Challenges for Performance Analysis in High-Performance Reconfigurable Computing

Seth Koehler, John Curreri, and Alan D. George
NSF Center for High-Performance Reconfigurable Computing (CHREC)
HCS Research Lab, ECE Department, University of Florida
Email: {koehler, curreri, george}@chrec.org

Abstract

Reconfigurable computing (RC) applications employing both microprocessors and reconfigurable hardware (e.g., FPGAs) have the potential for large speedup when compared with traditional parallel applications. However, this potential is marred by the additional complexity of these dual-paradigm systems, making it difficult to achieve the desired performance. For parallel software applications (e.g., in MPI), performance analysis tools are well researched and widely available, aiding designers in understanding the dynamic behavior of their applications. Unfortunately, for RC applications, such concepts and tools are lacking, despite being of great importance due to the increased complexity of these applications. In this paper, we explore challenges faced in attaining automated techniques for instrumentation and low-overhead runtime measurement of diverse resources in RC systems. We present ideas for the integration of these techniques into existing performance analysis tools for conventional parallel systems with the goal of creating a unified analysis tool for RC applications. Results from a case study are provided using a prototype of this new tool.

1. Introduction

As parallel computing systems (e.g., multicore CPUs, clusters, etc.), multiprocessor systems-on-chip (MPSoC), and reconfigurable computing (RC) systems continue to mature, the amount of processing power available to applications continues to increase. RC applications employ both microprocessors and reconfigurable hardware such as FPGAs to handle computationally intensive problems. These RC applications have the potential to achieve orders-of-magnitude performance gains, using less power and hardware resources than conventional software applications [1], [2]. However, the behavior of an RC application can be particularly difficult to track and understand due to additional levels of parallelism and complex interactions between heterogeneous resources inherent in such systems [2]. High-performance RC applications are by definition a superset of traditional parallel computing applications, containing all the problems and complexity of these applications and more due to their use of both microprocessors and FPGAs. To handle this complexity, performance analysis[1] tools will be indispensable in application analysis and optimization, even more so than in parallel computing applications where such tools are already commonly used and highly valued.

Unfortunately, traditional performance analysis tools are only equipped to monitor application behavior from the CPU's perspective. Systems such as the Cray XD1 [3] or those employing Opteron-socket-compatible FPGA boards (e.g., XtremeData [4] or DRC [5]) are advancing the FPGA from slave to peer with CPUs, enabling the FPGA to independently interact with resources including main memory, other CPUs, and other FPGAs. Due to the increasingly significant role FPGAs play in RC applications, conventional tools have an increasingly incomplete view of application performance, yielding the need for hardware-aware performance analysis tools that can provide a complete view of RC application performance. To illustrate this need, Figure 1 shows the hierarchy of parallelism and myriad of interactions inside an RC computer, differentiating between communications that can be monitored (light arrows) and others that cannot (dark arrows) by traditional performance analysis tools. With FPGA communication paths to CPUs, other FPGAs, and various levels of memory, the amount of unmonitored communication is significant, hindering the designer's ability to understand and improve application performance.

[1] Throughout this paper, performance analysis refers to experimental performance analysis (i.e., studying application behavior on an actual system at runtime).

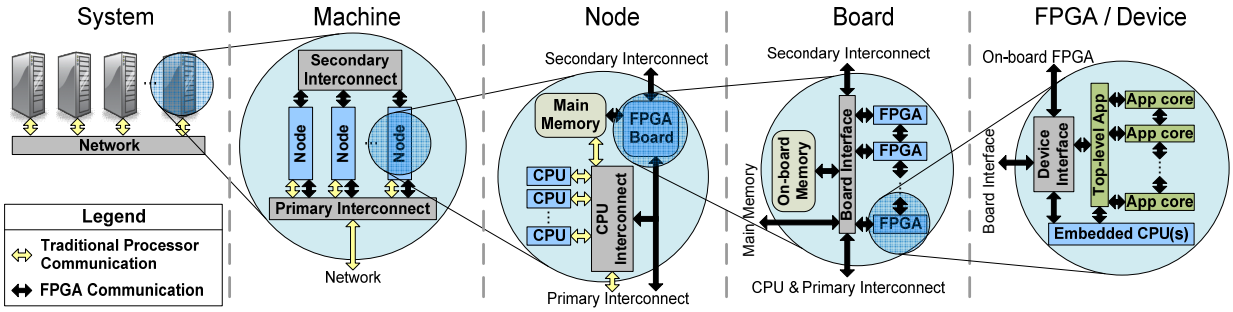

Figure 1: Potential communication bottlenecks (represented by arrows) in an RC application

In this paper, we explore the challenges faced in attaining low-overhead, automatable techniques for instrumentation and runtime measurement of RC applications. We also present concepts for the integration of these techniques into an existing parallel performance analysis tool, Parallel Performance Wizard (PPW), with the goal of creating a unified performance analysis tool for RC applications. Section 2 discusses background and prior work related to performance analysis in software and RC. Section 3 then explores the challenges and techniques for performance analysis of RC applications. Next, Section 4 provides a case study to demonstrate the overhead, benefits, and importance of performance analysis in RC applications using a prototype of our hardware measurement module. Finally, Section 5 concludes and gives directions for future work.

