Performance Analysis Challenges and Framework for High-Performance Reconfigurable Computing

Seth Koehler, John Curreri, and Alan D. George

NSF Center for High-Performance Reconfigurable Computing (CHREC)
ECE Department, University of Florida
Email: {koehler, curreri, george}@chrec.org

Abstract

Reconfigurable computing (RC) applications employing both microprocessors and FPGAs have potential for large speedup when compared with traditional (software) parallel applications. However, this potential is marred by the additional complexity of these dual-paradigm systems, making it difficult to identify performance bottlenecks and achieve desired performance. Performance analysis concepts and tools are well researched and widely available for traditional parallel applications but are lacking in RC, despite being of great importance due to the applications' increased complexity. In this paper we explore challenges and present new techniques in automated instrumentation, runtime measurement, and visualization of RC application behavior. We also present ideas for integration with conventional performance analysis tools to create a unified tool for RC applications as well as our initial framework for FPGA instrumentation and measurement. Results from a case study are provided using a prototype of this new tool.

Keywords: Reconfigurable computing, FPGA, performance analysis, instrumentation, measurement, visualization

1 Introduction

As parallel computing systems (e.g., multicore CPUs, clusters, etc.), multiprocessor system-on-chips (MPSoCs), 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 observe 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

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

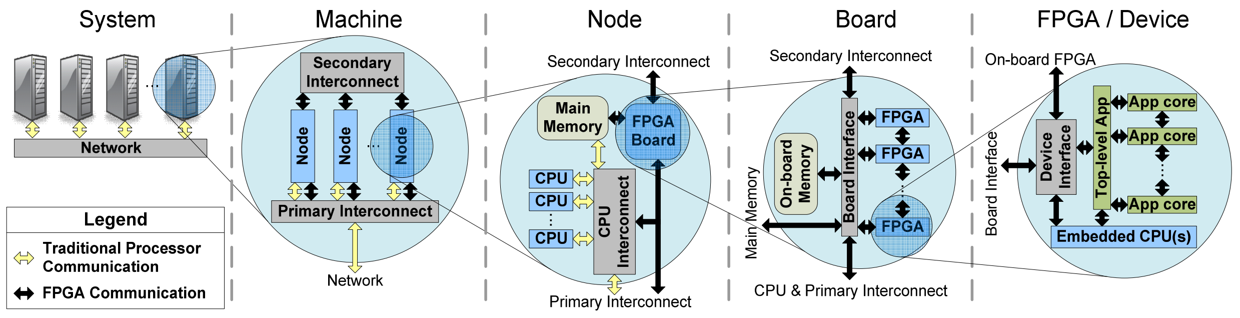

Figure 1: Potential communication bottlenecks (represented by arrows) in an RC application

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 system, 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.

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. The remainder of this paper is organized as follows. 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 an overview of our performance analysis framework. Section 5 provides results from a case study to demonstrate the benefits and importance of performance analysis in RC applications using a prototype of our hardware measurement module. Finally, Section 6 concludes this paper 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): instrumentation, measurement, analysis, presentation, and optimization. *Instrumentation* is the process of enabling 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.

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 tech-

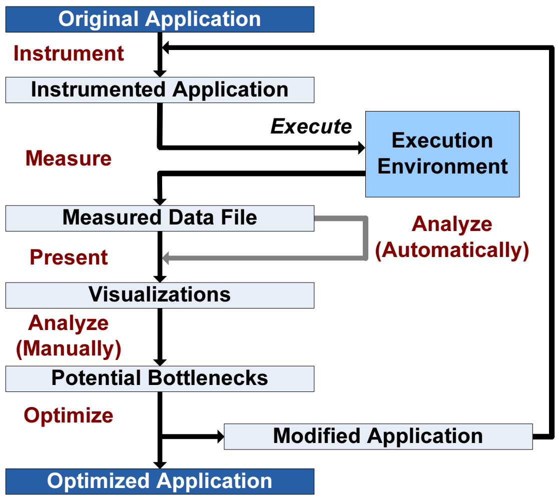

Figure 2: Stages of performance analysis

nique 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 the 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 thus these stages are 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 from 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. The following two goals for presentation provide some context for the four goals above, as presentation uses the measured data to reconstruct application behavior:

5. Display only what is necessary to capture application behavior and bottlenecks (be concise).
6. Format data to allow rapid understanding of application behavior (be intuitive).

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, acquiring 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). 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 are similar to the mechanisms used to control subcomponents in hardware (e.g., state machines, pipelines, loop counters, etc). 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 the same board), or on-chip (between components inside the FPGA device) communication. Communication off-board and on-board is widely known to be a potential bottleneck in FPGA-based system designs. Nonetheless, 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 (e.g., to observe 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 that obtained by a software performance analysis tool monitoring the amount of time an FFT subroutine required.

