RECAP: A NOVEL FRAMEWORK FOR APPLICATION PERFORMANCE OPTIMIZATION IN RECONFIGURABLE COMPUTING

By

SETH L. KOEHLER

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2010
I dedicate this dissertation to my Lord and savior, Jesus Christ – I would not be here without you; my wife, who has loved and supported me greatly throughout this journey (and who has made more chocolate cakes for my late-night work than should be numbered); my kids, Sammy and Emma, who have shown me love that can only come from your children – I am privileged to be your father; my parents (both mine and my wife’s) for prayers, financial support, advice, and general moral support; my grandparents, siblings, other family members, and friends for your prayers, thoughts, and for good conversations that make me think.

I am truly grateful for you all.
ACKNOWLEDGMENTS

This work is supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422 and by equipment or tools provided by Aldec, Altera, Cray, GiDEL, Nallatech, Synplicity, Xilinx, and XtremeData. I would like to thank my committee – Dr. Timothy Davis, Dr. Ann Gordon-Ross, Dr. Herman Lam, Dr. Jih-Kwon Peir, and Dr. Gregory Stitt – for their advice and assistance as well as my advisor, Dr. Alan D. George, for his guidance, direction, and significant support through this work. I would also like to thank John Curreri for his work on an N-Queens application case study as well as overall insight into various parts of this work. Additionally, I thank Dr. Karthik Nagarajan for his initial implementation of a two-dimensional probability density function estimator used as a case study in this work. Finally, I would like to express my thanks to Rafael Garcia who aided significantly in the production of our first, throw-away prototype of an RC performance analysis tool.
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>7</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>8</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>10</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1  INTRODUCTION</td>
<td>12</td>
</tr>
<tr>
<td>2  BACKGROUND AND RELATED RESEARCH</td>
<td>18</td>
</tr>
<tr>
<td>2.1  Reconfigurable Computing</td>
<td>18</td>
</tr>
<tr>
<td>2.2  Performance Analysis of Parallel Applications</td>
<td>21</td>
</tr>
<tr>
<td>2.2.1  Visualization</td>
<td>23</td>
</tr>
<tr>
<td>2.2.2  Knowledge-Based Bottleneck Detection</td>
<td>25</td>
</tr>
<tr>
<td>2.3  FPGA Performance Modeling and Simulation</td>
<td>27</td>
</tr>
<tr>
<td>2.4  FPGA Performance Analysis</td>
<td>29</td>
</tr>
<tr>
<td>3  PERFORMANCE ANALYSIS FRAMEWORK FOR RECONFIGURABLE-COMPUTING APPLICATIONS (PHASE 1)</td>
<td>32</td>
</tr>
<tr>
<td>3.1  Challenges for RC Performance Analysis</td>
<td>32</td>
</tr>
<tr>
<td>3.1.1  Challenges for Hardware Instrumentation</td>
<td>33</td>
</tr>
<tr>
<td>3.1.1.1  What to instrument</td>
<td>33</td>
</tr>
<tr>
<td>3.1.1.2  Levels of instrumentation</td>
<td>35</td>
</tr>
<tr>
<td>3.1.1.3  Modifying the application</td>
<td>37</td>
</tr>
<tr>
<td>3.1.2  Challenges for Hardware Measurement</td>
<td>38</td>
</tr>
<tr>
<td>3.1.2.1  Recording and storing performance data</td>
<td>39</td>
</tr>
<tr>
<td>3.1.2.2  Managing shared resources</td>
<td>41</td>
</tr>
<tr>
<td>3.1.3  Challenges for Performance Presentation</td>
<td>43</td>
</tr>
<tr>
<td>3.1.4  Unified Performance Analysis Tool</td>
<td>46</td>
</tr>
<tr>
<td>3.2  Framework</td>
<td>47</td>
</tr>
<tr>
<td>3.2.1  Instrumentation</td>
<td>48</td>
</tr>
<tr>
<td>3.2.2  Measurement</td>
<td>50</td>
</tr>
<tr>
<td>3.3  Case Study</td>
<td>52</td>
</tr>
<tr>
<td>3.4  Conclusions</td>
<td>56</td>
</tr>
<tr>
<td>4  PERFORMANCE VISUALIZATION AND EXPLORATION (PHASE 2)</td>
<td>58</td>
</tr>
<tr>
<td>4.1  RC Performance Visualization</td>
<td>60</td>
</tr>
<tr>
<td>4.2  RC Performance Exploration</td>
<td>65</td>
</tr>
<tr>
<td>4.3  Results</td>
<td>73</td>
</tr>
</tbody>
</table>
4.4 Conclusions ................................................................. 75

5 PLATFORM-AWARE KNOWLEDGE-BASED BOTTLENECK DETECTION (PHASE 3) ................................................................. 76

5.1 Platform Templates ....................................................... 78
  5.1.1 Software .............................................................. 78
  5.1.2 Hardware ............................................................. 82

5.2 Bottleneck Detection in RC Applications ................................ 83

5.3 Common Bottlenecks in RC Applications ............................ 93
  5.3.1 Communication Bottlenecks ....................................... 95
  5.3.2 Synchronization Bottlenecks ...................................... 98
  5.3.3 Internal Overhead Bottlenecks .................................... 101
  5.3.4 Imbalance Bottlenecks ............................................. 102

5.4 Case Studies ............................................................... 103
  5.4.1 Time-Domain Finite Impulse Response .......................... 103
  5.4.2 2-D Probability Density Function Estimation ................. 108

5.5 Conclusions ................................................................. 112

6 CONCLUSIONS ................................................................. 114

APPENDIX

A SYSTEMS SUPPORTED BY RECAP TOOL ............................ 116
  A.1 Nallatech H101 Cluster ............................................... 116
  A.2 XtremeData XD1000 .................................................... 116
  A.3 GiDEL PROCStar III Cluster ......................................... 117

B APPLICATION CASE STUDIES ........................................ 118
  B.1 N-Queens ............................................................... 118
  B.2 Time-Domain Finite Impulse Response ......................... 118
  B.3 Collatz Conjecture .................................................... 118
  B.4 2-D Probability Density Function Estimation ................. 119

REFERENCES ................................................................. 120

BIOGRAPHICAL SKETCH ................................................... 128
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Comparison of source and binary instrumentation.</td>
<td>37</td>
</tr>
<tr>
<td>3-2</td>
<td>Performance Analysis Overhead.</td>
<td>53</td>
</tr>
<tr>
<td>5-1</td>
<td>RC platforms employed during case studies.</td>
<td>104</td>
</tr>
<tr>
<td>5-2</td>
<td>Datasets evaluated for TDFIR benchmark.</td>
<td>105</td>
</tr>
<tr>
<td>5-3</td>
<td>2DPDF results for both the Nallatech and XD1000 platforms. Speedup is given with respect to the software baseline executed on a Pentium 4 Xeon 3.2GHz CPU.</td>
<td>112</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1-1</td>
<td>Potential communication bottlenecks (represented by arrows) in RC applications.</td>
<td>13</td>
</tr>
<tr>
<td>2-1</td>
<td>Typical island layout of FPGA logic and routing fabric.</td>
<td>19</td>
</tr>
<tr>
<td>2-2</td>
<td>Stages of performance analysis.</td>
<td>23</td>
</tr>
<tr>
<td>3-1</td>
<td>Mockup Jumpshot visualization of an RC application with 8 CPUs and 8 FPGAs.</td>
<td>43</td>
</tr>
<tr>
<td>3-2</td>
<td>Example hierarchical display of a complex RC application.</td>
<td>45</td>
</tr>
<tr>
<td>3-3</td>
<td>Additions made by source-level instrumentation of an RC application.</td>
<td>47</td>
</tr>
<tr>
<td>3-4</td>
<td>Instrumentation of user source code.</td>
<td>50</td>
</tr>
<tr>
<td>3-5</td>
<td>Hardware Measurement Module.</td>
<td>51</td>
</tr>
<tr>
<td>3-6</td>
<td>Distribution of cycles spent in core state machines of N-Queens.</td>
<td>55</td>
</tr>
<tr>
<td>3-7</td>
<td>Speedup of N-Queens Application.</td>
<td>56</td>
</tr>
<tr>
<td>4-1</td>
<td>Traditional timeline visualization (Jumpshot) example demonstrating difficulties associated with presenting RC data</td>
<td>59</td>
</tr>
<tr>
<td>4-2</td>
<td>Example of user-defined pragmas.</td>
<td>61</td>
</tr>
<tr>
<td>4-3</td>
<td>ReCAP visualizations for 2-core Collatz application.</td>
<td>64</td>
</tr>
<tr>
<td>4-4</td>
<td>Example of a system-level visualization of an RC application executing on a 6-CPU / 3-FPGA RC system assumed to be part of a larger cluster</td>
<td>66</td>
</tr>
<tr>
<td>4-5</td>
<td>Example of a node-level visualization of an RC application, showing 2 CPUs and an FPGA from the left hand side of the system-level visualization</td>
<td>66</td>
</tr>
<tr>
<td>4-6</td>
<td>Example application for exploration.</td>
<td>68</td>
</tr>
<tr>
<td>4-7</td>
<td>Potential timelines when optimizing an example application.</td>
<td>68</td>
</tr>
<tr>
<td>4-8</td>
<td>Performance exploration methodology.</td>
<td>70</td>
</tr>
<tr>
<td>4-9</td>
<td>Performance exploration using CST/CT updates.</td>
<td>72</td>
</tr>
<tr>
<td>4-10</td>
<td>Performance exploration for Collatz application.</td>
<td>74</td>
</tr>
</tbody>
</table>
5-1 Directed graph of an RC application that takes input from a sensor, processes data using a two-core pipeline, potentially offloads data to threads on a multi-core CPU for further processing, and finally stores results in DDR memory. 84

5-2 Examples of user-defined pragmas. 86

5-3 Default “reasons” provided by ReCAP for classifying API calls or HDL branches. 89

5-4 Example of bottleneck detection results, showing inclusion of warning icons to indicate blocks with bottlenecks and a portion of the detailed bottleneck report. 92

