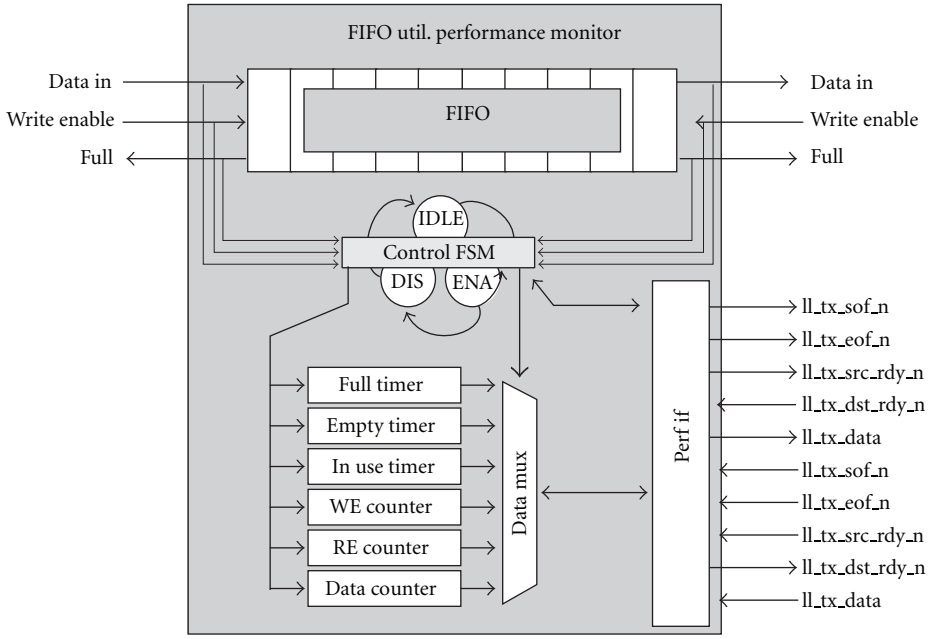

Figure 8: Block diagram of performance monitor interface connecting to a simple configurable timer monitor.

developed, the HwPMI System Insertion tool. The purpose of this tool is to insert the System Monitor Hub and Sideband Network Interface cores into the top-level design. The tool inserts the monitors directly into the VHDL source, prior to synthesis, MAP, and PAR.

It is important to emphasize that the HwPMI flow does not intelligently insert a subset of monitors when the available resources are depleted. Future work is looking into ways to weight monitors such that HwPMI can insert more important monitors; however, presently HwPMI recommends the available monitors that can be inserted into the design and it is up to the designer to choose the subset that will provide the best feedback versus resource availability trade-off.

When a performance monitor is created there is a set of criteria that must also be included to allow the recommendation to take place. For example, there is a PLB Slave Interface performance monitor which specifically monitors reads and writes to the hardware core's slave registers. All signals are identified during profiling, but until these signals are matched against a list of predetermined signals, there is no specific way to identify when those signals are being written to. Another example considers finite-state machines: once an FSM has been identified by the system, it is trivial for the respective performance monitor to be recommended for insertion. The actual insertion of the performance monitors is done at the HDL level. Each performance monitor's entity description and instance are automatically generated and inserted in the HDL.

3.5. Torc Netlist Modifications. The initial development of HwPMI focused on parsing designs written in VHDL and inserting monitors directly into the VHDL source. The advantage of this approach is the portability of the monitored design. However, by leveraging tools such as Torc, the insertion can be performed at the netlist level. The HwPMI tool flow has now been augmented to support inserting the monitoring infrastructure into either the VHDL source or into synthesized netlists, based on a user parameter. Continued work is underway to perform the netlist profiling with Torc instead of relying on the HDL parsing tools. Specifically, Torc inserts the monitors into the EDIF design. Figure 9 illustrates how Torc is currently used in the HwPMI flow to avoid modifying the VHDL source. The HwPMI flow remains identical throughout the initial stages, but once the monitors have been selected for insertion, Torc is used to merge them into the synthesized netlists. Torc provides a fast and efficient mechanism to generate modified design netlists with the monitoring infrastructure inserted.