It is important to note that instrumentation should generally 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. 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 (e.g., VHDL) 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. System-level instrumentation is thus not considered further here.

Graham et al. provides an excellent look at the various levels and associated 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 exist only in memory during synthesis and implementation). 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, is fairly portable across devices, is 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.

In 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 for FPGAs is very difficult, much more so than instrumenting assembly code in a software binary. In addition, 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 code are also lost.

It is also possible to apply instrumentation at both levels, allowing the designer to select the appropriate compromise for each instrumented datum. Table 1 provides a summary of the comparison between source and binary instrumentation.

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

3.1.3 Modifying the application

Once an instrumentation level has been selected, the application must be modified to allow access to whatever data has been chosen for instrumentation. While both source and binary instrumentation can draw heavily from similar techniques in software and FPGA debugging, automatic instrumentation based upon the decision to instrument control and communication (discussed in Section 3.1.1) still poses a challenge for FPGA instrumentation. For example, software instrumentation might involve scanning source code for specific API calls that are harbingers of communication (e.g., an MPI_Send call in an MPI program), whereas FPGA communication and control are not as easy to detect in either the High-Level Language (HLL) or Hardware Description Language (HDL) portion of the application. Due to a lack of standards, FPGA communication in HLLs is currently proprietary, appearing in forms such as vendor function calls, software pointers to FPGA memories, and I/O calls. In an HDL, communication with a CPU, other FPGA, or on-board memory can either be proprietary or conform to one of many standards depending on the actual hardware available (e.g., PCIx/e, HyperTransport, RapidIO, SDRAM, SRAM). On-chip communication, while better defined by the component inputs and outputs, still poses similar difficulties. For example, components in an FPGA may make use of a read enable signal which can be named arbitrarily, or the component may read data only when a complex set of conditions are true.

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 that type 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, except for 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 individual event times 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) for individual events, allowing the duration and relative ordering of these 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). As with tracing, profile counters can be updated based upon an event or by periodically checking some condition (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 design's maximum clock frequency is unaffected. In addition, it is possible to monitor extremely fine-grained events, even those occurring every cycle. Another significant difference involves limited memory availability in an FPGA. Software performance analysis typically has hundreds of megabytes of memory or more to store profile and trace data, while an FPGA such as Xilinx's Virtex-4 LX100 device contains only 540KB of block RAM and 13.5KB of memory in logic cells [16]. While profiling typically requires far less memory than tracing, profile counters that must be accessed simultaneously are likely to be placed in logic cells (a set of profile counters can be placed in block RAM if only one counter in the set will be updated per cycle). As an example, 512 36-bit profile counters require a minimum of 16.7% of logic cells in Xilinx's Virtex-4 LX100 device [16], and yet could be stored in a single block RAM (representing only 0.4% of block RAM on the same device). In contrast, tracing is well suited for block RAM and thus can make use of additional storage. Unfortunately, trace data can easily exhaust block RAM resources, yielding a shortage of resources for both profiling and tracing.

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 than off-chip memory and communication sharing, which must be managed manually. While recording performance data would ideally require no off-chip communication (or possibly use a separate communication channel such as JTAG), the typical volume of trace data, the limited number and size of large memories, and the limited number and bandwidth of communication channels will generally necessitate sharing of memory (e.g., the FPGA on-board memory or the CPU main memory), the communication channel, or both.

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 the performance analysis tool to handle these complexities. 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 also need to be shared if used by the application. When CPU-initiated data transfers are used, the application and performance monitoring software must use locks to guarantee they do not access the FPGA simultaneously.

Schemes to minimize the perturbation of application performance (Goal 1) must also be considered when sharing any resource. These schemes generally reduce to conflict resolution between the performance framework and the application. With FPGA-initiated data transfers, decisions may be fairly fine-grained, allowing a performance data transfer to be interrupted to permit application use of the shared resource. CPU-initiated data transfers typically cannot be interrupted, requiring coarse-grained techniques such as adapting the polling frequency dynamically to account for application use and available performance data.

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 the application's communication scheme and seamlessly integrate with it. Communication schemes such as memory maps or network packets are used with a variety of interconnects and by diverse APIs for FPGAs. In order to automate instrumentation, the tool must be able to detect and assign some unused portion of the application's address space (or some other unique identifier not used by the application), connect to the application at the proper level in the design in order to avoid dealing with the details of a specific interconnect, and detect and add locks around all FPGA communication in the software portion of the application. Providing both automated instrumentation and measurement techniques to support shared resources is thus a significant challenge for measurement in RC performance analysis.