5-5 Taxonomy of common bottlenecks in an RC system. 94

5-6 Platform transfer rate vs. transfer size across various RC systems and communication types, demonstrating common communication problems. The non-blocking transfer rate was computed as the sustained cumulative transfer rate of eight concurrent non-blocking transfers. 97

5-7 TDFIR performance on various devices (including both the original and optimized FPGA performance after bottleneck detection. 106

5-8 Speedup of initial and improved versions of the 2DPDF application when compared to a CPU baseline. 109
Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

RECAP: A NOVEL FRAMEWORK FOR APPLICATION PERFORMANCE OPTIMIZATION IN RECONFIGURABLE COMPUTING

By
Seth L. Koehler
December 2010

Chair: Alan D. George
Major: Computer Engineering

Reconfigurable-computing (RC) applications employing both microprocessors and FPGAs (field-programmable gate arrays) have potential for large speedup when compared with traditional (software-based) parallel applications. However, this potential is marred by the additional complexity of these hybrid 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 document, we explore challenges, tradeoffs, and various techniques for automated instrumentation, low-overhead measurement, hierarchical visualization, performance exploration, common bottleneck detection, and tool-assisted optimization for RC applications. We specifically propose the Reconfigurable-Computing Application Performance (ReCAP) framework to address these challenges and provide a cohesive structure for the various tradeoffs and techniques discussed. ReCAP seeks to facilitate or automate much of the time-consuming and error-prone processes associated with manually analyzing performance for an RC application, enabling an application designer to more productively locate and remedy performance bottlenecks. Through this research, novel concepts as well as infrastructure for performance analysis and optimization of RC applications are introduced, analyzed, prototyped, and evaluated. Case studies using
a prototype ReCAP tool are provided across a representative set of RC systems and applications to demonstrate the effectiveness of this framework. This tool is believed to be the first of its kind for RC, a relatively new but increasingly important paradigm of computing.
CHAPTER 1
INTRODUCTION

While the performance of parallel computing systems (e.g., multicore CPUs, clusters, multiprocessor system-on-chips (MPSoCs)) continues to increase in both high-performance embedded computing (HPEC) and high-performance computing (HPC), the growing importance of power consumption has caused programmers in both fields to rely increasingly on explicit parallelism and accelerators rather than on instruction-level parallelism or clock frequency increases [1], [2], [3]. Due to these trends, reconfigurable computing (RC) [4], which typically employs both CPUs and reconfigurable hardware such as field-programmable gate arrays (FPGAs), has emerged as a viable field for providing orders-of-magnitude performance gains over microprocessors [5], [6] while simultaneously consuming little power in comparison to both microprocessors and GPUs [7], [8], [9]. However, the behavior of an RC application can be particularly difficult to observe and understand due to additional levels of hierarchical parallelism and complex interactions between heterogeneous resources inherent in such systems [10]. 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 tools (tools that record application behavior during execution on a given system) are indispensable for application analysis and optimization, even more so than in traditional parallel computing 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 [11] or those employing CPU-socket-compatible FPGA boards (e.g., XtremeData [12] or DRC [13]) are advancing the FPGA from slave to peer with CPUs, enabling the FPGA to independently interact with resources including main memory, CPUs, or other
FPGAs. Due to the expanded capabilities of FPGAs in RC applications, conventional tools have an increasingly incomplete view of application performance, necessitating hardware-aware performance analysis tools that can provide a complete view of RC application performance. To illustrate this need, Figure 1-1 shows the hierarchy of parallelism and myriad of interactions inside an RC system, differentiating between communication that can be monitored (light arrows) and communication that cannot be monitored (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 application designer’s ability to understand and improve application performance.

Figure 1-1. Potential communication bottlenecks (represented by arrows) in RC applications.

To address a current lack of RC performance analysis research and tools, we present the details of our RC performance analysis framework and tool, ReCAP (reconfigurable computing application performance), which we believe to be the first of its kind. ReCAP is a single framework and tool for obtaining, visualizing, and analyzing performance data for an application using both CPU and FPGA devices inside an RC system, thus presenting a unified view of RC application behavior at runtime. We draw upon research, concepts, and techniques from HPC and HPEC performance analysis including instrumentation, measurement, visualization, bottleneck detection, and optimization strategies as well as from FPGA debugging, performance prediction, and tool portability. Without the support of a performance tool that implements such a
framework, application designers are forced to manually obtain access and interpret information believed to affect performance as well as formulate optimizations and predict their effects on the application, requiring a great deal of designer effort and expertise. ReCAP specifically provides statistical and timing information for application events, displays RC-targeted visualizations, detects common bottlenecks, and suggests potential optimization strategies for detected bottlenecks, thus permitting rapid, intuitive understanding of application behavior in both software and hardware, often with little-to-no user assistance. Simultaneously, we attempt to minimize resource and runtime overhead, reducing the chance that monitoring will alter the performance behavior of the application. With these capabilities, ReCAP aims to significantly increase application designer productivity, assisting designers throughout the entire optimization process.

We subdivide our research into three phases. The first phase involves exploration of the challenges faced in attaining low-overhead, automatable techniques for instrumentation and runtime measurement of RC applications as well as for basic visualization of performance data. Techniques are discussed to address these challenges and details of the ReCAP framework are presented, providing a comprehensive methodology for RC performance analysis. This framework is then integrated into an existing parallel performance analysis tool, Parallel Performance Wizard (PPW) [14], [15], and results from a case study are provided to demonstrate ReCAP’s utility and minimal overhead.

The second phase of this research then details an extension to the research presented in Phase 1 in order to provide performance visualization and exploration for RC applications. Traditional visualization strategies have serious limitations for RC due to their assumption of typically homogeneous, fixed-architecture devices (e.g., CPUs). In contrast, RC applications commonly employ intricate hierarchies of heterogeneous parallelism that does not scale well within current visualizations. Further, visualizations
cannot abstract away most of the hardware (as is typical for CPU-based visualization) since this hardware is now part of the designer's application. Without RC-targetted visualization, an application designer is hindered in their ability to quickly understand the behavior of, and thus optimize, their application's behavior. Additionally, once an application designer formulates potential optimizations, it is often unclear, due to complexity inherent in RC applications, as to what the effects of these optimizations will be on the entire application's performance. While design-space exploration (DSE) and performance prediction techniques are well researched, these techniques are typically employed before an application is constructed, and thus must make assumptions that can reduce prediction accuracy. In contrast, ReCAP’s performance exploration employs runtime performance data to predict the effects of an optimization, sacrificing flexibility in the magnitude of changes that can be modeled for potentially increased prediction accuracy. This phase will involve the study of visualization and DSE techniques as well as the development of visualizations within ReCAP. These techniques are then demonstrated within an application case study.

In the third phase of this research, we present a general framework for platform-aware, knowledge-based bottleneck detection, aiding an application designer in quickly locating and remedying performance bottlenecks across a diverse range of RC platforms. Applications often contain similar bottlenecks such as load-balancing or communication-related problems, but locating and resolving these bottlenecks remains a difficult and tedious process, especially given different approaches needed to effectively monitor and address these bottlenecks on diverse RC systems and APIs. The presented framework and tool are extensible in that users may easily add support for their own platform, as long as it fits within ReCAP’s platform-template model, and may also easily modify the bottleneck knowledge base. The first taxonomy of common RC bottlenecks, per an extensive literature review, is also presented (including detection and optimization strategies) and used to populate ReCAP’s knowledge base. Although useful for novice
RC designers who may be unaware of many potential bottlenecks and optimizations, experienced RC designers can benefit as well from quick feedback on the location and severity of performance problems.

It is important to note that while we focus on improving runtime performance throughout this research, these concepts and techniques can also be of use for reducing power consumption or resource usage as well. For example, an increase in raw application performance may allow an application designer to decrease the number of hardware cores or the clock frequency on the FPGA while still providing the required performance. Conversely, optimization suggestions (such as those given in Phase 3) may indicate that resources could be reduced without sacrificing performance, allowing these resources to be repurposed elsewhere for improved performance or to be left unused, reducing power or even permitting a smaller FPGA to be employed. Thus, optimizations may be used to achieve a balance amongst runtime performance, resource usage, and power consumption.

This research consists of background, details, and results from each phase of research as well as concluding remarks and appendices. Specifically, Chapter 2 provides background and related research in the area of traditional (CPU-based) performance analysis including visualization and bottleneck detection, RC debugging techniques that relate to performance analysis in RC, and limited research in RC performance analysis and RC tool portability. Chapter 3 then details the challenges, techniques, and tradeoffs of performance analysis in RC, proposing the initial ReCAP framework based on this research and providing a case study to demonstrate the framework's utility on an RC application (Phase 1). Next, Chapter 4 covers challenges and techniques associated with model-appropriate visualization of RC performance data and performance exploration using runtime performance data, including details of a prototype visualization for rapid, intuitive understanding of RC application behavior (Phase 2). Then, Chapter 5 describes an extension to the Phase 1 and 2 frameworks
that aids in the detection and optimization of common performance bottlenecks across diverse RC systems, providing the user with an easy mechanism to add support for additional RC systems and bottlenecks as needed (Phase 3). Finally, conclusions are provided in Chapter 6 with appendices containing data on RC systems (Appendix A) and RC applications (Appendix B) used in the study and evaluation of ReCAP.
CHAPTER 2
BACKGROUND AND RELATED RESEARCH

The background and related research is split into four sections. Section 2.1 presents an overview of RC systems and application design, focusing almost exclusively on FPGAs. Then, Section 2.2 details background and related research in performance analysis of parallel applications (including visualization and bottleneck detection). Next, Section 2.3 provides an overview of RC performance modeling and simulation. Finally, Section 2.4 covers background and related research specifically for FPGA performance analysis, along with related concepts from analytical and simulative FPGA performance prediction and FPGA debugging.