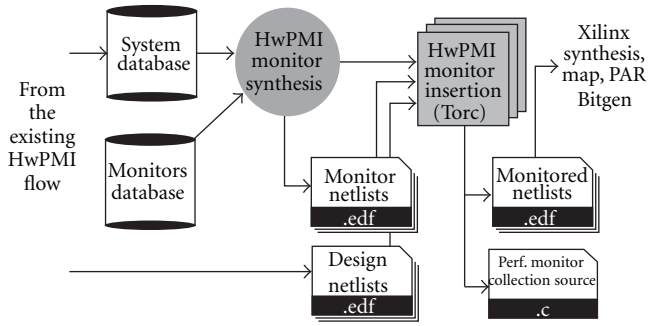

Figure 9: Insertion of Torc into Hardware Performance Monitoring Infrastructure's Tool Flow.

3.6. Retrieval of Monitored Data. Once the design is running, it is necessary to retrieve the performance monitoring data with minimal invasion to the system. This is accomplished through the use of the sideband network in the HPRC system

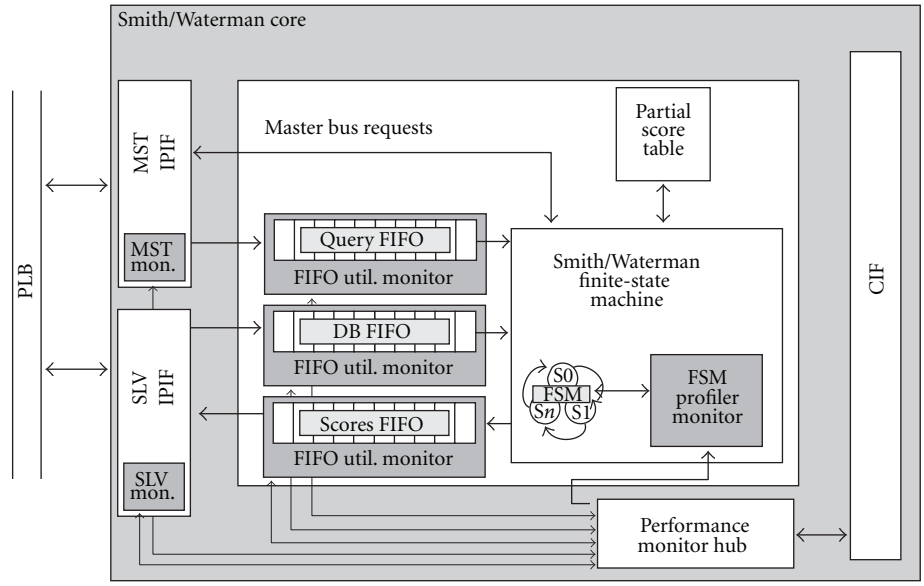

Figure 10: Smith/Waterman core's performance monitors.

or the HwPMI SoC in an embedded system. The head node issues requests to retrieve data from a node, core, or a specific hardware monitor anytime the application is running. To aid in the retrieval, the Performance Monitor Collection tool assembles the entire system's performance monitoring data structure for the head node to use for runtime data collection. This data is stored in a *C* struct that is generated for the specific design. Also available are subroutines for the head node to automatically collect and report each monitor's data back to the designer. Presently, the data that is retrieved must be manually evaluated by the designer to make design modifications, if deemed necessary. In HPRC systems this monitoring data can be fed into SDAflow, a tool developed in [27] to reallocate the resource utilization along with memory and network interfaces.

4. Results

Three applications are used to demonstrate our HwPMI tool flow: single precision matrix-matrix multiplication, a hardware implementation of the Smith/Waterman `FLOCAL_ALIGN()`, and a hardware implementation of the Collatz Conjecture core. This section will highlight different use cases of HwPMI in these applications.