2. Background & related research

The goal of performance analysis is to understand a program's runtime behavior on a given system in order to locate and alleviate performance bottlenecks. For parallel-computing applications, Maloney's TAU framework [6] and Chung et al.'s recent study of performance analysis tools on the Blue Gene/L [7] provide a good introduction to the various challenges, techniques, and tools in performance analysis. Performance analysis can generally be decomposed into five stages (shown in Figure 2): instrument, measure, analyze, present, and optimize. *Instrumentation* is the process of gaining access to application (or system) data to be measured and stored at runtime. *Measurement* is then the act of recording and storing data while the application is executed on the target system. The resulting measured data is next *presented* via visualizations and *analyzed* by the application designer to locate potential performance bottlenecks. Optionally, some analysis may be automated, allowing visualizations to be augmented with the locations of potential bottlenecks. The designer then *optimizes* the application, attempting to remove performance bottlenecks discovered in the previous stages. These steps may then be repeated until desired performance is achieved or no further performance gains seem likely.

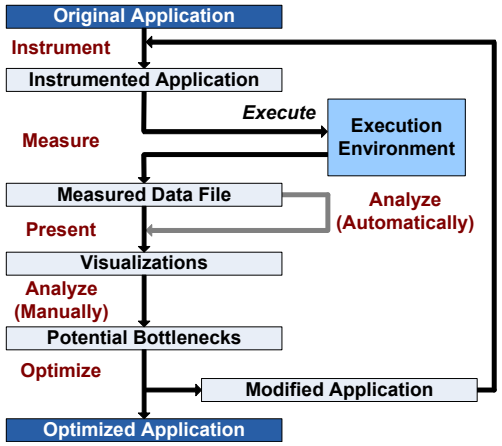

Figure 2: Stages of performance analysis

Performance analysis should not be confused with analytical models or simulation, which provide estimates of application performance that must eventually be verified against actual performance. Performance analysis is essential to capture actual application behavior on a given target system for the purposes of optimization. Similarly, although debug techniques can be useful to performance analysis, this overlap is limited by fundamental differences in their purpose. For example, debug techniques such as breakpointing and FPGA readback must stop the FPGA application in order to retrieve data [8]. Unfortunately, this technique effectively isolates the FPGA from the rest of the system, which typically cannot be paused. While isolation is encouraged in debugging, it is extremely problematic in performance analysis since component interaction in the system is a key factor. Tools such as Altera's SignalTap [9] and Xilinx's ChipScope [10] do allow an FPGA to run at or near full speed in a system (minimizing changes to application behavior), but are designed to monitor exact values at each cycle over a given period to ensure correctness, much like a logic analyzer. In contrast, performance analysis assumes correctness

and is instead concerned with timeliness of application progress, often allowing data to be summarized or ignored. By reducing the data recorded, fewer storage and communication resources are necessary to monitor an application, minimizing the distortion of the original application's behavior. In addition, SignalTap and ChipScope require separate connectors (e.g., JTAG) to acquire data, which are not readily accommodated or available for many systems.

While some preliminary work exists in RC performance analysis, this field is significantly less mature than its software counterpart. DeVille et al.'s paper investigates the use of distributed and centralized performance analysis probes in an FPGA, but is limited in scope to efficient measurement within a single FPGA [11]. Schulz et al.'s OWL framework paper proposes use of FPGAs for system-level performance analysis, monitoring system components such as cache lines, buses, etc. However, their work is directed at monitoring software behavior from hardware, rather than monitoring hardware itself [12].

3. Challenges for RC performance analysis

Significant challenges exist in each of the five stages of performance analysis. Instrumentation and measurement form the foundation of performance analysis that is built upon by analysis, presentation, and optimization. Thus, our focus in this paper will be instrumentation and measurement. We also briefly present challenges in presentation as well as concepts toward a unified performance analysis tool. As automating analysis and optimization are still open research areas (and often performed manually), these topics are beyond the scope of this paper. Within instrumentation and measurement, the key goals of performance analysis tools are the following (adapted from [6]):

1. Perturb the original application's behavior as little as possible (minimize impact).
2. Record sufficient detail & structure to accurately reconstruct application behavior (maximize fidelity).
3. Allow flexibility to monitor diverse applications and systems (maximize adaptability and portability).
4. Require as little effort and trouble for the designer as possible (minimize inconvenience).

Goals 1 and 2 are opposed to one another, as are Goals 3 and 4. Thus, the challenges faced generally stem from attempting to reach a compromise.

3.1. Challenges for hardware instrumentation

Instrumenting a hardware design involves gaining entry points to signals (i.e., wires) in the application. A logic analyzer exemplifies this process with logic probes connected to external pins that are in turn connected to values of interest in the application. By taking advantage of the reconfigurability of an FPGA, we can use the built-in routing resources to temporarily access application data, gaining the necessary entry points for measurement. Instrumentation involves choosing *what data to instrument*, choosing *the level(s) of instrumentation* (e.g., source, binary, etc.), and finally *modifying the application* at the chosen level to gain access to the selected data. These issues are discussed in the following subsections.