2.1 Reconfigurable Computing

Only a brief, related overview of the field of RC is given here; see Compton and Hauck’s superb overview of this field [4] as well as a treatment of RC for accelerating computation [16] for a more thorough treatment. RC includes a broad array of devices that allow some form of hardware programmability, in contrast to instruction-controlled devices or hard-wired application-specific integrated circuits (ASICs). While RC employs devices such as programmable array logic devices (PALs), complex programmable logic devices (CPLDs), and field-programmable object arrays (FPOAs) [17], field-programmable gate arrays (FPGAs) have become the predominant device used in RC systems [4], [16]. Modern FPGAs typically consist of a mesh of logic blocks and routing blocks, which are depicted in Figure 2-1 (see Altera’s Stratix II [18] and Xilinx’s Virtex 4 [19] devices as well as [4] for more information). Logic blocks typically contain LUTs (lookup tables, which define the mapping between some number of inputs and one or more output bits) and state holding circuitry such as DFFs (D flip-flops) or latches. Routing blocks, such as connection and switch boxes, are used to connect logic components to each other and to I/O pins in a myriad of ways. FPGAs often
contain coarse-grained components as well, such as memories, arithmetic blocks (e.g., multipliers), embedded processors, and communications hardware.

Designers have traditionally programmed RC systems using hardware description languages (HDLs) such as VHDL and Verilog. The hardware portion of the design is described in an HDL, synthesized into a netlist (gate-level model of design) via synthesis, and then implemented into an object file via mapping, place and route, and object file generation. The software portion of the application that interacts with the FPGA is written using a typical language such as C/C++ and uses an API to access the FPGA (e.g., function or method calls, memory mapped I/O). This software is then compiled and executed on the target RC system’s CPU; this software typically configures the FPGA with the generated object file (although the object file may be preloaded before software execution).

Alternatively, designers may use high-level languages (HLLs) such as C/C++ or Java to describe FPGA hardware. The HLL must then be translated into an HDL or netlist by the compiler and may then follow the standard flow given in the previous
paragraph; this process is commonly referred to as high-level synthesis (HLS). HLLs are an attractive alternative to HDLs since HLLs are significantly easier to use, enable faster development times, and may aid portability across platforms. In practice, these benefits may be offset by performance that is lower and more difficult to predict (when compared to a pure-HDL design) as well as by incomplete translation tools that only permit a subset of the HLL to be used [20].

In addition to the CPUs and FPGAs found in RC systems, other accelerators such as graphics processing units (GPUs), the Cell processor, and digital-signal processors (DSPs), may be employed as well. Each device provides a tradeoff between metrics such as speed, cost, power, ease of programming, and flexibility; see [6], [7], [8], [9], [21], [22] for comparisons between these devices. In contrast to these accelerators, FPGAs permit a programmer to customize hardware directly for a given application, potentially yielding greater speed and efficiency (in terms of resources employed by the application as well as power consumption). While ASICs also permit hardware to be customized and perform even better in terms of speed and efficiency, ASICs are fixed after fabrication and incur prohibitive, non-recurring costs. Due to these costs, many ASIC designs target 130nm or larger process nodes, eliminating any advantage these ASICs might have in terms of speed or efficiency with respect to current 40nm FPGAs [22]. Thus, when compared to other accelerators, FPGAs are well-suited for general-purpose application acceleration due to their ability to provide fast, efficient, customized logic for a given task while simultaneously retaining significant flexibility (in that the device may be reconfigured within milliseconds to accelerate another application).

Working in a larger, heterogeneous system, FPGAs can assist CPUs as a coprocessor, act as peers with other CPUs, or even as a master depending on the system architecture and application. FPGAs may be coupled within the system at various points including inside the CPU (as a functional unit), in a system-on-chip (SoC),
in a proprietary coprocessor, in a standalone CPU-socket, in a peripheral connection such as PCI-X/e, or as a standalone unit connected via external I/O such as Ethernet or USB [4]. A tighter coupling usually implies a lower latency to access the device, which may in turn allow the FPGA to be used for finer-grained tasks; tasks accelerated by loosely coupled FPGAs are typically coarse-grained in order to compensate for increased latency (e.g., given a 1 ms latency between the CPU and FPGA, the FPGA is unlikely to accelerate a simple addition or multiplication since the CPU could compute this itself in far less time).

In summary, RC incorporates a large number of devices, languages, and system architectures that have the potential to significantly accelerate applications while simultaneously using less power and retaining flexibility (when compared to traditional microprocessor-only systems), permitting the acceleration of a wide range of tasks. This flexibility is concomitantly the source of strength and weakness for RC devices. Flexibility allows custom hardware to be employed for the task at hand, typically improving performance and reducing power usage by removing the overhead associated with the fetch, decode, and management of instructions as well as with the memory hierarchy that is ubiquitous in microprocessors. Unfortunately, this flexibility also corresponds to increased difficulty in programming, debugging, and analyzing the performance of RC applications within these devices.

2.2 Performance Analysis of Parallel Applications

A “performance bottleneck” refers to a (proper) subset of components in a system that adversely affects system performance as a whole (the term “bottleneck” is taken from the notion of a bottle’s neck that restricts the flow of liquid from the larger bottle through a smaller opening). Thus, the goal of performance analysis is to understand a program’s runtime behavior on a given system in order to locate and remedy performance bottlenecks. Concepts for optimization, such as making the common case
fast (often derived from Amdahl's law) and focusing on bottlenecks in the application’s critical path, motivate assorted questions in performance analysis such as the following:

- Where does an application spend most of its time?
- Where and why does an application get delayed?
- What resources are used heavily by an application?
- What effect would an optimization of one portion of an application have on the entire application?

Designers may take a manual approach to performance analysis, inserting timing routines and print statements into their code to locate problems and find additional information needed to remedy the problem. However, given the tedious and error prone techniques involved in gaining access to, recording, storing, analyzing, and presenting a typically vast amount of data related to application performance, performance analysis tools were developed to facilitate this process; these tools assist a user in obtaining and presenting performance data recorded at runtime from the actual execution of an application on a given system (hereafter, “performance analysis” refers to tool-assisted performance analysis rather than manual performance analysis).

For parallel-computing applications, Maloney's TAU framework [23] provides a good introduction to the various challenges and techniques in performance analysis. As shown in Figure 2-2, performance analysis may be decomposed into stages including gaining access to application data (instrumentation), recording and storing that data at runtime (measurement), optionally analyzing recorded data for performance problems (automated analysis), visualizing performance data and analysis results (presentation) in order to allow the designer to carry on further analyses (manual analysis), and finally strategizing and implementing changes within the application in order to ameliorate located performance bottlenecks (optimization). These steps may be repeated until desired performance is achieved or no further performance gains seem likely.
A number of performance analysis tools for parallel programs have been developed with a variety of purposes and features. Tools may be simple MPI (message-passing interface) library wrappers such as mpiP [24], visualization suites such as Jumpshot [25] and Vampir [26], or full-fledged performance tools supporting multiple languages, programming models, methods of instrumentation and measurement, and automated analyses (e.g., IBM High-Performance Computing Toolkit [27] and High-Productivity Computing Systems Toolkit [28], Intel Trace Analyzer and Collector Tool (based on Vampir) [29], Paradyn [30], Paraver [31], PPW [14], [15], Scalasca (formerly known as KOJAK) [32], SCALEA [33], SvPablo [34], TAU [23]). Chung et al.’s recent study of performance analysis tools on the Blue Gene/L provides a comparison of some of these tools [35]. These tools provide a wealth of concepts and techniques that are leveraged throughout this research. ReCAP extends PPW by adding support for RC applications, leveraging PPW’s support for traditional parallel applications using CPUs.

2.2.1 Visualization

Performance visualization seeks to rapidly convey application behavior to a user. Due to a lack of previous work in visualization for RC applications, we employ research in visualization for traditional CPU-based parallel applications, which suggests that visualizations should be concise, easy-to-use, informative, appropriate to the programming model, scalable, and interactive [36], [37]. Performance data may be
presented in a number of ways, including text-based methods such as data tables as well as more common graphical forms such as charts, graphs, timeline views, Kiviat diagrams, hypercubes, animations, etc. [37], [38] Examples of visualization methodologies and tools can be found in HPC Toolkit’s HPCView [39], the IBM High Performance Computing Toolkit’s PeekPerf [40], Intel’s Trace Analyzer and Collector [29], Jumpshot [25], Parodyn [30], Paraver [31], PPW [14], [15], Scalasca’s CUBE [32], SCALEA [33], SvPablo [34], and TAU’s Paraprof and PerfExplorer [23].

Goals such as simplicity and scalability have received significant study. Knupfer et al. discuss problems associated with the size of generated trace data, problems associated with visualizing this amount of data, and techniques to reduce the amount of visualized data by exploiting patterns found and highlighting any deviation from these patterns [41]. Hackstadt et al. focus on challenges and techniques of scalability of timeline views, demonstrating the use of 3-D visualizations [42]. Reed et al. present techniques and associated benefits of using virtual reality to present an ever increasing amount of data in a sensible fashion [43].

Visualization research pertaining to “appropriateness to programming model” has traditionally received less attention in HPC due to MPI’s dominance in the parallel programming community, although the advent and spread of newer parallel languages, such as partitioned global address space (PGAS) languages, has been coupled with new visualizations for those languages [14], [15]. Nonetheless, visualization can often benefit from a user’s high-level model of the application that is not present in (or not easily extracted from) source code, using this model to structure performance data in an intuitive, application-specific context (and potentially reducing instrumentation overhead as well) [44], [45], [46]. Specifically, Sefika et al. purport the idea of architecture-aware visualization to aid in the design of complex software systems (they specifically focus on the design of the Choices operating system), where “architecture” here refers to the high-level conceptual organization of software [45]. Their approach involves
instrumenting the application and visualizing performance data in terms of the high-level subsystems designers envisioned during formulation of the system (e.g., file system, device system, virtual memory system), before these were translated to many lower-level classes, objects, and methods that implement these systems. They demonstrate that by visualizing the performance in terms of the application as the designer wrote and envisioned it, such an approach is immediately more intuitive to the designer and facilitates analyses that are not possible when viewing the design as a disparate set of classes, objects, and methods. García et al. employ this concept in the embedded realm by describing instrumentation and visualization in terms of a behavioral model of the application to gain additional insight into concurrent execution inherent in these types of systems [44]. In both cases, a designer’s model of the system is used to both guide instrumentation and to visualize performance data. This permits more targeted, low-overhead instrumentation and allows visualizations to provide an intuitive view of overall system performance that matches the programming model in use. From this vantage point, a designer can view problems areas from a high-level, delve deeper into component details to find additional information within a hierarchy, and then quickly connect their findings back to the high-level abstraction of their design, significantly streamlining the optimization process.