4.1. Matrix-Matrix Multiplication. Matrix-Matrix Multiplication (MMM) is a basic algebraic operation where two matrices, *A* and *B*, are multiplied together to form the resultant matrix *C*. This operation is highly parallel but is very demanding upon the memory hierarchy. During Static HDL Profiling the MMM hardware core successfully identified the following subcomponents: `plb_slave_ipif`, `user_logic`, `mac_array`, thirty-two `mac_units`, and eighteen 32-bit $\times$ 512 deep FIFOs. From these components the static HDL parser correctly identified 25 software-addressable registers that were connected via the PLB's IPIF by tracing from the PLB's address and data signals to the core's registers set and accessed by these signals. These registers are used partly for control and also for data inputs and outputs which can be monitored via HwPMI to provide processor and hardware accelerator interface efficiency.

Three sets of performance monitors were selected and inserted into the MMM core: Firstly, for the PLB slave IPIF, to monitor the efficiency of the transfers from the processor. Secondly, for the eighteen FIFOs to monitor capacity and determine if more buffer resources should be allocated in the future to improve performance. And thirdly, for the utilization of the core itself, to monitor performance and determine how much time the core spends on actual computation versus I/O. The results indicate the largest bottleneck in the current design is the PLB slave IPIF. The processor spends over 98% of the total execution time transferring the matrix data and results into and out of the MMM hardware core. Furthermore, the results for the FIFOs showed very low overall utilization, which indicates that the FIFO depth can be reduced.

4.2. Smith/Waterman. The second design evaluated with HwPMI is a hardware-accelerated implementation of the Smith/Waterman algorithm, commonly used in protein and nucleotide sequence alignments [28]. The particular implementation on the FPGA was developed as a proof of concept [29] for accelerating the `FLOCAL_ALIGN()` function of the *SSEARCH* program—an implementation of the Smith/Waterman algorithm from the *FASTA35* code package [30].

From the Static HDL Profiling and Component Synthesis stages, six performance monitors were identified for inclusion into the Smith/Waterman hardware core. Figure 10 shows a high-level block diagram of the performance monitors in their locations relative to the Smith/Waterman hardware core.

Table 1: Performance monitor results for PLB SLV IPIF.

Register	Original		Modified	
name	# Reads	# Writes	# Reads	# Writes
Control_reg	0	186	0	186
Core_status	29540	0	29540	0
aa1	0	2095745	0	2095745
n1	0	2095838	0	93
n0	0	2095838	0	93
GG	0	2095838	0	93
HH	0	2095838	0	93
f_str_waa_s	0	2095838	0	93
score	186	0	186	0
ssj	0	2095838	0	93
Counter_idle	186	0	186	0
Counter_work	186	0	186	0

```
ssearch->aa1 = *aa1p;
if (only_once == 0) {
   ssearch->n1 = n1;
   ssearch->n0 = n0;
   ssearch->GG = GG;
   ssearch->HH = HH;
   ssearch->f_str _waa_s = PWA_BASE;
   ssearch->ssj = SS_BASE;
   /* ADDED MISSING GUARD HERE: */
    only_once = 1;
}
```

Figure 11: Modification made to original dropgsw2.c.