3.3 Challenges for performance presentation

Conventional trace-based display tools such as Jumpshot [17] use timeline views to show communication and computation for parallel computing applications. These timeline views can be extended to include FPGAs as additional processing elements for RC applications. A (mockup) visualization example is shown in Figure 3. In this example, the CPUs (nodes 0-7) are performing work and receiving data from the FPGAs (nodes 8-15). Nodes 3 (CPU) and 11 (FPGA) complete first near the middle of the diagram, while nodes 4 and 12 are lagging, completing toward the end, finally allowing global synchronization of all nodes before a new iteration begins.

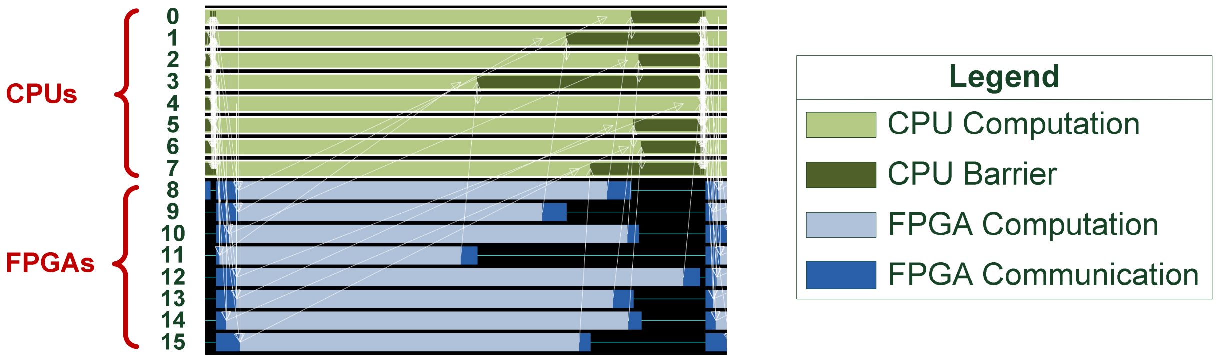

Figure 3: Mockup Jumpshot visualization of an RC application with 8 CPUs and 8 FPGAs

Unfortunately, such timeline views scale poorly as system sizes increase to hundreds or thousands of nodes. Knüpfer et al. argue that as trace data sizes continue to grow, timeline views must continue to show smaller fractions of this data and yet still convey meaningful information to the user [18]. They propose detecting repetitive patterns and collapsing these patterns visually into a single box, thus highlighting irregular behavior in the application to combat these scalability issues. These issues are present in RC performance presentation as well and are further complicated by the task of concisely, yet accurately, displaying the heterogeneous parallelism inside the FPGA. Traditional performance analysis tools treat any possibility for parallel execution (e.g., via multiple cores in a CPU, CPUs in a symmetric multiprocessor, or nodes in a cluster) as separate "threads" that serially run functions, communicate, wait for other nodes to complete, etc. Due to the vast amount of parallelism possible in an FPGA, such an abstraction can be unwieldy (violating Goal 5). Worse yet, treating the FPGA as a large uniform multicore device is inaccurate, given the differing types of components and the specific hierarchy within which they exist. On the opposite end of the spectrum, portraying an FPGA as a single processing element (as is done in Figure 3) may not be useful as this excludes too much parallelism inside the FPGA.

Hierarchical views may be better suited to the heterogeneous devices and behavior in a large-scale RC system. For example, Figure 4 shows one possibility of such a display capturing both potential and actual communication and computation in the context of a system. Actual communication rates are given numerically, while the maximum bandwidth for each channel is depicted in the associated rectangular boxes. Computation is depicted similarly, representing the percentage of time that the device or FPGA core was busy. From the figure, four possible performance bottlenecks are readily visible: the network interface, the communication channel to CPU 1, the two cores in FPGA 1, and the communication and computation surrounding CPUs 4 and 5 and FPGA 2. It is also evident that there is relatively little use of resources such as CPUs 2 and 3 and FPGA 0. A more detailed profile and trace data view could be integrated when the user clicks on a node or communication channel (shown on left of figure). For example, from the detailed communication channel view, the designer is able to see the duration of a communication spike that would not be evident from a statistical average, maximum, or minimum.