2.2.2 Knowledge-Based Bottleneck Detection

Knowledge-based bottleneck detection is a form of automated analysis designed to locate and describe common performance bottlenecks in an application based upon specific knowledge about where and how bottlenecks may occur. The goal of knowledge-based bottleneck detection is to reduce both the effort and expertise required to optimize an application, accelerating the optimization process; for a thorough overview of various frameworks and tools for automatic analysis, see [47]. Without bottleneck detection, the designer must understand the intricacies involving where bottlenecks can occur, understand how to detect each bottleneck, instruct the
tool to monitor relevant data, interpret the performance data to locate bottlenecks, understand what optimizations may be effective against each bottleneck, and determine the expected performance improvement if a given bottleneck were remedied (in order to ascertain whether the bottleneck is worth remedying).

While no direct research exists for enumerating or categorizing RC bottlenecks, nor has any work demonstrated automatic bottleneck detection for RC applications, DeHon et al. presented numerous “design patterns” for the purpose of improving application efficiency on reconfigurable systems; these design patterns could be included as optimization suggestions for relevant bottlenecks. For traditional HPC systems (i.e., systems without FPGAs), Mohr and Wolf argue for the necessity of automatic detection and categorization of bottlenecks when analyzing the performance of an application, which they implement in the KOJAK performance tool (now called Scalasca) [48]. This tree-based categorization breaks up total time spent into execution and idle time, with execution further broken down by language used (useful for multi-language applications such as MPI/OpenMP hybrid applications). Each language is then subdivided into communication, I/O, and synchronization related bottlenecks, which are then further subdivided; for example, the communication category is divided into subcategories such as collective operations (e.g., late broadcast, early reduce, all-to-all) and point-to-point operations (e.g., late sender/receiver). Jorba et al. discuss another knowledge-based bottleneck detection tool, KappaPI 2, that employs an XML format to store their knowledge base of bottleneck definitions, allowing the tool to locate patterns representing performance problems in trace data collected from message-passing applications [49]. Static code analysis is used in conjunction with a specification of bottleneck “instances” (also in XML format) to indicate the cause or conditions of each bottleneck and to provides hints as to how the bottleneck may be remedied. Su et al. provide details of their automated analysis framework within Parallel Performance Wizard (PPW) [47]. Their tool provides automated, model-independent detection as
As distributed analysis capabilities to reduce analysis time. Truong and Fahringer provide a very detailed scheme for performance overhead classification in SCALEA, a profile- and trace-based performance analysis tool that allows user specification of metrics and code regions of interest [33]. Their classification for performance bottlenecks includes data movement, synchronization, control of parallelism, additional computation, and loss of parallelism, each of which are further subdivided to better specify the specific reason for the bottleneck (e.g., the “data movement” category subdivides into numerous subcategories including point-to-point and collective communication, remote get/put, memory accesses including cache behavior and page faults, and both local and remote file I/O). Chung et al. detail their framework and tool for automatic bottleneck detection using an extensible, rule-based system and performance data obtained from the IBM High-Performance Computing Toolkit [50]. “Rules” are created using “metrics” (both of which may be user-defined) which in turn can depend on parameters such as the target system, enabling the tool to provide detailed information for all bottlenecks detected including expected performance improvement if the bottleneck was removed. However, there are a number of differences in RC systems and applications that must be addressed.

### 2.3 FPGA Performance Modeling and Simulation

FPGA performance analysis should not be confused with analytical modeling or simulation, which provide estimates of application performance that must eventually be verified against actual runtime performance; performance analysis is essential for capturing actual application behavior on a target system for the purpose of optimization. Nonetheless, performance modeling and simulation (both of which are used in the more general context of design-space exploration, or DSE) are vital for predicting application performance. DSE employs analytical modeling or simulation to strategically explore alternative application designs or system architectures based on metrics such as performance, power, and resources. As generally no source code exists, the user...
supplies parameters or models of application or system behavior to predict performance. Balsamo et al. provide a good survey of performance-driven modeling techniques for software applications [51]. Process algebras and calculi can also be used to predict performance, from which some concepts are leveraged in ReCAP's performance exploration framework [52].

Holland et al. proposed an analytical model called the RC amenability test (RAT) for quickly estimating the speedup of an RC application over a software baseline using various parameters including system bandwidth, read/write efficiency, input and output size, and FPGA clock frequency [53]. Smith et al. present an analytical model for a network of shared, heterogeneous workstations each containing RC hardware, including many parameters for predicting load-imbalances and shared contention on nodes as well as the communication and computation parameters of the system and application itself [54]. Such techniques have been shown to be very useful for early estimates of application performance before application development has even begun. However, analytical models may leave out many details that can significantly effect performance on an actual system, may be inaccurate if parameters (or even interactions between parameters) are mis-predicted, and are generally unhelpful in explaining why an application performed differently than expected.

Simulation-based methods for performance analysis typically provide higher fidelity than analytical methods since less detail is abstracted away. While simulation is also very useful for estimating application performance before investing significant resources into RC systems and application development, simulation must generally balance accuracy and speed; extremely accurate simulations can be orders of magnitude slower than actual execution. In addition, building a simulator is a significant undertaking in itself. Reardon et al. developed an environment for simulating RC application performance on an RC system [55]. The application is characterized by a script while the RC system is typically built from pre-defined components (e.g., network
switches, CPUs, FPGAs). An accuracy of under 7% was reported for simulations that executed reasonably fast (roughly 12 to 14 seconds) for a synthetic-aperture radar (SAR) application, although this accuracy depends in-turn on the accuracy of the RC-application script model. Densmore et al. presented an approach for automatically extracting performance information from an IP core library on an embedded RC system and incorporating this data into the METROPOLIS simulation environment for more accurate simulation [56]. However, this work assumed that the design made substantial use of these IP cores. Bondalapati and Prasanna provide details of their DRIVE framework for simulation and visualization of dynamically reconfigurable systems [46]. High-level parameterized models for reconfigurable hardware are provided by the user, which are then used to analyze expected application performance and produce visualizations. Unfortunately, even with accurate simulations of an FPGA, it is generally not possible to model the entire system at a high-level of accuracy (e.g., modeling all CPUs, communication hardware, memories) due to performance constraints, a lack of high-fidelity models for each component, or a lack of interactivity between different simulators for each component. Without an inclusive simulation of the system, the behavior and performance of the simulated application may differ significantly from the actual application.

2.4 FPGA Performance Analysis

While 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, simple measurement within a single FPGA [57]. 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. [58]. However, their work is directed at monitoring software behavior from hardware, rather than monitoring hardware itself. Similarly, Kirschbaum et al. detail the HarMonIC (Hardware Monitor for
Interprocess Communication), but their focus is on monitoring bus communication in embedded environments using an FPGA, rather than monitoring an application on the FPGA itself [59].

The field of RC debugging provides some overlapping techniques for instrumentation and measurement of application data. Unfortunately, this overlap is limited by fundamental differences in their respective purposes. For example, debug techniques such as breakpointing and FPGA readback are useful for reading (and even controlling) the entire state of the FPGA. Camera et al. propose these techniques in the BORPH (Berkeley Operating system for ReProgrammable Hardware) [60] while Wheeler et al. propose a fairly portable approach to readback and control of the FPGA using a custom scan-chain [61]. Bellows also discusses similar ideas, including clock-stepping and transparent access to memories, in the SLAAC (System Level Applications of Adaptive Computing) project [62]. Unfortunately, these techniques must “pause” the FPGA application’s clock in order to retrieve data, effectively isolating the FPGA from the rest of the system, which typically cannot be paused in unison. 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 [63] and Xilinx’s ChipScope [64] 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 some 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.
Finally, a brief overview of portability issues for RC performance analysis is discussed. Due to the lack of standardized APIs for RC systems, tool designers must either attempt to support a myriad of platform APIs or provide the user with a mechanism to add support for their own platform. Due to the difficulties in adding and maintaining support for, or even gaining access to, each new platform and API version, the first approach will typically ensure a limited number of supported platforms and thus limited tool applicability. The latter approach is taken by some HLS tools, such as ImpulseC’s platform-support packages [65] or ROCCC’s platform interface abstraction layer [66], as well as by frameworks supporting generic communication between heterogeneous devices, such as Auto-Pipe [67] or the System-Level Coordination Framework (SCF) [68]. Unfortunately, for performance analysis (and specifically bottleneck detection), additional information is required beyond what is necessary for portability (e.g., bytes transferred or expected bandwidth for each transfer type). In addition, adding platform support may require significant expertise and effort (e.g., both ImpulseC and ROCCC require a manual conversion between a platform’s API and their specified interface), limiting the potential for widespread use.
CHAPTER 3
PERFORMANCE ANALYSIS FRAMEWORK FOR RECONFIGURABLE-COMPUTING
APPLICATIONS (PHASE 1)

In this phase of research, we explore the challenges faced in attaining low-overhead, automatable techniques for instrumentation and runtime measurement of RC applications as well as for visualization of obtained performance data. While significant challenges exist in each of the five stages of performance analysis, the instrumentation, measurement, and to a lesser extent presentation stages form the core of performance analysis that is built upon by more advanced visualization, automated analysis, and optimization. Thus, this phase focuses on challenges associated with instrumentation and measurement while briefly discussing concepts towards presentation and a unified performance analysis tool. We also discuss techniques to address these challenges and present the details of the ReCAP framework that provides the core methodology for RC performance analysis using these techniques. We then integrate this framework into an existing parallel performance analysis tool, Parallel Performance Wizard (PPW) [14], [15], and provide results from a case study to demonstrate the utility and minimal overhead of our ReCAP tool.

3.1 Challenges for RC Performance Analysis

Within instrumentation and measurement, the key goals of performance analysis tools are the following (adapted from [23]):

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).