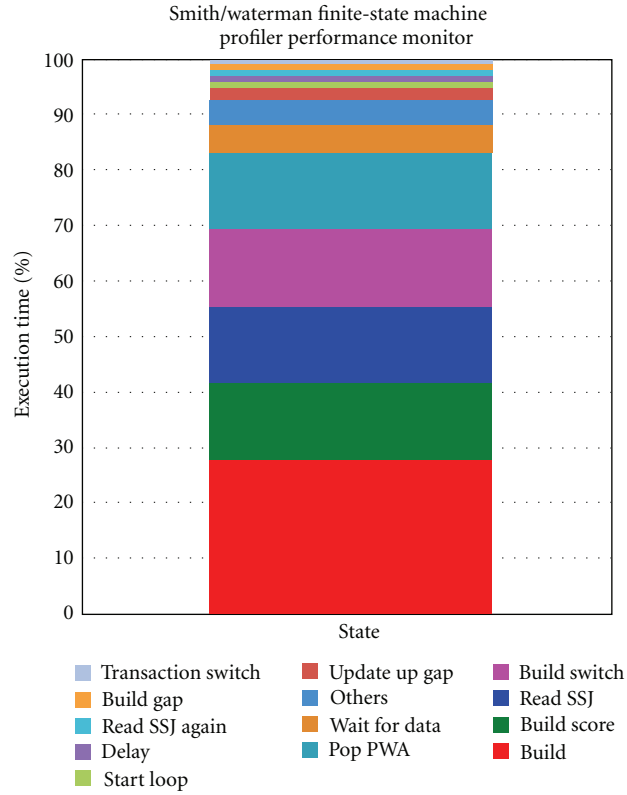

Figure 12: Smith/Waterman core's FSM profiler monitor results.

The first performance monitor identifies the number of writes to the software-addressable registers in the Smith/Waterman hardware core via the PLB Slave interface (PLB SLV IPIF), the results of which are listed in Table 1. In addition to the register name, number of reads and number of writes, Table 1 also presents these reads and writes when run in Original and Modified modes. The performance monitoring data indicated that several software registers were being written to unnecessarily, and the modified version of the application eliminates these extra writes. This demonstrates the benefit of HwPMI for debugging: the results quickly revealed that the software application was missing a guard, as shown in Figure 11.

Also identified by HwPMI were additional performance monitors to evaluate the PLB Master interface (PLB MST IPIF), which identified that only off-chip memory transactions were performed by the core. Moreoever, the off-chip memory transactions were 118,144 read-only requests, which is a significant number of transfers and warrants the evaluation of a DMA interface. The designer could also adapt the core to leverage an alternative interface, such as the Native Port Interface (NPI), to reduce memory access latency, increase bandwidth, and reduce the PLB contention.

An FSM profiler performance monitor was added that provides feedback in the form of a histogram, to identify the percentage of time each FSM state is active. Figure 12 presents the breakdown of the time in each state. This shows that `BUILD` is the longest running state, accounting for 27.67% of the execution time. The next four states, BUILD_SCORE, READ_SSJ, BUILD_SWITCH, POP_PWAA, each occupies ~13.8%. Thirteen of the remaining states account for less than 1% each, and have been group together in the `OTHERS` category. Overall, this profiling data should

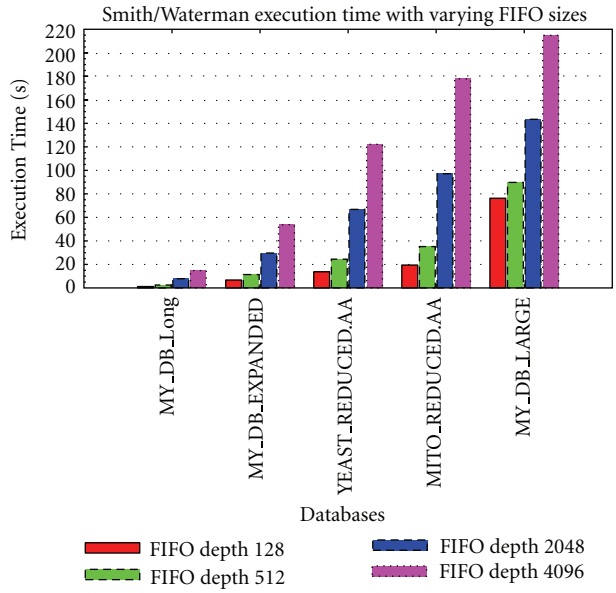

Figure 13: Smith/Waterman's performance with varying FIFO depths.