3.1.1. What to instrument

Instrumenting an application begins with a selective process that determines what data to record and what to ignore. The data chosen should reflect application behavior as closely as possible while simultaneously minimizing perturbation of that behavior (Goals 1 and 2). While application knowledge is useful in making these selections, it is desirable to automate this time-consuming process when possible (Goal 4). In addition, automation is essential if hardware was generated from a high-level language (HLL), as the designer will not be aware of the implementation details. Software performance analysis has demonstrated that such automation is possible by using knowledge of what constitutes a common performance bottleneck to guide instrumentation. Thus, one key challenge in FPGA instrumentation is determining common performance bottlenecks in a typical FPGA design.

Applications consist of communication and computation, both of which must be monitored to understand application behavior. Software performance analysis typically monitors specific constructs that invoke communication explicitly or implicitly through synchronization primitives such as barriers, and locks. Computation is typically monitored by timing function calls or other control structures such as loops, which in hardware is similar to the control of subcomponents, such as through a state machine, pipeline, or loop counter. Thus, these hardware communication and control constructs provide a starting point for studying common performance bottlenecks in an FPGA.

In an FPGA, communication includes off-board (e.g., to another FPGA, CPU, main memory, etc.), on-board (e.g., to on-board DDR memory or other FPGAs connected to the FPGA on-board), or on-chip (between components inside the FPGA device) communication. Communication off-board and on-board is widely known to be problematic in FPGA-based system designs, but on-chip communication can be a significant

bottleneck as well, especially if some form of routing network or data distribution is implemented in the design (a common technique used in applications containing multiple cores to exploit parallelism). Instrumenting on-chip communication between components such as frequency of communication or bytes transferred can help the designer to better understand how the component is used. However, due to the large amount of parallelism possible in an FPGA, monitoring all on-chip communication can incur significant overhead.

Control can become a bottleneck when too many cycles are used for setup, completion, or bookkeeping tasks. However, the primary reason for instrumenting control is to gain insight into the application's behavior, helping the designer to locate other bottlenecks. As an example, if a state machine contains a state that waits for data from an FFT core, recording the number of cycles spent in this wait-state can determine whether the FFT core is a bottleneck in the application. This information is comparable to a software performance analysis tool monitoring the amount of time an FFT subroutine required.

It is important to note that instrumentation should be restricted to clocked elements in hardware. Synthesis and place-and-route tools already optimize delays associated with unclocked (combinatorial) signals; these delays can be analyzed via timing analysis, simulation, or debugging tools. However, even in designs that are primarily combinatorial, there is inevitably some clocked portion of the design that handles control or communication (and often multiple levels of communication and control), demonstrating the wide applicability of these areas across designs (Goal 3).

Thus, communication and control are reasonable points to instrument initially. However, application knowledge can often give further insight into what should be instrumented. Certain control and communication may be unnecessary to monitor in a specific application; performance may be better understood by monitoring a specific input value to a component. This application knowledge is extremely difficult to automate, and thus determining what to instrument remains a significant challenge in RC performance analysis.

3.1.2. Levels of instrumentation

Before reaching the challenge of modifying an application for analysis, the level at which instrumentation will occur must be selected. The hardware portion of an RC application can be instrumented at any level between source code (HDL) and bit file (binary loaded directly onto the FPGA). While it is also possible to use system-level instrumentation (e.g., OWL [12] discussed in Section 2), this approach lacks portability due to the requirement of dedicated hardware to monitor system components such as cache lines, buses, etc. In addition, data unrelated to the application is also captured, such as the behavior of the operating system and other running applications, making system-level instrumentation less suitable for performance analysis of a *specific* application; thus, system-level instrumentation is not considered further here.

Graham et al. provides an excellent look at the various levels and tradeoffs of application-level instrumentation inside an FPGA [13]. They indicate that while instrumenting at intermediate levels between source code and binary offers some advantages (e.g., modifying clean abstract syntax trees as opposed to source code or binaries), these advantages are not significant enough to counterbalance the poor documentation and difficulty of accessing these levels (some levels occur only in memory during synthesis). Thus, the levels of instrumentation are in practice polarized into source-level and binary-level instrumentation.

Source instrumentation is attractive since it is easier to implement, fairly portable across devices, flexible with respect to which signals can be monitored, and often minimizes the change in area and speed of the instrumented design due to optimization of the design after instrumentation. Source instrumentation also offers the possibility of source correlation, allowing behavior to be linked back to source code, although the meaning and implementation of this correlation is less clear in a language modeling hardware than in one modeling execution of instructions on a CPU.

By contrast, binary-level instrumentation is attractive because it requires less time to instrument a design (e.g., minutes instead of hours as it occurs after place-and-route), is portable across languages for a specific device, and perturbs the design layout less, again since it is mostly added after the design has been optimized and implemented. Unfortunately, binary instrumentation loses some flexibility since synthesis and implementation may have significantly transformed or eliminated some data during optimization or made some data inaccessible via the FPGA routing fabric. Links between behavior and source are also lost. Applying instrumentation at both levels is also possible, allowing the designer to select the appropriate compromise for each instrumented datum. Table 1 provides a summary of this comparison.