3.1.1 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.

3.1.1.1 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 where and how performance bottlenecks can occur 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. Off-chip communication (i.e., off-board and on-board communication) 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 control and communication), demonstrating the wide applicability of these concepts 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.

3.1.1.2 Levels of instrumentation

Before reaching the challenge of modifying an application for measurement, 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 the FPGA configuration file (binary loaded directly onto the FPGA). While it is also possible to use system-level instrumentation (e.g., OWL [58] discussed in Section 2.4), 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 [69]. 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 3-1 provides a summary of the comparison between source and binary instrumentation.

Table 3-1. Comparison of source and binary instrumentation.

<table>
<thead>
<tr>
<th></th>
<th>Source-level</th>
<th>Binary-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Difficulty</td>
<td>Text parsing</td>
<td>Bit-level signal routing</td>
</tr>
<tr>
<td>Design perturbation</td>
<td>Low change in area &amp; speed</td>
<td>Low change in on-chip physical layout</td>
</tr>
<tr>
<td>Time to instrument</td>
<td>Long (hours)</td>
<td>Short (minutes)</td>
</tr>
<tr>
<td>Portability</td>
<td>Good across devices</td>
<td>Good across languages</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Access to all signals</td>
<td>Some data inaccessible</td>
</tr>
<tr>
<td>Source correlation</td>
<td>Possible</td>
<td>Generally not possible</td>
</tr>
</tbody>
</table>

3.1.1.3 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 3.1.1.1) 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 HLL or HDL portion of an 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. These issues are addressed in Section 5.
Source-level instrumentation for hardware can employ a parser 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 parser 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 component. After synthesis and implementation, tools such as Xilinx’s JBits SDK [70] can be used to place the measurement component in the device and route signals to it from the application.

3.1.2 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.
3.1.2.1 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 [71] 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 [72]. 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 [72], 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 FPGA configuration 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.

3.1.2.2 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.
3.1.3 Challenges for Performance Presentation

Conventional trace-based display tools such as Jumpshot [25] 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 3-1. 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 3-1. Mockup Jumpshot visualization of an RC application with 8 CPUs and 8 FPGAs.](image)

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 [41]. They propose detecting repetitive patterns and collapsing these patterns visually into a single box, thus highlighting irregular behavior in the application to increase scalability. These problems 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 execute 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 3-1) 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 3-2 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.

From a practical standpoint, obtaining the communication and computation shown in Figure 3-2 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 performed 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 3-2 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.

Figure 3-2. Example hierarchical display of a complex RC application.
3.1.4 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) [14], [15] as a specific software performance analysis tool to discuss integration here, although these concepts should apply to other tools as well. PPW supports performance analysis for PGAS programming models such as UPC and SHMEM as well as for message-passing models such as MPI. PGAS performance analysis is enabled by way of the Global Address Space Performance (GASP) interface [73], 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.

### 3.2 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.

Figure 3-3. Additions made by source-level instrumentation of an RC application.
### 3.2.1 Instrumentation

Figure 3-3 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 HDL code directly via a standard VHDL grammar and parser. The user’s HDL code is then partially elaborated to gain information such as number of iterations in a VHDL “for-generate” statement or whether a component within a VHDL “if-generate” statement is active, permitting more accurate and targeted instrumentation (e.g., by instrumenting only one of four application cores on the FPGA).

2. An automated selection of data is made based on a desire to monitor communication and computation. For example, all “case” or “if-else” statements (or both) 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 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 3-4a). Data types are converted to \texttt{std_logic_vector} whenever possible for simplicity of monitoring (e.g., enumerated types are converted to \texttt{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., \texttt{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 address 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. In order to increase automation, ReCAP v0.7 and later eliminate this new top-level file by making all changes directly to a copy of the original top-level file and performing name-shifting on signals connected to the input and output ports. This methodology ensures the user no longer needs to modify their hardware project to include additional files.

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. In contrast, ReCAP v0.7 and later perform any analysis or combination of signals as close to their origin as possible in the hierarchy, passing up a single-bit event line from that point in the hierarchy; this methodology allows for lower resource overhead when employing event pipelining (a technique where events may be registered several times to help reduce or eliminate a drop in the application’s maximum frequency when instrumented).

6. In software, a Hardware Data Transfer Module (HDTM) is added that will start 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. In order to reduce overhead and support systems without threads, ReCAP v0.7 and later provide (and default to) post-mortem-only collection; threaded data collection is typically only desirable when trace data will exhaust FPGA memory resources, thus justifying periodic data collection to reduce or eliminate dropped trace records.

7. If threads are employed for the HDTM, 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 HDTM (this and previous step are illustrated in Figure 3-4b).

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
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 3-4. Instrumentation of user source code.

3.2.2 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 3-5 illustrates the design of the HMM.

Figure 3-5. Hardware Measurement Module.

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 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.

### 3.3 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 3.2. In our prototype version, the instrumentation steps are performed manually (Section 3.2.1), 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) [11]. 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 [74] 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 [75]. 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 \( 16 \times 16 \). 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 3-2 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.

Table 3-2. Performance Analysis Overhead.

(a) Cray XD1

<table>
<thead>
<tr>
<th>Design Aspect</th>
<th>Orig.</th>
<th>Device %</th>
<th>Inst.</th>
<th>Device %</th>
<th>Diff.</th>
<th>Diff. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices (23616 total)</td>
<td>9041</td>
<td>38.3%</td>
<td>9901</td>
<td>41.9%</td>
<td>+860</td>
<td>3.7%</td>
</tr>
<tr>
<td>Block RAM (232 total)</td>
<td>11</td>
<td>4.7%</td>
<td>15</td>
<td>6.5%</td>
<td>+4</td>
<td>1.7%</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>124</td>
<td></td>
<td>123</td>
<td></td>
<td>-1</td>
<td>-0.8%</td>
</tr>
<tr>
<td>Communication (B/s)</td>
<td>80</td>
<td></td>
<td>33290</td>
<td></td>
<td>+33210</td>
<td></td>
</tr>
</tbody>
</table>

(b) Nallatech Cluster

<table>
<thead>
<tr>
<th>Design Aspect</th>
<th>Orig.</th>
<th>Device %</th>
<th>Inst.</th>
<th>Device %</th>
<th>Diff.</th>
<th>Diff. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices (49152 total)</td>
<td>23086</td>
<td>47.0%</td>
<td>26218</td>
<td>53.3%</td>
<td>+3132</td>
<td>6.4%</td>
</tr>
<tr>
<td>Block RAM (240 total)</td>
<td>21</td>
<td>8.8%</td>
<td>22</td>
<td>9.2%</td>
<td>+1</td>
<td>0.4%</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>101</td>
<td></td>
<td>101</td>
<td></td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>Communication (B/s)</td>
<td>40</td>
<td></td>
<td>29860</td>
<td></td>
<td>+29820</td>
<td></td>
</tr>
</tbody>
</table>

53
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).

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 (technically, one of the profile counters is unnecessary since its value can be ascertained from the other counters and the total cycle count). From this data, the percentage of cycles spent in each state was calculated and is shown in Figure 3-6. 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 3-6 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 states, 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.

![Figure 3-6. Distribution of cycles spent in core state machines of N-Queens.](image)

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 3-7 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.
3.4 Conclusions

In the first phase of this research, 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, and tradeoffs between accurate measurement and overhead incurred 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 presented 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.
CHAPTER 4
PERFORMANCE VISUALIZATION AND EXPLORATION (PHASE 2)

In the first phase of this research, basic visualizations already available in traditional HPC performance tools were used to provide quick graphical understanding of FPGA performance data, and potential concepts for future work in RC visualization were presented. These visualizations are essential in quickly conveying complex application behavior at runtime, allowing an application designer to rapidly locate and address performance problems in their application. As mentioned in Phase 1, traditional HPC visualizations are rooted in a homogeneous-parallel-thread model wherein all threads execute on the same (or similar) hardware, all threads are generally assumed to be at the same level in the application hierarchy, and where a single thread is considered to be sequential in nature. Unfortunately, these assumptions do not hold in RC systems with heterogeneous devices, special-purpose components, multiple hierarchies of parallelism, and parallelism even within the same component (e.g., components may perform computation and communication simultaneously). In attempting to convert this disparate model to the homogeneous-parallel-thread model, a significant amount of information is neither captured nor presented intuitively to the user.

Figure 4-1 demonstrates the difficulties associated with presenting RC data in a traditional timeline view using Jumpshot. Assuming this visualization depicts eight CPUs and boxes for each CPU represent different functions, this visualization can be of use. However, assuming this visualization depicts eight different RC components and boxes represent work, overhead, and so forth, the results are far less clear since the visualization leaves the purpose and relationship between different lines unspecified. Unfortunately, the number of components inside even one FPGA for a typical RC design may exceed the scalability of such a figure; in a multi-CPU/FPGA system, this figure would only become worse. Thus, this phase of research seeks to extend the framework
and tool detailed in Phase 1 to provide RC performance visualization that is appropriate to the heterogeneous software/hardware environment of RC systems and applications.

Figure 4-1. Traditional timeline visualization (Jumpshot) example demonstrating difficulties associated with presenting RC data

Additionally, even given an intuitive RC performance visualization where bottlenecks can be quickly located and optimizations formulated, a user is still left with the challenge of predicting the expected performance improvement for each optimization, weighing this improvement against other expected costs (e.g., implementation effort, power, resources used). If the prediction is inaccurate, a significant amount of implementation and reverification effort may be wasted. While a significant amount of work has been performed in RC performance modeling within DSE, this literature typically makes significant assumptions about the system and application that can reduce accuracy and are unnecessary now that an implementation for a specific system has been realized. Rather, we present a performance exploration methodology that employs runtime performance data to capture application behavior, allowing our framework to provide accurate predictions of application performance if optimizations were made. Thus, in this phase of research, we also extend our Phase 1 framework to provide RC performance exploration (in addition to the aforementioned visualization framework and tool), culminating in an application case study that demonstrates the utility of both RC performance visualization and exploration. We first present our RC performance visualization and exploration research (including details of our prototype implementation
for RC visualization), followed by a demonstration of both visualization and exploration via an application case study.