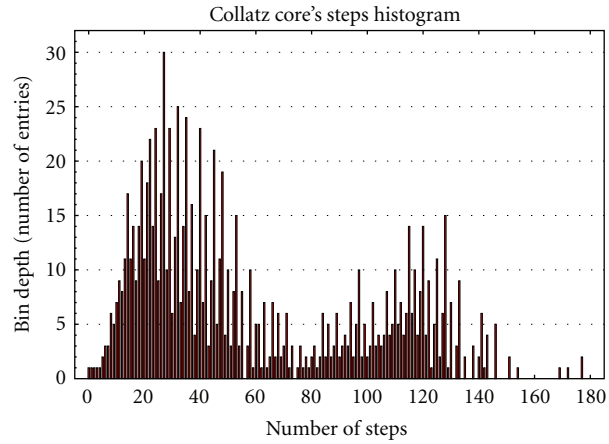

Figure 14: Collatz core's histogram of steps.

more quickly focus the designer's attention on the BUILD state, to determine if there is a more efficient way to implement this state.

Another useful feature of HwPMI is its ability to evaluate designs with different resource utilizations. Designers often find themselves adding buffers or FIFOs into designs without knowing a priori how large they should be. In these cases, HwPMI can collect run-time data as a designer modifies the FIFO depth. Sometimes these modifications can reveal interesting design tradeoffs, such as those shown in Figure 13, where a design utilizing smaller parallel buffers runs more efficiently than one using fewer larger buffers. In order to collect this data, a designer simply modifies the FIFO depth, and HwPMI collects the information at user defined intervals.

4.3. Collatz Conjecture. The third application used in our HwPMI evaluation is Collatz. The Collatz Conjecture states that given a natural number, it is possible reduce that number to one, by either dividing by two when even, or multiplying by three and adding one when odd [31]. For even numbers the resulting calculation reduces the size in half; however, for odd numbers the new number produced is greater than the original number. Thus, it is not obvious that it will converge to one. For example, given a small number such as $n = 3$, seven iterations are required to reduce n to one.

The performance monitor data is collected with the assistance of HwPMI. In addition to the utilization and interface performance monitors, an additional monitor was added that yielded a highly beneficial result. This is an example of supplemental data collection. When a designer needs to collect additional information, HwPMI offers the ability to add custom monitors without the need to augment how the system will retrieve the data. This can be especially useful for quick one-off data that might only be useful for a short period of time. Rather than manually adding the infrastructure to the original core, only to remove it later—or retain it and waste resources—HwPMI can collect the data quickly and efficiently. Figure 14 shows a histogram of the number of steps taken for each input number to be reduced to one.

Another interesting monitor is the processor interrupt monitor. For latency sensitive applications with processor-to-hardware core communication, an interrupt is often used. However, configuring the interrupt or optimizing the interrupt service routines is critical. In the case of Collatz the time for the processor to respond to a single interrupt was measured as $\approx 11.12\,\mu$s.

4.4. Resource Utilization. Finally, we present the resource utilization of HwPMI. Our goal is to be minimally invasive both in terms of processing overhead and resource utilization overhead. Listed in Table 2 are some of the performance monitor cores that have been used in the three applications. This includes varying sizes of the monitors to show the overall scalability. While the individual monitor's utilization is heavily dependent on the function of the monitor, we show that with very low overhead HwPMI can be added to a design. Especially when compared to a design that requires the addition of a bus interface to a hardware core for performance data retrieval, HwPMI offers an attractive alternative. Furthermore, the overhead of the hardware monitor interface, which is 34 Slice FFs and 74 4-input LUTs on the V4FX60 FPGA, makes the standard monitor infrastructure of HwPMI very appealing compared to custom monitoring cores.

5. Conclusion

The Hardware Performance Monitoring Infrastructure (HwPMI) presented in this work expedites the insertion of a minimally invasive performance monitoring networks into existing hardware designs. The goal is to increase designer

Table 2: Example of HwMPI's resource utilization on V4FX60.