3.1.3. Modifying the application

Once an instrumentation level has been selected, the application must be modified to allow access to

Table 1: Source vs. binary instrumentation

	Source-level	Binary-level
Difficulty	Text parsing	Bit file signal routing
Design Perturbation	Low change in area & speed	Low change in on- chip physical layout
Time to Instrument	Long (hours)	Short (minutes)
Portability	Good across devices	Good across languages
Flexibility	Access to all signals	Some data inaccessible
Source Correlation	Possible	Generally not possible

whatever data has been chosen for instrumentation. While both source and binary instrumentation can draw heavily from similar techniques in both software and FPGA debugging, automatically inserting instrumentation based upon the decision to instrument control and communication (discussed in Section 3.1.1) still poses a challenge for FPGA instrumentation. While software performance analysis often scans source code for specific API calls that are harbingers of communication (e.g., an MPI_Send call in an MPI program), FPGA communication and control are more difficult to detect.

Source-level instrumentation for hardware can employ a preprocessor to scan application code and insert lines to extract the desired data at runtime (e.g., in VHDL, the component interface can be modified to allow access to performance data). The challenge here lies in the expressiveness of the given language; the preprocessor must be able to cope with the various ways in which a designer may structure or express the behavior of their application. For example, the use of an enumerated type in VHDL along with a clocked case statement using it would usually suggest a state machine. However, the same structure could be represented with constants and a complicated if-then-else structure.

Binary-level instrumentation suffers similar difficulties. Now control and communication must be detected from a fully optimized and implemented design. While escaping the problem of the source language's expressiveness, the hierarchy and structure behind much of the application has been flattened and reformed during synthesis and implementation. Given a set of physical lookup tables (LUTs) in the FPGA to monitor, binary-level instrumentation can be performed by synthesizing and implementing the original design as usual, save the need to reserve space and connection points for the measurement device. After synthesis and implementation, tools such as Xilinx's JBits SDK [14] can be used to place the measurement framework in the device and route signals to it from the application.

3.2. Challenges for hardware measurement

Measurement is concerned with how to record and store data selected during instrumentation. An integral challenge of this process is to record enough data to understand application behavior while at the same time minimizing perturbation caused by recording (Goals 1 and 2). Due to limited resources and a lack of resource virtualization on an FPGA, resource sharing between the application and measurement framework presents a unique challenge for RC performance analysis.

3.2.1. Recording and storing performance data

To balance fidelity and overhead, software performance analysis employs techniques such as tracing (recording the event time and associated data) and profiling (recording summary statistics and trends, not when specific events occurred). These methods can be triggered to record information under specific conditions (event-based) or periodically (sampling). The efficacy of one technique over another is dependent upon what behavior needs to be observed in the application.

Tracing is the methodology of recording data and the current time (based on a device clock), allowing duration and relative ordering of events to be analyzed. To maintain event ordering between devices, clock offset and drift must be periodically monitored on all CPUs and FPGAs; methods such as those in [15] estimate round-trip delay, enabling clock drift to be corrected postmortem. While closely related to hardware debugging, tracing in performance analysis must be sustainable for an indefinite period of time in order to capture application behavior (debug techniques often record until memory is exhausted). To reduce the amount of data recorded, event-based tracing records data only under specified conditions, whereas sample-based tracing records data periodically. Based on the event conditions or sampling frequency, a different compromise is reached between fidelity and perturbation.

Profiling differs from tracing in that no specific event timing is stored. Rather, summary statistics of the data are maintained, usually with simple counters that are extremely fast and fairly small. Profiling sacrifices some of the fidelity of tracing for less perturbation of the design. Profile counters can provide statistics such as totals, maximums, minimums, averages, and even variance and standard deviation, although at the cost of additional hardware (and possible performance degradation). Like tracing, profile counters can be updated based upon an event or periodically (sampling).

One significant difference between software and hardware performance analysis with respect to profiling and tracing is parallelism. While software measurement

requires additional instructions to profile or trace the application that generally degrade performance, profile counters and trace buffers can work independently of the application and each other in hardware. Thus, hardware performance analysis can incur no performance degradation if sufficient resources are available and the maximum design frequency is unaffected. In addition, it is possible to monitor extremely fine-grain events, even those occurring every cycle. Another significant difference involves resource availability. While profiling typically requires far less memory than tracing, profile counters that must be accessed simultaneously will likely be placed in logic cells on an FPGA rather than block RAM. Unfortunately, logic cells are scarce in comparison to block RAM, the latter of which is suitable for tracing. For example, 512 36-bit profile counters require 16.7% of logic cells in a Xilinx Virtex-4 LX100 device, and yet could be stored in a single block RAM (representing only 0.4% of block RAM on the same device) [16]. Unfortunately, block RAM can be limiting as well when tracing (e.g., the LX100 contains only 540KB of block RAM) [16]. In contrast, software performance analysis has hundreds of megabytes of memory or more to store profile and trace data.

In hardware, the tradeoffs between tracing and profiling provide a significant challenge to automating the selection of the measurement type to use for a specific signal in an FPGA. While the designer may recognize data that would be problematic for tracing or poorly represented by profile counters, this knowledge is rarely explicit in the application code or bit file. Worse, once a selection has been made, the measurement framework necessary to monitor this selection may not fit in the remaining logic or memory on the FPGA, or the measurement framework may cause significant degradation of the maximum frequency at which the application can run. Thus, finding a balance between perturbation and fidelity may require significant knowledge of both the application and tradeoffs in measurement strategies.