### 4.1 RC Performance Visualization

Unfortunately, traditional parallel-processor performance visualizations are not readily suited for RC applications. First, these visualizations typically assume devices are general-purpose, interchangeable, and within a relatively shallow hierarchy (e.g., timeline visualization with one processor per row). In contrast, many components programmed on the FPGA are special-purpose (e.g., a component that scatters data to cores) and may be connected in arbitrarily deep hierarchies. If treated as a flat list of interchangeable devices, much of the structure and semantics of the application are lost. Second, even with heavily pipelined, superscalar processors executing instructions out-of-order, a single thread or process is still abstractly viewed as roughly sequential (since sequential ordering of application code is preserved), while a single FPGA component may perform an arbitrary amount of tasks in a given cycle (HDL code is inherently parallel). Finally, HDLs lack standardized, high-level communication and synchronization functions (e.g., `MPI_Send`), complicating attempts to automatically classify and visualize such behavior.

To address these issues, we follow a similar approach to [44], [45], having a user provide a high-level model of application behavior. However, to make our framework easy-to-use, we provide simple pragmas that allow a user to augment their source code with high-level information while not changing the application itself. Each pragma defines the current state of a hardware or software “block” (the finest-grained unit that operates in parallel with all other blocks, typically a software thread or process or a VHDL process block or Verilog always block); note that while the term “state” is used, no explicit state machine is needed in the application’s source code to employ our methodology. In software, pragmas are only used to classify FPGA API calls (since PPW, and therefore ReCAP, already monitors traditional parallel-program
communication and synchronization). In hardware, pragmas are used to classify how time is spent in each clocked block. Figure 4-2 shows several examples of software and hardware pragmas.

```
#pragma recap wrX send(top.in, data): words > 0
  fpgaWrite(fpga, data, addr, words);
#pragma recap waitDone wait_recv(top.out, done)
  while (fpgaReadReg(fpga, addr2) == 0);
```

A SW pragmas

```
case current_state is
  when recvXCoord =>
    --pragma recap getX recv($CPU, data)
  when waitAck =>
    --pragma recap w1 wait_recv(top.out, ack): ack='0'
    --pragma recap d1 recv(top.out, ack): ack='1'
```

B HW pragmas

Figure 4-2. Example of user-defined pragmas

Each pragma-defined state is given a name and one or more classifications such as wait_send, wait_recv, send, recv, and busy (performing internal tasks), although other categories such as wait_sync and overhead could be useful as well (the chosen categories are similar to those visualized in current performance tools [32] and used for performance modeling [52]). The “busy” category requires no arguments, while “wait” and “communication” categories require two arguments specifying dependencies and a “purpose” tag used to distinguish between similarly-typed messages from the same block. While a pragma automatically inherits all conditions that contain it, an optional condition field (after the colon) permits additional control over when a block is in a given state.

Given this framework, we extended ReCAP to automatically monitor these pragma-defined states. While trace (timeline) information is ideal for viewing exact details of application behavior (and is supported by ReCAP on both CPUs and FPGAs), limitations on FPGA memory make it difficult to rely on such information. Instead, we
build a Markov model of all CPU and FPGA blocks by recording time spent in each state and state transition, accepting a potential loss in fidelity. While this approach incurs \( O(n^2) \) counters, where \( n \) is the number of states, it is common for many transitions to be impossible. Thus, we use a basic sparse-matrix representation for state transitions on both devices; due to the potentially high cost of implementing sparse matrices on FPGAs, we provide optional syntax in hardware pragmas to specify which transitions are possible, thus creating a static sparse matrix. A synthesis tool could potentially determine impossible transitions automatically, but this is difficult and may not be possible in all cases.

To present performance in an intuitive, model-appropriate fashion, we visualize performance data within the system and application hierarchy. While HDL code is naturally hierarchical, we found the digraph generated from block dependencies better suited for understanding behavior than the actual structure of code (since HDL components or modules may simply pass data through or connect two sub-components together). For our visualization, we define the “system,” “node,” “device,” “block,” and “state” levels, allowing any of these levels to have one or more nested groups within it. A “node” is generally the smallest networked homogeneous unit of a system (which may contain multiple “devices” such as CPUs and FPGAs), while the block level refers to the definition of “block” given earlier (e.g., thread, VHDL process block), which are broken into “states” via user-supplied pragmas. Examples of application- or system-specific grouping include FPGAs on the same card and threads or cores performing similar tasks. By employing a hierarchical structure, scalability is improved and our visualization is appropriate to the hierarchical structure common in RC applications. As these levels are fairly generic, our visualization and exploration framework could also apply to other computing devices (e.g., GPUs, DSPs).

In ReCAP, we prototyped this visualization (without grouping mentioned above) by generating a single file per logical CPU for use in Graphviz [76]; thus, multi-CPU/FPGA
applications are supported, but the system-level view is not yet constructed. Graphviz can then generate scalable vector graphics (SVG) files, allowing a user to open the top-level SVG file and traverse the system and application hierarchy via hyperlinked nodes. Figure 4-3 shows actual tool-generated output for the device, node, and block levels, which have been cropped for brevity. The state level (not shown) is also visualized within our tool as an HTML table including the number of iterations, user pragma, source location, tool overhead, time spent in the state (including min, max, average, and total), and similar summaries of bytes transferred and communication bandwidth. Dependency information between blocks within the FPGA or between the FPGA and CPU are also visualized (e.g., the arrow connecting the two blocks within the device level in Figure 4-3). While visualization of CPU-CPU communication and synchronization is currently unimplemented in this view, statistical and timeline views of this behavior are available in PPW+RC’s frontend GUI. Figure 4-3 will be analyzed for performance problems in Section 4.3.

The block level (for CPUs and FPGAs) shows each user-defined state’s name and type, along with the cumulative time spent in that state (numerically and visually via partially filled bars). In addition, transitions are labeled with a percentage representing the frequency of a given transition in comparison to all outgoing transitions from the state. In CPU blocks, transitions are also labeled with the total time spent transitioning from one state to another (numerically and visually), as an arbitrary amount of work may occur between FPGA API calls. The CPU block level is shown at the bottom of Figure 4-3.

At higher levels in the hierarchy, it is crucial to summarize data such that a user can quickly determine whether that item deserves further attention. From a performance analysis perspective, an application is performing ideally if all computational resources committed to the application are fully used; certainly better algorithms can have less utilization and still perform better, but performance analysis is concerned with
maximizing a given solution, assuming all computation performed is necessary. Deviation from this ideal is considered a (local) performance problem and potentially an application bottleneck (a critical performance problem degrading the entire application’s performance) and should be highlighted.
Therefore, we display total time spent in each category when summarizing blocks within the device and node levels, and we display the maximum of each category’s totals (along with the minimum total busy time) when summarizing devices and nodes themselves. The maximum wait-time, communication-time, and minimum busy-time show deviations from the ideal, while the maximum busy-time shows hot-spots that may benefit significantly from optimization (including algorithmic improvement). It may also be of use to provide standard deviation and automatic grouping or binning of blocks with other similarly performing blocks, depending on the application.

Based on our visualization approach, it is often beneficial to define as few states as necessary to describe high-level application behavior, inserting additional pragmas in key areas if more detail is needed. In addition, while hierarchical structuring has many advantages, it may also be desirable to query or aggregate performance data. For example, a query such as `SELECT * FROM FPGA[*] WHERE TIME(SEND) > 0.1*TOTAL_TIME()` could be used to find all FPGA states sending data more than 10% of the time. Due to significant implementation cost, this feature is not currently implemented.

The visualization implemented here is not intended to be a definitive visualization for RC. Additional views and data could be of use. For example, multi-node visualizations containing communication channel statistics, such as shown in Figure 4-4 (similar to Figure 3-2 shown in Section 3.1.3) and a node-level view visualizing communication, buffer, and memory statistics, such as shown in Figure 4-5, could be of significant use. Further visualization research is left for future work.

4.2 RC Performance Exploration

Once performance is visualized, the user must formulate potential optimizations and weigh expected performance gains against expected costs of optimization (e.g., effort, power, resources). Unfortunately, predicting performance can be difficult in complex RC applications, potentially leading to wasted implementation effort. As mentioned in
Figure 4-4. Example of a system-level visualization of an RC application executing on a 6-CPU / 3-FPGA RC system assumed to be part of a larger cluster.

Figure 4-5. Example of a node-level visualization of an RC application, showing 2 CPUs and an FPGA from the left hand side of the system-level visualization.

Section 2.3, DSE is invaluable for providing early performance estimates. However, these estimates may be inaccurate due to assumptions concerning the system and application; thus, we employ actual performance data gathered for our visualization to support performance exploration in the optimization stage. Our proposed exploration methodology compliments existing methods in DSE and performance prediction.
Since our goal is to maximize performance of an existing system and design, we take advantage of detailed performance data, targeting finer-grained changes and more accurate exploration rather than broad changes common in traditional DSE. Ideally, our data and methodology could be integrated with existing DSE tools (e.g., augmenting application and system models with real performance data), enabling more accurate prediction and providing an end-to-end environment in which to study application performance, from conceptual design through optimization.

We now motivate our approach. Figure 4-6 shows mockup performance data for an application consisting of two blocks. Each state contains the cumulative time (CT) spent in that state (over all iterations) as well as the number of incoming and outgoing transitions, from which a potential timeline can be constructed (Figure 4-7A). Supposing an optimization could reduce the CT of the busy2 state from 3s to 1s, Amdahl’s law suggests an ideal speedup of 22.9% \((1/(1 - 3s/10.75s + 3s/10.75s))\) is possible. However, in reality, the performance of other blocks could inhibit us from achieving this potential.

Assuming the average case, the above optimization will reduce each of the four iterations of busy2 by 0.5s, leaving gaps in Block B’s timeline (Figure 4-7B). Unfortunately, the first three gaps precede a wait state; since Block B was already waiting on Block A, Block B will have to wait that much longer. However, the final gap is followed by communication with Block A, where Block A is waiting for this communication. Thus, Block A could now wait less time, allowing the final send/recv pair to complete 0.5s earlier (Figure 4-7C) and providing a final speedup of 4.9%, much less than the ideal 22.9% (this ideal could be realized with a similar change to busy1 instead).