Component	Configuration	FFs (%)	LUTs (%)
Performance monitor hub	1 port	14 (0.03%)	70 (0.14%)
Performance monitor hub	2 ports	17 (0.03%)	78 (0.15%)
Performance monitor hub	4 ports	21 (0.04%)	153 (0.30%)
Performance monitor hub	8 ports	21 (0.04%)	250 (0.49%)
Performance monitor hub	16 ports	23 (0.05%)	419 (0.83%)
Timer monitor	1 32-bit timer	37 (0.07%)	96 (0.19%)
Match counter monitor	1 64-bit counter	67 (0.13%)	109 (0.22%)
Match counter monitor	2 64-bit counters	132 (0.26%)	207 (0.41%)
Match counter monitor	16 64-bit counters	1034 (2.05%)	1593 (3.15%)
FIFO monitor	1 32-bit FIFO	402 (0.80%)	594 (1.17%)
Histogram monitor	512 Bins	20 (0.04%)	3207 (6.34%)
Finite state machine monitor	12 states	775 (1.53%)	1266 (2.50%)
Finite state machine monitor	64 states	4116 (8.14%)	6332 (12.52%)
System monitor hub	1 port (1 Hw Core)	212 (0.42%)	513 (1.01%)
System monitor hub	2 ports (2 Hw Cores)	213 (0.42%)	565 (1.12%)
System monitor hub	4 ports (4 Hw Cores)	216 (0.43%)	691 (1.37%)
System monitor hub	8 ports (8 Hw Cores)	224 (0.44%)	911 (1.80%)
System monitor hub	16 port (16 Hw Cores)	230 (0.45%)	1369 (2.71%)

productivity by analyzing the existing design and automatically inserting monitors with the necessary infrastructure to retrieve the monitored data from the system. As a result of HwPMI the designer can focus on the development of the hardware core rather than trying to include front-end application support to monitor performance. Toward this goal, a collection of hardware cores have been assembled, and a series of software tools have been written to parse the existing design and recommend and/or insert hardware monitors directly into the source HDL.

HwPMI also integrates with an existing sideband network to retrieve the performance monitor results in High Performance Reconfigurable Computing without requiring modifications to the original application. Embedded systems can leverage HwPMI through a dedicated System-on-Chip controller which reduces run-time overhead on existing processors in the system. This work demonstrated HwPMI's capabilities across three applications, highlighting several unique features of the infrastructure.

This work also leverages Torc to provide netlist manipulations quickly and efficiently, in place of the original HDL modifications [25] which were limited to VHDL and were less efficient. Future work will integrate Torc more fully into the tool flow, replacing the static HDL analysis in favor of netlist analysis. In addition, HwPMI is being prepared for an open-source release which includes the tool flow and hardware IP core repository of both the monitoring infrastructure and performance monitor cores.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001-11-C-0041. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA). DARPA Distribution Statement A. Approved for Public Release, Distribution Unlimited.

References

[1] "Torc: Tools for Open Reconfigurable Computing," 2012, http://torc.isi.edu/.

[2] R. Sass, W. V. Kritikos, A. G. Schmidt et al., "Reconfigurable Computing Cluster (RCC) project: investigating the feasibility of FPGA-based petascale computing," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '07)*, pp. 127–140, IEEE Computer Society, April 2007.

[3] D. Burke, J. Wawrzynek, K. Asanovic et al., "RAMP Blue: implementation of a Manycore 1008 Processor System," in *Proceedings of the Reconfigurable Systems Summer Institute 2008 (RSSI '08)*, 2008.

[4] R. Baxter, S. Booth, M. Bull et al., "Maxwell—a 64 FPGA supercomputer," in *Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS '07)*, pp. 287–294, August 2007.

[5] P. P. Kuen Hung Tsoi, A. Tse, and W. Luk, "Programming framework for clusters with heterogeneous accelerators," in *International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2010.

[6] NSF Center for High Performance Reconfigurable Computing (CHREC), "Novo-g: Adaptively custom research supercomputer," April 2005.

[7] Xilinx, Inc., "Xilinx CORE Generator System," July 2011, http://www.xilinx.com/tools/coregen.htm.

[8] Xilinx, Inc., *Embedded System Tools Reference Manual EDK 10.1*, 2010.

[9] Altera Corporation, *System-on-Programmable-Chip (SOPC) Builder User Guide (UG-01096-1.0)*, 2010.

[10] Xilinx, Inc., "ChipScope Pro and the Serial I/O Toolkit," http://www.xilinx.com/tools/cspro.htm.