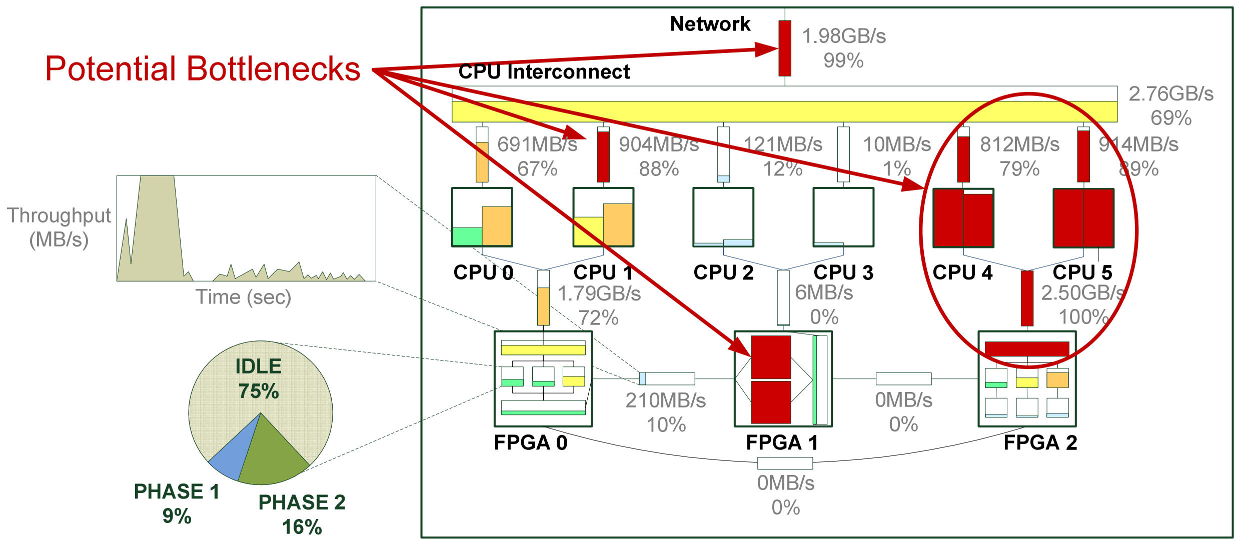

Figure 4: Example hierarchical display of a complex RC application

From a practical standpoint, obtaining the communication and computation shown in Figure 4 is fairly straightforward, requiring profile (and optionally trace) data to be collected on both the CPUs and FPGAs. Generating the diagram layout automatically is non-trivial since the system architecture must be understood, but this problem could be handled by querying the system for as many parameters as possible, filling in the remaining gaps manually for a given system; this process would only need to be done once per system unless the configuration was changed. If actual sustained throughput values are desired rather than maximum theoretical bandwidth values, these must be obtained as well, usually via microbenchmarks.

Ultimately, the purpose of any performance visualization is to aid the designer in forming strategies for optimization. For example, one possible improvement to the application in Figure 4 would offload some of the workload from FPGA 1 to underutilized resources such as FPGA 0 or CPUs 2 and 3 (or both). Another possibility consists of sharing some tasks currently assigned to CPUs 4 and 5 with CPUs 2 and 3. Both of these improvements are dependent on the ability to further parallelize or partition tasks, and have the possibility of affecting communication significantly. For example, CPUs 4 and 5 have nearly saturated their communication channels; moving some of the work to CPUs 2 and 3 may incur additional communication or may distribute some of the communication from CPUs 4 and 5 to CPUs 2 or 3. Given potential bottlenecks and solutions, the designer can apply application knowledge to make the appropriate optimizations.

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 can take advantage of strategically choosing where to monitor a specific event (i.e., from 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 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 models such as UPC and SHMEM as well as to message-passing models such as MPI. PGAS performance analysis is enabled by way of the Global Address Space Performance (GASP) interface [20], which specifies the interaction between a performance tool (such as PPW) and the programming model implementation (such as Berkeley UPC or 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-defined 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 can 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 average bytes transferred, maximum time taken to reconfigure the FPGA, or minimum latency observed, providing a detailed view of FPGA communication from software. However, automatically adding these extended GASP functions around FPGA communication is difficult due to the variety of ways FPGA communication can appear in software. Ideally, a standard API for FPGA access could make detection of FPGA calls trivial. In the absence of such a standard, the performance analysis tool must detect each vendor's FPGA access methods and map them to the appropriate generic event.

4 Framework

In this section we propose an initial framework to instrument and measure an FPGA's performance at runtime using the same communication channel used by the application for performance data transfers. For simplicity, portability, and flexibility in what can be monitored, our framework employs source instrumentation (specifically of VHDL). To ensure applicability to systems without FPGA-initiated transfers, interrupts, or access to other large memories, we use CPU-initiated retrieval of FPGA performance data at runtime using only on-chip FPGA resources. We discuss the instrumentation methodology first, followed by the measurement portion of the framework.

4.1 Instrumentation

Figure 5 illustrates the changes to an RC application during instrumentation in order to support measurement. These changes can be divided into the following seven steps:

1. All signals, variables, component ports, and other data available in the HDL source files are enumerated along with their types, locations in the hierarchy, and other useful information. This information is gathered by parsing the user's VHDL code directly via a standard VHDL grammar and parser.