3.2.2. Managing shared resources

One of the greatest challenges in RC performance analysis is the management of shared resources that were once exclusively controlled by the application. Although the sharing of on-chip resources is important, this sharing is handled by the synthesis and implementation tool, and thus is of less concern. Off-chip communication sharing is more difficult due to the limited number of communication paths. While recording data would ideally require no off-chip communication, limited FPGA memory necessitates periodic transfers of performance data to a larger storage medium at runtime.

Software performance analysis tools can share memory, communication, and processor time with the application through operating system and hardware virtualization (processes, virtual memory, sockets, etc.). FPGAs have none of this infrastructure, requiring performance analysis to handle these complexities manually. To share the FPGA interconnect, performance analysis frameworks must ensure performance and application data can be distinguished so that each is delivered to the correct location, usually by allocating memory and address space for performance data to use exclusively. Arbitration between the application and performance analysis hardware is also necessary to ensure that only one can access the interconnect at a time.

One added complexity is that the communication architecture may only allow the CPU to initiate a data transfer from the FPGA to main memory. This scenario can be handled by instrumenting the application software to periodically poll the performance analysis hardware for data, either directly between other application tasks or via a separate process or thread. If supported, interrupts can be used to have the CPU initiate a transfer, although interrupts are often scarce and thus may need to be shared if used by the application. When processes, threads, or interrupts are used, the application and performance monitoring software must use locks to guarantee they do not access the FPGA simultaneously.

It is important to note that while measurement determines the need for communication sharing, instrumentation is affected as well, since it must now be aware of these communication schemes and seamlessly integrate with them. Communication schemes such as memory maps or network packets are used with a variety of interconnects. Combined with diverse APIs for FPGAs, virtualizing communication is a major challenge for RC performance analysis.

Figure 3 illustrates the possible changes to an RC application during instrumentation in order to support measurement. In hardware, component interfaces are modified to allow performance data to be accessed, and a new top-level file is added to control communication access between the original application and the performance analysis module. In addition, the communication path between the CPU and FPGA is now shared and controlled by the new top-level file. In software, a data transfer module is added to support CPU-initiated performance transfers. Modifications to the HLL source are also necessary to retrieve performance data.

3.3. Challenges in performance presentation

Current trace-based displays such as Jumpshot [17] show communication and computation for parallel com-

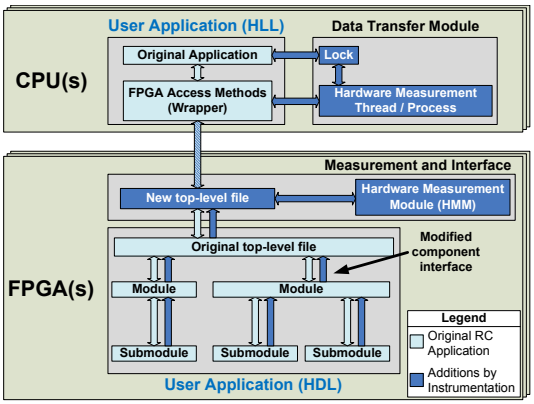

Figure 3: Additions made by source-level instrumentation of an RC application

puting applications. These displays can be extended to include FPGAs as additional processing elements for RC applications. A visualization (mockup) example is shown in Figure 4. In this example, the CPUs (nodes 0-7) are actively completing their work and receiving data from the FPGAs (nodes 8-15). Nodes 3 (CPU) and 11 (FPGA) are idle while most processors finish the current iteration somewhere in the middle of the diagram. Nodes 4 and 12 are lagging, completing towards the end of the interval and finally allowing global synchronization of all nodes before a new iteration begins.

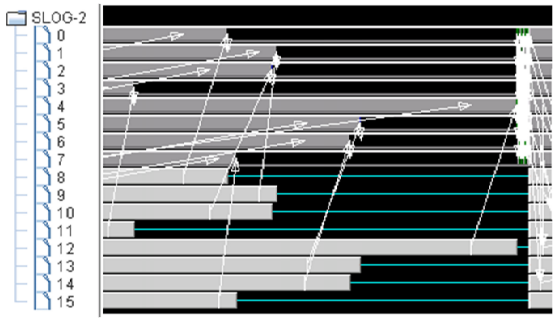

Figure 4: Mockup Jumpshot visualization of an RC application with 8 CPUs and 8 FPGAs

Unfortunately, trace-based displays scale poorly with system size. If the performance data included hundreds of CPUs and FPGAs communicating with each other, this diagram may become ineffective. Worse yet, the classification of an FPGA as a single processing element above does not accurately account for the levels of parallelism inside the FPGA. Treating the FPGA as a multicore processor may be inadequate as well if the FPGA design uses different cores in the same device or possibly in a hierarchy.

Hierarchical views of clusters such as Ganglia [18] have been developed and could be useful in large-scale RC systems for displaying heterogeneous devices and behavior. Finding the right balance between abstraction and detail in large-scale visualizations remains a significant challenge in presentation for parallel computing as well as RC.