[11] Altera Corporation, "Design Debugging Using the SignalTap II Embedded Logic Analyzer," http://www.altera.com/literature/hb/qts/qts_qii53009.pdf.

[12] M. Schulz, B. S. White, S. A. McKee, H.-H. S. Lee, and J. Jeitner, "Owl: next generation system monitoring," in *Proceedings of the 2nd Conference on Computing Frontiers*, pp. 116–124, ACM, May 2005.

[13] S. Koehler, J. Curreri, and A. D. George, "Performance analysis challenges and framework for high-performance reconfigurable computing," *Parallel Computing*, vol. 34, no. 4-5, pp. 217–230, 2008.

[14] R. A. Deville, I. A. Troxel, and A. D. George, "Performance monitoring for run-time management of reconfigurable devices," in *Proceedings of the 5th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '05)*, pp. 175–181, June 2005.

[15] J. M. Lancaster, J. D. Buhler, and R. D. Chamberlain, "Efficient runtime performance monitoring of FPGA-based applications," in *Proceedings of the IEEE International SOC Conference (SOCC '09)*, pp. 23–28, September 2009.

[16] J. M. Lancaster and R. D. Chamberlain, "Crossing timezones in the timetrial performance monitor," in *Proceedings of the Symposium on Application Accelerators in High Performance Computing*, 2010.

[17] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "Crash test: a fast high-fidelity FPGA-based resiliency analysis framework," in *Proceedings of the 26th IEEE International Conference on Computer Design (ICCD '08)*, pp. 363–370, October 2008.

[18] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: towards an open-source tool flow," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*, pp. 41–44, March 2011.

[19] V. Betz and J. Rose, "VPR: a new packing, placement and routing tool for FPGA research," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, W. Luk, P. Y. K. Cheung, and M. Glesner, Eds., vol. 1304 of *Lecture Notes in Computer Science*, pp. 213–222, Springer, 1997.

[20] J. Luu, I. Kuon, P. Jamieson et al., "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling," in *Proceedings of the 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '09)*, pp. 133–142, February 2009.

[21] J. Rose, J. Luu, C. W. Yu et al., "The VTR project: architecture and CAD for FPGAs from verilog to routing," in *Proceedings of the 20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 77–86, 2012.

[22] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapid prototyping tools for FPGA designs: RapidSmith," in *Proceedings of the 2010 International Conference on Field-Programmable Technology (FPT '10)*, pp. 353–356, December 2010.

[23] B. Huang, A. G. Schmidt, A. A. Mendon, and R. Sass, "Investigating resilient high performance reconfigurable computing with minimally-invasive system monitoring," in *Proceedings of the 4th International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA '10)*, pp. 1–8, November 2010.

[24] A. G. Schmidt, B. Huang, R. Sass, and M. French, "Checkpoint/restart and beyond: resilient high performance computing with FPGAs," in *Proceedings of the 19th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*, pp. 162–169, May 2011.

[25] A. G. Schmidt and R. Sass, "Improving design productivity with a hardware performance monitoring infrastructure," in *Proceedings of the 6th Annual International Conference on Reconfigurable Computing and FPGAs*, 2011.

[26] A. G. Schmidt, W. V. Kritikos, R. R. Sharma, and R. Sass, "AIREN: a novel integration of on-chip and off-chip FPGA networks," in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 271–274, April 2009.

[27] A. G. Schmidt, *Productively scaling hardware designs over increasing resources using a systematic design analysis approach [Ph.D. thesis]*, The University of North Carolina at Charlotte, 2011.

[28] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[29] S. Ganesh, *Implementation of the smith-waterman algorithm on fpgas [Ph.D. thesis]*, University of North Carolina at Charlotte, 2009.

[30] W. R. Pearson, "FASTA Sequence Comparison at the University of Virginia," July 2011, http://fasta.bioch.virginia.edu/fasta_www2/.

[31] J. C. Lagarias, "The 3x+1 problem and its generalizations," American Mathematical Monthly, pp. 3–23, 1985.