2. An automated selection of data is made based on a desire to monitor communication and computation. For example, all state machines can be profiled, average use of the input and output ports of the top-level file can be monitored, and any identifiable control signals for subcomponents can be profiled or traced. This automation is based on common practices in VHDL code (e.g., state machines are often used for component control and generally appear in case statements), and thus may fail to find data to monitor (or conversely may monitor unnecessary data) depending on coding style and the nature of the

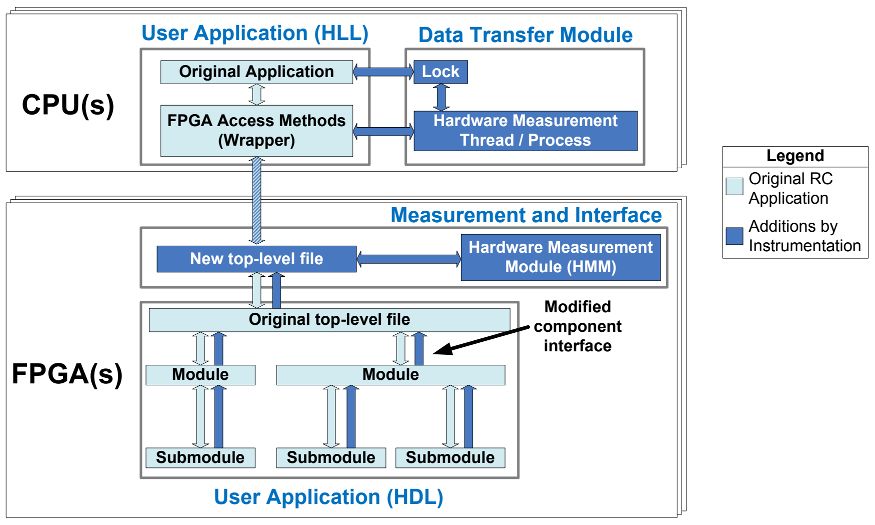

Figure 5: Additions made by source-level instrumentation of an RC application

application. Therefore, the user is optionally given the ability to override any decision made by the tool by adding or removing items to monitor (e.g., signals, variables, component ports), specifying whether to use profiling or tracing, and choosing the amount of FPGA resources to devote to monitoring.

3. Given the final list of what data is to be monitored (and how to monitor it), the tool automatically modifies a copy of the user's VHDL code to output all monitored data to the user's top-level file (illustrated in Figure 6a). Data types are converted to `std_logic_vector` whenever possible for simplicity of monitoring (e.g., enumerated types are converted to `std_logic_vector` using the state's position in the enumerated list); any type that is not understood is passed out as a new type (e.g., `HMM_Type1`) defined identically to the original type, assuming the user will manually handle the analysis. Note that the new type ensures the user's data can traverse the component hierarchy to the top-level file; the user's original type may have been defined only in the component where it was used. Each component in the design's hierarchy must output its monitored data as well as all monitored data from its subcomponent(s), if any.

4. A new top-level file is created by duplicating the user's top-level file interface and splicing into the communication scheme to allow the performance tool to gain access to the interface as well (e.g., the performance tool might be assigned unused memory space to allow routing of incoming data to the correct location). This step is extremely difficult to automate in a fool-proof manner and thus is permitted to fail, allowing manual control by the user if necessary. Note that the new top-level file is permitted to have additional HDL files above it (e.g., a wrapper to interface with a bus or other interconnect) that the user has no interest in instrumenting.

5. The signals, variables, and other data to be monitored are then connected to the Hardware Measurement Module (HMM) (see next section), which handles all recording of performance data and transferring of that data to the CPU when requested. Any analysis or combination of the signals is handled here, such as triggering an event only if both the error flag and write enable are true. If only part of a signal or a subset of components needs to be monitored, then only these signals or components are connected to the HMM, leaving the remainder unconnected for removal by the synthesis tool.

6. In software, a data transfer module is added immediately after the user's initialization of the FPGA. This module will execute as a separate thread that periodically polls the FPGA for performance data and then transfers that data to main memory. This thread uses generic FPGA calls, coupled with the appropriate mapping between these calls and the actual vendor-specific FPGA calls, to improve portability.

7. A lock is placed around any call to the FPGA in the user's application, as the FPGA must be guarded against simultaneous access by the application and data transfer thread. The same lock is already present in the data transfer thread (this and previous step are illustrated in Figure 6b).