While the previous example focused on changing a state’s CT (along with the corresponding adjustments to other states), other changes can be modeled as well. For example, changing FPGA clock frequency can be modeled as scaling the total time of each FPGA state (except those communicating or waiting on external resources such
Figure 4-6. Example application for exploration.

Figure 4-7. Potential timelines when optimizing an example application.

as SRAMs or CPU). Changing a block's replication factor by $F$ (e.g., doubling CPUs or FPGA cores) can be modeled as scaling all busy and communication state times by a factor of $1/F$ (assuming perfect load balancing). Our framework does not constrain the magnitude of such changes, allowing users to investigate performance if a larger or faster device were employed; however, potential changes in resource usage and
achievable frequency should be considered when targeting specific devices. Ideally, a tool could automatically map these higher-level changes into corresponding state-level changes rather than requiring the user to manually perform this mapping.

This approach makes several assumptions. Since a generated timeline represents the average case, it could differ significantly from actual runtime behavior (especially with shared effects such as contention). However, a number of RC applications exhibit very regular behavior, and thus our approach is reasonable given the significant memory reduction afforded by using summary statistics (even trace data can be insufficient to predict performance as a single execution may not include all execution paths). Also, although a user’s change to the CT of one or more states could affect the CT of a busy or communication state, we assume only the CT of wait states is adjusted, as this case is far more common and easier to model. Finally, although matching communication states between blocks can wait on each other (e.g., in one iteration Block A waits on Block B while in the next Block B waits on Block A), we assume that communication can be shifted such that only one block is waiting on another; while not always possible, this represents an ideal case if such a synchronization problem is remedied. We show in Section 4.3 that our methodology can still be fairly accurate, even with non-uniform behavior.

Unfortunately, generating timelines consistent across all blocks can be difficult, as matching send/recv pairs must be aligned globally (e.g., in Figure 4-7A, Block B’s first wait state was elongated in order to line up communication states). Thus, for simplicity and efficiency, we forego timeline generation, instead employing a heuristic methodology that calculates the CST, or cumulative time a state is shifted in the timeline (e.g., the final -0.5s shift in Figure 4-7C), as well as the CT for each state; pseudocode for our approach is provided in Figures 4-8A and 4-8B. Note that while CST/CT calculation may yield different results from a timeline approach (as the latter ensures matching transfers occur together rather than just ensuring that dependencies, on average, are met), this
does not imply the timeline approach is more accurate; both are effectively assuming a schedule that may differ from actual execution.

```java
void propBlk(State s, double CST) {
    if (CST == 0) return;
    s.CST += CST;
    // add dependencies to queue
    foreach (dep in s.dependencies) depQueue.add(dep);
    // compute next state changes
    double total = sum(s.outTrans);
    foreach (n in s.nextStates) {
        double frac = s.outTrans(n) / total;
        if (n.visited)
            handleCycle(n, s, frac * s.CST);
        else propBlk(n, frac * s.CST);
    }
}
```

**A Block-propagation methodology**

```java
void handleDep(State s, State sw, State d, State dw) {
    double CST, CT;
    // modify wait times
    CST = max(-dw.CT, s.CST - d.CST);
    CT = min(CST, max(CST - sw.CT, 0));
    dw.CT += CT;
    sw.CT += CT - (s.CST - d.CST);
    // propagate and back-propagate
    propBlk(dw, CT);
    propBlk(sw, CT - (s.CST - d.CST));
}
```

**B Dependency-handling methodology**

Figure 4-8. Performance exploration methodology.

We now re-predict performance for our example application via the CST/CT methodology. As shown in Figure 4-9A, the user changes the CT of busy2 from 3s to 1s (checkered). The propBlk function (Figure 4-8A) is then called with the busy2 state
and -2s CST. This function is responsible for propagating a change in expected CST throughout a block, adding any dependencies to a queue for later processing. The CST is split among all potential next states based on the frequency of transitions to each. If a state has been visited before, the `handleCycle` function (definition not shown) finds all states outside the cycle that have a transition to them from within the cycle (exit states) and determines the correct split of CST between these states (determined either by iteration or geometric series if the number of iterations is sufficiently large). In our example application, the call to `propBlk` adds dependencies for `recv2` and `send2` to the queue and results in the final CST values shown (black ellipses) in Figure 4-9A.

At this point, each dependency is incrementally applied by calling `handleDep` with the four relevant states in the dependency: the source `s`, its corresponding wait state `sw`, the destination `d`, and its corresponding wait state `dw` (Figure 4-8B). If there is no corresponding wait state for either `s` or `d`, an implicit one with zero CT is added. The `handleDep` function attempts to shift communication as early as possible, resolving any conflict detected by increasing wait time at the source. Figure 4-9B shows the result of triggering the `recv2` dependency, which resolves a conflict where Block B could progress faster but Block A cannot, thus increasing the wait time in `wait2` to 4.25s. Figure 4-9C then shows the final outcome after triggering the `send2` dependency, which allows the `recv1` state in Block A to shift 0.5s earlier (by reducing the CT of `wait1` to 0.25s), yielding a final reduction of 0.5s out of 10.75s (4.9% speedup) for the application, as predicted earlier. Note that our methodology not only predicts overall performance, but also the CT for each state, thus aiding a user in locating and potentially remedying bottlenecks in an optimized version of the application before the optimization has been implemented.

While our intent is to allow users to modify a state’s CT, block replication, or frequency within our visualization, performance exploration is not currently implemented.
Figure 4-9. Performance exploration using CST/CT updates in ReCAP. Thus, we manually employ our exploration framework to demonstrate its utility.
4.3 Results

To validate the utility of our framework, we investigate the performance of a Collatz application, which tests the lengths of sequences generated under repetitive iteration of a simple function on the natural numbers (see [77] for details). While this application is not readily used for practical purposes, it contains patterns that closely resemble cryptanalysis (i.e., a large number space is pre-filtered by a CPU, the FPGA performs a large parallel search, and finally the CPU collects and performs post-processing on returned numbers) [78], [79]. In addition, both the CPU and FPGA are involved in computation, with FPGA cores requiring an unknown number of cycles to complete, thus providing a case study that would be difficult to predict performance for via other methods.

The application was executed on a Pentium-4 3.2GHz 64-bit Xeon processor containing a Nallatech H101-X PCI-X card with a Virtex-4 LX100. We used GCC 4.4.2 with “O3” optimization and Xilinx ISE 11.3 with default settings to compile the application software and hardware, respectively; the same application served as a C baseline (with FPGA tasks performed on the CPU instead). All execution times were computed from the average of three executions, with the FPGA operating at 100MHz. Each execution consisted of running sequence-length tests on the first 154 billion numbers.

Figure 4-3 shows visualizations generated by ReCAP for an initial 2-core version of the Collatz application. From the visualizations, we quickly noticed the CPU is waiting a significant amount of time for the FPGA to complete its work (100.8s, as shown in the CPU-block-level visualization), representing poor hardware/software partitioning that could be remedied easily by increasing the number of cores in the FPGA. Assuming the wait-time problem can be removed, the instatus1 state is taking up a significant portion of the remaining time (13.5%). Decreasing the transmission frequency (and correspondingly increasing buffer size) could reduce this overhead. In support of this decision, clicking on the send1 state indicates an average bandwidth of 42.5MB/s and
message size of 0.75KB (not shown); platform benchmarks indicated that increasing message size could improve bandwidth significantly.

We now predict the effects of scaling the number of FPGA cores in our application from 2 to 56 via our performance exploration methodology (64 cores could not be validated due to FPGA resource limitations). Figure 4-10 shows the ideal prediction given by Amdahl's law (triangles), our methodology’s prediction (squares and dotted line), actual application performance (asterisk), and performance achieved from an optimized version of the application discussed below (circles). Since FPGA computation represents the majority of the application time, Amdahl's law predicts near-linear speedup, with a 56-core version achieving a total 54.2x speedup over the software baseline. However, our performance exploration predicts that performance will scale in a roughly linear fashion until leveling off around 12 cores, with the 56-core version achieving a total 12.1x speedup over the software baseline.

![Figure 4-10. Performance exploration for Collatz application.](image)

Our actual 56-core version achieved a total 11.7x speedup over the software baseline, resulting in a maximum error in predicted performance of only 3.3% for our performance exploration methodology (using Amdahl's law yields 361% error), thus
demonstrating our framework’s utility, even in the presence of non-uniform behavior.
As examining the predicted state CTs indicated the FPGA would be mostly idle if 56-cores were employed, we developed an auto-tuning step to perform coarse-grained load-balancing between the CPU and FPGA and reduced transmission frequency (as mentioned earlier), yielding a 3.8x speedup over the original 56-core design (44.7x speedup over the software baseline), as shown in Figure 4-10.

Due to the number of application resources used (85% LUTs, 32% regs, and 57% block RAMs (BRAMs)), we chose to instrument only 2 of the 56 hardware cores (all other software and hardware were instrumented normally). We observed a change of +1.7% (or less) in software runtime, -2% LUTs, +4% regs, +35% BRAMs, and no change in maximum frequency. The unexpected changes to LUTs and BRAMs were due to instrumentation interfering with a RAM-packing synthesis optimization that decreases BRAMs at the expense of LUTs. Unfortunately, ReCAP replicates components in loops to ease instrumentation, preventing ISE from detecting this optimization for this application; better handling of loops in ReCAP would avoid this issue. Disabling this optimization yields a more informative baseline of 77% LUTs, 32% regs, and 92% BRAMs for the original application, yielding an actual overhead of +6% LUTs, +4% regs, and +0% BRAMs.