3.4. Unified performance analysis tool

To create a holistic view of an RC application's behavior, a unified software/hardware tool is essential. Separate tools will give a disjointed view of the system, requiring significant effort to stitch the two views back together. In addition, each tool must make decisions about instrumentation and measurement without any knowledge of what is being monitored by the other. A unified tool could potentially take advantage of strategically choosing where to monitor a specific event, such as software, hardware, or both, based upon factors such as efficiency, difficulty in accessing information, and accuracy of that information. Also, some instrumentation and measurement techniques require complimentary modifications to software and hardware (e.g., modifying a memory map to allow CPU-initiated transfers of performance data).

We use the Parallel Performance Wizard (PPW) [19] as a specific software performance analysis tool to discuss integration here, although these concepts apply to other tools as well. PPW supports performance analysis for Partitioned Global Address Space (PGAS) programming languages such as UPC and SHMEM (with addition of MPI support planned) via the Global Address Space Performance (GASP) interface [20], which is currently implemented by compilers such as Berkeley's UPC and gcc-upc. Based on a specific language, many constructs such as synchronization primitives will warrant monitoring, which the compiler instruments by using event callback functions (user-definable events are also possible). These events can then be received by any tool supporting the GASP interface, where the tool can choose to profile, trace, or ignore these events.

To track FPGA activity from software, the GASP interface could be extended with generic events such as FPGA reset, configure, send, and receive. Upon receiving an FPGA event, the performance tool could store information such as bytes transferred, time taken to configure the FPGA, or transfer latency, providing a detailed view of FPGA communication from software. However, automatically adding these extended GASP functions around FPGA communication is difficult; FPGA communication can appear in a variety of ways in software, including vendor APIs, pointers, and I/O calls. Ideally, a standard API for FPGA access could make detection of FPGA calls trivial. In the absence of such a standard, performance analysis tools must detect each vendor's FPGA access methods.

4. Case study

To demonstrate the overhead, benefits, and importance of RC performance analysis techniques, we present a case study using a prototype version of our hardware measurement module (HMM). The HMM allows for profiling and tracing of data via event-based triggering or sampling and uses CPU polling via a separate thread to periodically transfer performance data off-chip since this is universally supported. Due to portability, flexibility, and difficulty concerns, instrumentation in our framework is performed at the source level. While instrumentation is currently performed manually, the HMM allows quick customization of (and easy access to) profile counters and trace buffers, eliminating the time-consuming and error-prone process of manually measuring performance. Figure 5 illustrates the design of our prototype HMM.

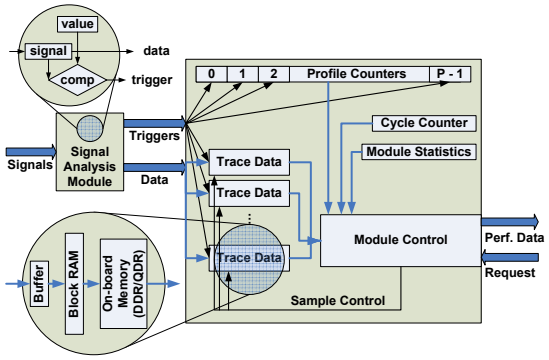

Figure 5: Hardware Measurement Module

For our case study, we executed the N-Queens benchmark application on two RC systems. The first RC system, the Cray XD1, consists of six nodes, each containing two Opteron 250 CPUs and a Xilinx Virtex-2 Pro 50 FPGA connected via a high-speed interconnect (3.2GB/s ideal peak) [3]. The second RC system is a 16-node Gigabit Ethernet cluster, each node containing a 3.2GHz Intel Xeon EM64T processor and a Nallatech H101-PCIXM application accelerator [21] employing a Xilinx Virtex-4 LX100 user FPGA and connected via a PCI-X bus (1GB/s ideal peak). The N-Queens application was implemented using UPC (software) and VHDL (hardware). Compilation for the Cray XD1 was performed using Synplicity's Synplify Pro 8.6.2, Xilinx's ISE 7.1.04i, and Berkeley UPC 2.4.0. Compilation for the cluster was performed using Nallatech's Dimetalk 3.1.5, Xilinx's ISE 9.1.03i, and Berkeley UPC 2.4.0.

The N-Queens problem asks for the number of distinct ways that N queens can be placed onto an NxN chessboard such that no two queens can attack each other [22]. As only one queen can be in each column, a simple algorithm was employed to check all possible positions via a back-tracking, depth-first search. Parallelism was exploited by assigning two queens within the first two columns; each core then receives a partial-board and generates all possible solutions by moving queens in the remaining $N - 2$ columns, returning the number of solutions to software. The program was executed on both RC systems using a board size of 16×16. The N-Queens application was first executed without hardware instrumentation to acquire baseline timing, and then with instrumentation to collect measured data. The HMM was configured to include 16 profile counters in each FPGA (six for monitoring application communication, nine for monitoring an N-Queens core state machine, and one to monitor the number of solutions found by that core) and one 2KB trace buffer to monitor the exact cycle in which any core in the application completed. The UPC and VHDL code were modified to allow performance data to be transfered at runtime, with the CPU polling the FPGA once per millisecond for performance data (the application independently polled the device once per 100 milliseconds). Table 2 provides the overhead incurred by adding instrumentation to the N-Queens cores and periodically measuring profile counters and trace data from the N-Queens application at runtime. From this data, a maximum overhead bandwidth of 33.3KB/s was observed, which is negligible when compared to the interconnect bandwidth. Less than 7% of the FPGA's resources and 2% of the block RAM were needed to monitor the application. Frequency degradation ranged from 1% on the XD1 to no degradation on the LX100s in the cluster.