In step 6, the software interface to launch and manage this thread is wrapped into four simple calls: `HMM_Init`, `HMM_Start`, `HMM_Stop`, and `HMM_Finalize` (shown in Figure 6b). The initialize and finalize routines manage the setup and cleanup of all necessary memory and thread resources on the software side. The start and stop routines act as a stopwatch, launching and stopping the data transfer thread (and optionally the hardware measurement itself) for portions of code that do not require monitoring. In general, it is possible to override key vendor API functions to manage this thread automatically, allowing HLL source instrumentation (steps 6 and 7) to be fully automated. Note that source instrumentation of the user's HDL requires the application to be resynthesized and implemented before executing, while source instrumentation of the user's HLL requires use of the vendor's API to access the FPGA.

```
--Application libraries
use work.HMM_Types.all;

entity ... is port (
  HMM_Data1 : out HMM_Type1;
  HMM_Data2 : out HMM_Type2;
  -- Application signals
);
end ...

architecture ... Is
...
begin
  HMM_Data1 <= App_Value_1;
  ...
  process ... begin
    HMM_Data2 <= App_Value_2;
    ...
  end process;
end ...
```

(a) HDL source code modifications

```
#include "HMM.h"

int main() {
  // Application initializes FPGA
  HMM_Init(...);
  ...
  HMM_Start(...);
  ...
  // Application FPGA call
  pthread_mutex_lock(lock);
  FPGA_Write(...);
  pthread_mutex_unlock(lock);
  ...
  HMM_Stop(...);
  ...
  HMM_Finish(...);
  // Application closes FPGA
}
```

(b) HLL source code modifications

Figure 6: Instrumentation of user source code

4.2 Measurement

At the center of measurement in the FPGA is the Hardware Measurement Module (HMM). The HMM is responsible for implementing all profile, trace, and sampling capabilities, as well as packaging that data for retrieval by software. The HMM allows quick customization of (and easy access to) all of these resources, eliminating the time-consuming and error-prone process of manually measuring performance. Features include arbitrary counter and trace sizes (limited by resources); storage of maximums, minimums, and averages of selected values; counters for each trace buffer indicating the number of records dropped (trace records may be dropped due to insufficient buffer space); ability to clear, stop, hold, and acknowledge errors in profile and trace units; packaging of all performance data to the specified interface width for export to the CPU; and storage of records in logic cells, block RAM, or on-board memories (if available). Figure 7 illustrates the design of the HMM.

At runtime, the polling thread inserted by instrumentation periodically retrieves all trace data (and optionally profile data as well) from the FPGA. To minimize perturbation of the application's communication channel, the polling rate can be adaptive, increasing or decreasing based upon a target usage of the communication channel, the application's usage of the communication channel, or the number of recently dropped trace records on the FPGA. The HMM receives a request for profile data, trace data, or module statistics (e.g., dropped trace records), and splits the data up into pieces the size of the communication channel width. The HMM can also receive commands to clear, stop, or acknowledge overflows of profile counters and trace

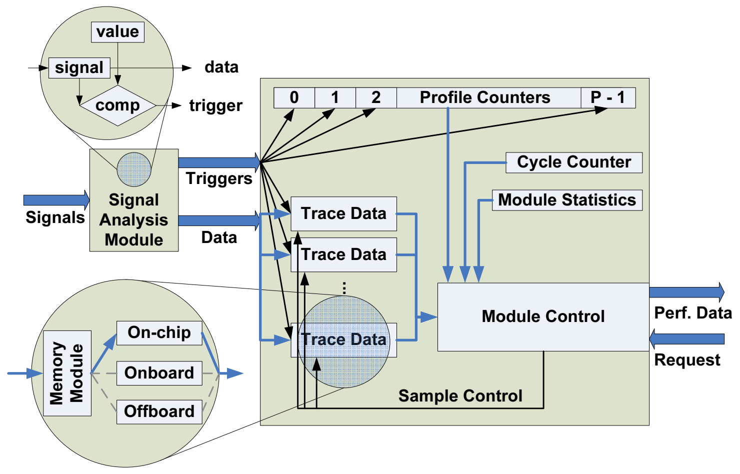

Figure 7: Hardware Measurement Module

buffers. Sampling capabilities are also available, allowing trace buffers to record for a specified number of cycles. Once data is retrieved from the HMM, this data may be lightly processed to reduce storage overhead.

5 Case Study

To demonstrate the benefits and importance of RC performance analysis techniques, as well as explore the associated overhead, we present results from a case study using a prototype version of the framework discussed in Section 4. In our prototype version, the instrumentation steps are performed manually (Section 4.1), with profile counters and a trace buffer available in the HMM. The data transfer module retrieves data at a fixed rate, polling once per millisecond.

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 Infiniband 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, while compilation for the 16-node Infiniband 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 $N \times N$ 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.

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 (the application used very little bandwidth as well, polling the device only once per

100 milliseconds). Less than 7% of the FPGA's logic 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 larger LX100 devices in the Nallatech cluster.

It is important to note that even though some decrease in maximum frequency may occur, some FPGA systems have only coarse-grained or fixed clocks, polarizing the importance of frequency degradation. For example, if the FPGA clock operates at either 75 or 100MHz, a drop from 102MHz to 101MHz would have no effect while a drop from 102MHz to 99MHz would necessitate a significant drop in FPGA speed, reducing the accuracy of performance data measured. However, both the XD1 and the Nallatech hardware allow for fine-grained control of the clock, permitting a change of 1MHz or less via the Digital Clock Managers (DCMs).

Table 2: Performance Analysis Overhead

	Original (XD1)	**Instrumented (XD1)**	**Resource change (XD1)**
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
	Original (Cluster)	**Instrumented (Cluster)**	**Resource change (Cluster)**
Slices (49152 total)	23086 (47%)	26218 (53%)	+3132 (6.4%)
Block RAM (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 accessible 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 8. 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 8 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, yielding a potential speedup of 16.3% versus the non-optimized version (based upon removing these states from the graph); the actual speedup is expected to be less as communication and setup portions of the application have not changed. While merging states could reduce the maximum frequency of the design, 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.

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 theoretically improve speedup by 11%.

Figure 9 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.

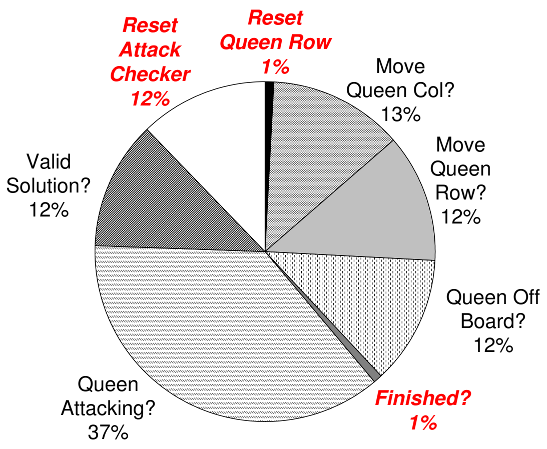

Figure 8: Distribution of cycles spent in core state machines of N-Queens

6 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, difficulties in representing increasingly large FPGAs and RC systems in meaningful visualizations are significant barriers as well. To address these challenges, we proposed a framework to instrument an RC application and measure runtime performance data as well as concepts for visualizations involving RC resources. We argued 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 trace buffers 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 as well as development of large-scale visualizations will be critical in order for performance analysis of RC applications to achieve widespread use. Recording data from additional, more complex designs, including designs with multiple clock domains or an embedded processor, is also important. In addition, performance analysis of hardware generated from a high-level language is extremely important since another layer of abstraction is added, further obfuscating application behavior. 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.

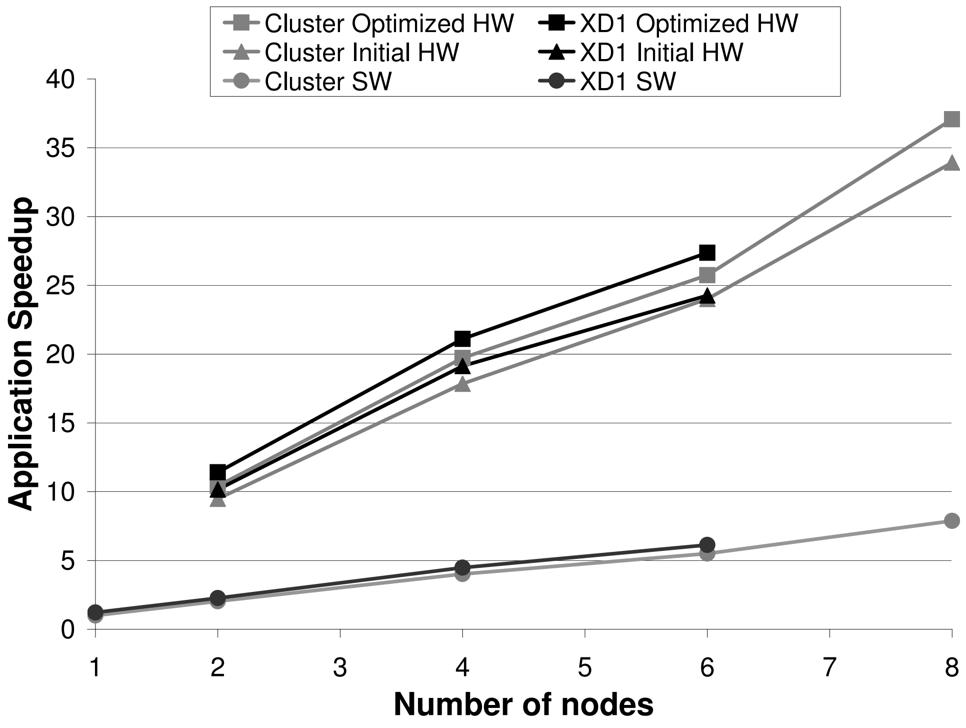

Figure 9: Speedup of N-Queens Application

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] Melissa C. Smith, Jeffery S. Vetter, and Xuejun Liang. Accelerating scientific applications with the SRC-6 reconfigurable computer: Methodologies and analysis. In *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS) - Workshop 3*, page 157.2, Washington, DC, USA, Apr. 2005. IEEE Computer Society.