Table 2: Performance Analysis Overhead

	XD1	XD1 (instr.)
Slices (23616 total)	9041 (38.3%)	9901 (41.9%) +860 (3.7%)
Block RAM (232 total)	11 (4.7%)	15 (6.5%) +4 (1.7%)
Frequency (MHz)	124	123 -1 (-0.8%)
Communication (KB/s)	0.08	33.29 +33.21
	Cluster	**Cluster (instr.)**
Slices (49152 total)	23086 (47%)	26218 (53%) +3132 (6.4%)
Block RAMs (240 total)	21 (8.8%)	22 (9.2%) +1 (0.4%)
Frequency (MHz)	101	101 0 (0%)
Communication (KB/s)	0.04	29.86 +29.82

The number of cycles spent in each state of an N-Queens core state machine was monitored in order to understand a core's behavior at runtime. While not ac-

cessible from a software performance analysis tool, this information is easily obtained by using as many profile counters as there are states, with each counter incrementing when that state occurs. From this data, the percentage of cycles spent in each state was calculated and is shown in Figure 6. More than a third of the total time is spent determining whether any queens can attack each other. While this state would normally be targeted for optimization, it was already heavily optimized, leaving little room for improvement. However, Figure 6 also shows that the *Reset Attack Checker* state consumes 12% of the total state machine cycles, which is surprising given the relatively small job that this state performs. Thus, a relatively simple modification was made to combine the *Reset Attack Checker* state, as well as the *Finished* and *Reset Queen Row*, with the remaining states, giving an ideal speedup of 16.3% versus the non-optimized version, based upon removing these states from the graph. This speedup is ideal as the optimization only applies to a portion of the application (neglecting setup and communication times). While frequency degradation can also affect speedup, a negligible drop in core clock frequency of the optimized version was observed. The optimized N-Queens core was then measured on the target systems, giving an average speedup of 10.5%. This performance gain was greatly facilitated by the use of hardware performance analysis, removing guesswork from understanding the application's behavior and aiding in the detection of performance bottlenecks.

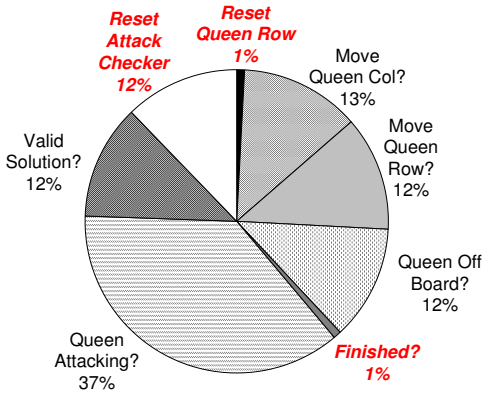

Figure 6: Distribution of cycles spent in core state machines of N-Queens

The trace buffer was used to monitor the cycle in which any core in the device completed in order to understand the penalty of the application's static scheduling, which requires all cores in the device to complete before receiving further work. Tracing data (ignoring trivial completions of invalid starting boards) revealed that the first core to complete was idle 25% of the time, waiting for the last core to complete; on average cores were idle 10% of the time. Thus, a dynamic scheduling algorithm could ideally improve speedup by 11%.

Figure 7 shows the speedup of the parallel software and both the initial and optimized hardware versions of N-Queens over the baseline sequential C version. The 8-node software version was able to achieve a speedup of 7.9 over the sequential baseline. The cluster executing the optimized hardware on 8 FPGAs achieved a speedup of 37.1 over the baseline.

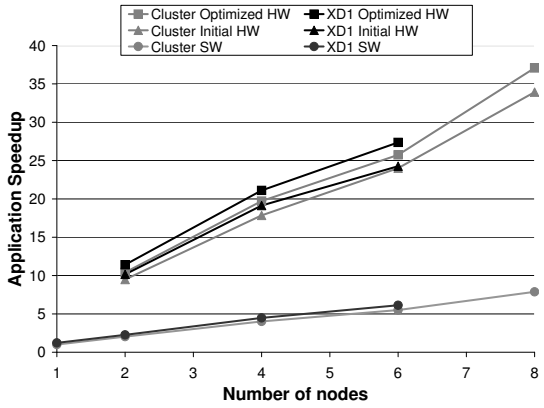

Figure 7: Speedup of N-Queens Application

5. Conclusions

In this paper we have explored various challenges faced in RC performance analysis. While we discussed some challenges that are shared by software performance analysis, many of these challenges are more difficult in or unique to RC. Challenges such as resource sharing, automation of instrumentation and measurement, as well as compromises between accurate and precise measurement all need to be addressed for performance analysis to be successful. Furthermore, as systems continue to increase in size, the difficulty in presenting meaningful visualizations continues to increase. We proposed that, due to the complexity inherent in large-scale RC systems and applications, unification of software and hardware performance analysis into a single tool is crucial to efficiently record and understand application behavior at runtime.