[2] Justin L. Tripp, Anders A. Hanson, Maya Gokhale, and Henning Mortveit. Partitioning hardware and software for reconfigurable supercomputing applications: A case study. In *Proc. of the 2005 ACM/IEEE Conference on Supercomputing (SC)*, page 27, Washington, DC, USA, Nov. 2005. IEEE Computer Society.

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

[4] XDI. XD1000$^{\text{TM}}$ FPGA coprocessor module for Socket 940. http://www.xtremedatainc.com/pdf/XD1000_Brief.pdf.

[5] DRC. RPU110: DRC reconfigurable processor unit. http://www.drccomputer.com/pdfs/DRC_RPU110_datasheet.pdf, 2007.

[6] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications (HPCA)*, 20(2):287–311, May 2006.

[7] I-Hsin Chung, Robert E. Walkup, Hui-Fang Wen, and Hao Yu. MPI performance analysis tools on Blue Gene/L. In *Proc. of the 2006 ACM/IEEE Conference on Supercomputing (SC)*, page 123, New York, NY, USA, Nov. 2006. ACM Press.

[8] Kevin Camera, Hayden Kwok-Hay So, and Robert W. Brodersen. An integrated debugging environment for reprogrammble hardware systems. In *Proc. of the 6th International Symposium on Automated Analysis-Driven Debugging (AADEBUG)*, pages 111–116, New York, NY, USA, Sep. 2005. ACM Press.

[9] Altera. Design debugging using the SignalTap II embedded logic analyzer. http://www.altera.com/literature/hb/qts/qts_qii53009.pdf, May 2007.

[10] Xilinx. Xilinx ChipScope Pro software and cores user guide, v. 9.2i. http://www.xilinx.com/ise/verification/chipscope_pro_sw_cores_9_2i_ug029.pdf, May 2007.

[11] R. DeVille, I. Troxel, and A. George. Performance monitoring for run-time management of reconfigurable devices. pages 175–181, June 2005.

[12] Martin Schulz, Brian S. White, Sally A. McKee, Hsien-Hsin S. Lee, and Jürgen Jeitner. Owl: next generation system monitoring. In *Proc. of the 2nd Conference on Computing frontiers (CF)*, pages 116–124, New York, NY, USA, May 2005. ACM Press.

[13] Paul Graham, Brent Nelson, and Brad Hutchings. Instrumenting bitstreams for debugging FPGA circuits. In *Proc. of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 41–50, Washington, DC, USA, Apr. 2001. IEEE Computer Society.

[14] Steve A. Guccione, Delon Levi, and Prasanna Sundararajan. Jbits: A Java-based interface for reconfigurable computing. In *Proc. of 2nd Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, page 27, Sep. 1999.

[15] F. Cristian. A probabilistic approach to distributed clock synchronization. pages 288–296, June 1989.

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

[17] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From trace generation to visualization: a performance framework for distributed parallel systems. In *Proc. of the 2000 ACM/IEEE Conference on Supercomputing (CDROM) (SC)*, page 50, Washington, DC, USA, Nov. 2000. IEEE Computer Society.

[18] Andreas Knüpfer, Bernhard Voigt, Wolfgang E. Nagel, and Hartmut Mix. Visualization of repetitive patterns in event traces. In *Proc. of Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, June 2006.

[19] Adam Leko and Max Billingsley, III. Parallel performance wizard user manual. http://ppw.hcs.ufl.edu/docs/pdf/manual.pdf, 2007.

[20] Adam Leko, Dan Bonachea, Hung-Hsun Su, Hans Sherburne, Bryan Golden, and Alan D. George. GASP! A standardized performance analysis tool interface for global address space programming models. In *Proc. of Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, June 2006.

[21] Nallatech. H100 series FPGA application accelerators. http://www.nallatech.com/mediaLibrary/images/english/5595.pdf, Apr. 2007.

[22] Cengiz Erbas, Seyed Sarkeshik, and Murat M. Tanik. Different perspectives of the N-Queens problem. In *Proc. of the 1992 ACM Annual Conference on Communications (CSC)*, pages 99–108, New York, NY, USA, Mar. 1992. ACM Press.