To demonstrate the overhead and benefits of these techniques, results from an N-Queens case study were provided. Using N-Queens on two RC platforms, we demonstrated that our prototype hardware measurement module (HMM) incurred little overhead. Measuring application behavior using profile counters and tracing cost no more than 6.4% of the logic resources in a medium-sized FPGA, 1.7% of the block RAM, 1% in

frequency degradation, and 33KB/s in bandwidth when polled once per millisecond. From the performance data returned, including statistics on time spent in the main N-Queens state machine, the behavior of the application was readily understood, resulting in a 10.5% speedup with minimal modifications.

Directions for future work include studying more advanced methods of signal analysis, measurement approaches (e.g., FPGA-initiated transfers), and other techniques to minimize overhead and improve the fidelity of measured data. In addition, further study of automated instrumentation techniques (especially with high-level languages) as well as development of large-scale visualizations will be critical in order for performance analysis of RC applications to achieve wide use. Recording data from more complex designs, such as designs with multiple clock domains or an embedded processor, are also important. Finally, automation of analysis and optimization are open areas of research that could enable more widespread and effective use of performance analysis without intricate design knowledge.

Acknowledgments

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by Xilinx, Cray, Nallatech, Aldec, and Synplicity.

References

[1] M. C. Smith, J. S. Vetter, and X. Liang. "Accelerating scientific applications with the SRC-6 reconfigurable computer: methodologies and analysis", *Proc. of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2005, p. 157b.

[2] J. L. Tripp, A. A. Hanson, M. Gokhale, and H. Mortveit. "Partitioning hardware and software for reconfigurable supercomputing applications: a case study", *Proc. of 2005 ACM/IEEE Conference on Supercomputing (SC)*, Nov. 2005, p. 27.

[3] Cray, "Cray XD1 datasheet", 2005, http://www.cray.com/downloads/Cray_XD1_Datasheet.pdf.

[4] XDI, "XD1000™FPGA Coprocessor Module for Socket 940", http://www.xtremedatainc.com/.

[5] DRC, "RPU100-L60 DRC Reconfigurable Processor Unit", 2006, http://www.drccomputer.com/.

[6] S. S. Shende and A. D. Malony. "The Tau Parallel Performance System", *International Journal of High-Performance Computing Applications (HPCA)*, May 2006, 20(2):297–311.

[7] I. Chung, R. E. Walkup, H. Wen, and H. Yu. "MPI performance analysis tools on Blue Gene/L", *Proc. of 2006 ACM/IEEE Conference on Supercomputing (SC)*, Nov. 2006, p. 123.

[8] K. Camera, H. K. So, and R. W. Brodersen. "An integrated debugging environment for reprogrammable hardware systems", *Proc. of 6th International Symposium on Automated Analysis-Driven Debugging (AADEBUG)*, Sep. 2005, pp. 111–116.

[9] Altera, "Design Debugging Using the SignalTap II Embedded Logic Analyzer", Mar. 2007, http://www.altera.com/literature/hb/qts/qts_qii53009.pdf.

[10] Xilinx, "Xilinx ChipScope Pro Software and User Guide, v. 9.1.01", Jan. 2007, http://www.xilinx.com/ise/verification/chipscope_pro_sw_cores_9_1i_ug029.pdf.

[11] R. DeVille, I. Troxel, and A. George. "Performance monitoring for run-time management of reconfigurable devices", *Proc. of International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, June 2005, pp. 175–181.

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

[13] P. Graham, B. Nelson, and B. Hutchings. "Instrumenting Bitstreams for Debugging FPGA circuits", *Proc. of 9th annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2001, pp. 41–50.

[14] S. A. Guccione, D. Levi, and P. Sundararajan. "JBits: A Java-based interface for reconfigurable computing", *Proc. of 2nd Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, Sep. 1999, p. 27.

[15] F. Cristian. "A probabilistic approach to distributed clock synchronization", *Proc. of 9th International Conference on Distributed Computing Systems (ICDCS)*, June 1989, pp. 288–296.

[16] Xilinx, "Virtex-4 Family Overview", Jan. 2007, http://direct.xilinx.com/bvdocs/publications/ds112.pdf.

[17] C. E. Wu et al. "From trace generation to visualization: a performance framework for distributed parallel systems", *Proc. of 2000 ACM/IEEE Conference on Supercomputing (Cdrom) (SC)*, Nov. 2000, p. 50.

[18] M. L. Massie, B. N. Chun, and D. E. Culler. "The Ganglia Distributed Monitoring System: Design, Implementation and Experience". Technical report, University of California, Berkeley, Feb. 2003.

[19] A. Leko and M. B. III. *Parallel Performance Wizard User Manual*, 2007, http://ppw.hcs.ufl.edu/docs/pdf/manual.pdf.

[20] A. Leko, D. Bonachea, H. Su, H. Sherburne, B. Golden, and A. George. "GASP! A Standardized Performance Analysis Tool Interface for Global Address Space Programming Models", *Proc. of Workshop on Applied Parallel Computing (PARA)*, June 2006, http://www.hcs.ufl.edu/pubs/PARA06.pdf.

[21] Nallatech, "H100 Series FPGA Application Accelerators", Apr. 2007, http://www.nallatech.com/mediaLibrary/images/english/5595.pdf.

[22] C. Erbas, S. Sarkeshik, and M. M. Tanik. "Different Perspectives of the N-Queens problem", *Proc. of 1992 ACM Annual Conference on Communications*, Mar. 1992, pp. 99–108.