### 4.4 Conclusions

In conclusion, this phase provided a methodology for both performance visualization and performance exploration of RC applications, implementing a prototype visualization by extending the ReCAP framework and tool described in Phase 1. The utility of our methodology was demonstrated by predicting application performance before actually implementing changes, yielding only 3.3% error when validated. Based on these results, a scalability problem and inefficient communication were quickly located and remedied, yielding a 3.8x speedup of the original 56-core design (increasing application speedup from 11.7 to 44.7 when compared to the software baseline).
CHAPTER 5
PLATFORM-AWARE KNOWLEDGE-BASED BOTTLENECK DETECTION
(PHASE 3)

While the extended ReCAP framework from Phase 2 provides performance analysis for RC applications, including performance visualization and exploration, the designer must still manually search for, discover, and characterize bottlenecks as well as formulate optimization strategies to address each bottleneck, requiring significant effort and expertise. However, different applications may share similar bottlenecks and thus may benefit from similar techniques to locate these bottlenecks as well as from similar suggestions on how to remedy them. Thus, knowledge-based bottleneck detection (a promising area of research in traditional HPC that attempts to locate common bottlenecks) could reduce both the expertise and effort required to optimize applications, significantly accelerating the optimization process.

Unfortunately, providing bottleneck detection for RC applications incurs additional challenges not typically present in traditional HPC. For example, while attempts have been made to standardize some aspects of RC-system APIs, such as OpenFPGA’s GenAPI [80], the continued widespread diversity of both hardware and software APIs for RC systems complicates bottleneck detection, as there are no standardized constructs such as MPI_Send (a construct within the MPI library that sends data from the calling node to another node in a system) that provide hooks for recording relevant statistics (e.g., measured bandwidth or bytes transferred). Another key challenge exists in defining, locating, reporting on, and suggesting remedies for application bottlenecks. Many approaches in traditional HPC are ill-suited for RC applications due to common assumptions that all system processing elements are relatively homogeneous, general-purpose, and at roughly the same level in the system hierarchy (i.e., applications execute on CPUs that can perform general computation and communication, individually or in groups). In contrast, components within an RC application are almost certainly heterogeneous, may be designed solely for scatter communication or pipeline control
(non-computational components), exist in fairly deep and intricate hierarchies, and may perform an arbitrary amount of work in a single cycle, thus resisting conventional bottleneck detection and classification schemes. In addition, an FPGA’s hardware flexibility significantly expands the possibilities for optimization, increasing the complexity of formulating bottleneck remedies. Finally, from a practical standpoint, trace data, which is used heavily for bottleneck detection in traditional HPC, cannot be relied on in FPGAs as these devices typically have limited memories and real-time requirements (i.e., it can be difficult to “pause” an application on an FPGA due to low-level interaction with external hardware, such as memories or other FPGAs).

Thus, in this phase of research we extend our Phase 1 and 2 framework and tool with platform-aware, knowledge-based bottleneck-detection to address the current lack of bottleneck detection available for RC applications. The presented framework and tool are extensible in that users may easily add support for their own platform, as long as it fits within our platform-template model, and may also easily modify the bottleneck knowledge base. We also present what we believe to be the first taxonomy of common RC bottlenecks, including detection and optimization strategies, which we use to populate our ReCAP tool’s knowledge base. Although this work should aid novice RC designers who may be unaware of many potential bottlenecks and optimizations, experienced RC designers can benefit as well from quick feedback on the location and severity of performance problems. We first focus on platform-templates to address bottleneck detection and portability across diverse RC platforms, followed by a discussion of our bottleneck detection framework and tool. We then present our taxonomy of common bottlenecks that ReCAP can detect and suggest optimizations for, followed by a demonstration of this research on two application case studies on a total of three diverse RC systems.
5.1 Platform Templates

Platform templates have two purposes in ReCAP: bottleneck detection and portability. In this section we detail our platform-template framework and its implementation in ReCAP. While this framework handles many common API styles in both software and hardware, we will also point out cases not currently handled as well as concepts that could potentially address these situations.

ReCAP provides a single location within the HDL Instrumenter that encapsulates all platform-specific information, permitting a typical user to quickly add support for their own platform; this information is divided into several software and hardware API tabs for easier access. ReCAP allows each platform to be given a unique name and saves all platform information in a separate file, allowing platform templates to be easily loaded and shared. In fact, ReCAP can support user APIs built on top of a platform’s API, allowing several different platform templates to coexist for the same platform; the user simply selects the corresponding template that matches the current API in use.

5.1.1 Software

In order to locate bottlenecks involving CPU-FPGA communication, or to even determine when such communication is occurring, ReCAP must know what API calls are possible as well as some auxiliary information (e.g., purpose, bytes transferred if a transfer) in order to effectively instrument these calls. Specifically, a user must enter a C/C++ prototype for a function, macro, or class method into the HDL Instrumenter. As this information can be directly obtained from the platform API’s header, which must be available on the system, the user can simply copy and paste this information into ReCAP.

---

1 Macros are currently handled by providing an equivalent function prototype; while this approach could cause problems by presupposing argument and return types, an improved implementation could avoid this issue by handling macros separately.
For each prototype supplied, the user must indicate its type (e.g., configuration, acquisition, data transfer, release). In addition, C/C++ expressions can be given to identify the FPGA number associated with this call (if any), bytes transferred (for data transfers), and filenames (for configuration). These expressions are free to access function arguments, global variables, class methods, or class public fields; for example, the FPGA number expression may simply be `fpgaNum` or `this->getFpgaId()`, assuming these represent an argument in the prototype or a public class method, respectively.

From this information, ReCAP identifies and instruments each API call in the user’s source code. Function- and macro-based API calls are overridden via macros that effectively change the name of each API call in user source code in order to instead call instrumented wrapper functions or macros, which record various applicable statistics (e.g., time spent, bytes transferred, bandwidth achieved) in addition to performing the original API call\(^2\). Class methods are instrumented by subclassing each class within a platform API, with each subclass method recording statistics before calling the corresponding base class’ method; in addition, any instantiation of a platform API class in the user’s source code is replaced with an instantiation to the instrumented subclass in a copy of the user’s source code. Unfortunately, this approach would not be sufficient for constructors and destructors due to the order in which constructors and destructors are called. For example, timing a class constructor in a platform’s API would require a timer to be started before that constructor was called, whereas the subclass’ constructor won’t be called until after the platform API’s constructor. To handle this situation, ReCAP’s subclass employs multiple inheritance, first inheriting from another ReCAP-generated class that, in the case of the constructor, handles starting timers and

\(^2\) The use of macros for overriding functions causes problems with C++’s function overloading, although method overloading is handled correctly; an improved implementation could handle this case properly via name-mangling for overloaded functions.
other measurement code before the platform API's constructor is called; the subclass’ constructor then stops timers and records statistics. Destru ctors are handled in the reverse fashion due to the reverse order in which destructors are called.

For bottleneck detection, ReCAP also allows a user to associate a default “reason” for an API call (e.g., initializing, broadcasting, waiting to send due to a full buffer); “reasons” will be discussed in more detail in Section 5.2. For transfer functions, microbenchmark data can be entered in tabular form, where each row includes the transfer size (in bytes) and time taken to transfer that amount of data (in seconds). Microbenchmark data permits ReCAP to detect bottlenecks and make specific suggestions for communication-related bottlenecks (discussed in Section 5.3.1). ReCAP could also perform microbenchmarks automatically, such as upon installation or during a special calibration step, although additional information about how to use each API call would then be required; in the absence of microbenchmark data, ReCAP could simply detect performance deviations between different instances of the same API call (these techniques have been demonstrated in traditional HPC). Finally, a generic text field for platform-specific bottleneck suggestions is provided to associate known issues with given API calls. For example, on a Cray XD1 system [11], it is roughly 200 times more efficient to have an FPGA write data into the CPU's memory than to have the CPU read from the FPGA's QDR SRAMs [81].

A single API call could represent more than one behavior. For example, a transfer function may represent a read or a write depending on a class field or function argument, or standard library functions may be used to access the FPGA, such as an XtremeData XD1000 system’s [12] use of the standard C open function to initialize the FPGA. To support these scenarios, ReCAP supports a condition to determine whether or not monitoring is active for a given API call, allowing a single transfer that can read and write to be monitored separately for each case (by entering the prototype twice with conditions testing for a read and write, respectively) and for other uses of
standard functions (such as open) to be ignored unless the correct arguments are given that indicate an FPGA access. A subtype label is also provided to allow a user to easily distinguish different conditional versions of the same API call in visualizations and bottleneck reports.

While the information discussed above is sufficient for monitoring and bottleneck detection, ReCAP must also be able to initiate transfers in order to retrieve hardware performance data at runtime, requiring additional information about how to actually perform these transfers. While a general implementation could provide additional detail about all transfer types, we reduce the amount of data a user must enter by requiring this additional information for only one send and receive type. Thus, the user must provide include directives for all FPGA libraries, a minimum and maximum transfer size permitted, the data type for FPGA transfers, and a short code fragment (usually one or two lines) that shows how to send (or receive) an array of data to (or from) the FPGA. ReCAP must also be aware of how to allocate, free, and access these arrays; default code for these tasks is provided, but can be overridden since FPGAs sometimes require page-aligned data, and thus special allocation functions and data structures. Further, ReCAP may be provided with conditions indicating a send or receive error for better error checking. Additional miscellaneous information can be provided as well, including a size multiplier on the FPGA data type for platforms that have different data widths in hardware and software, a file exclusion list to prevent instrumenting the inside of a user-defined API, and application-specific constants needed for FPGA access.

Unfortunately, some issues remain with our approach. First, some APIs employ memory-mapped FPGA access, where reading or writing to a specific memory location from the CPU actually constitutes an FPGA transfer. It is not possible, in general, to statically determine whether a given pointer access is in the FPGA’s memory range, although runtime support is possible and many common cases could be detected statically. Thus, ReCAP currently requires wrapping such pointer accesses in simple
macros or functions before they can be monitored; functions can be inlined to prevent performance loss. Second, we assume the read and write functions used by ReCAP to retrieve hardware performance data have addresses associated with them; in fact, during instrumentation, the user must provide an address range that is unused by the application so that ReCAP can hijack and use this range for transferring data. This address-based approach precludes hijacking stream-based API calls that lack address information; this issue could be resolved in a number of ways including hijacking another API call that does provide address information (ReCAP only needs one address-based read and write API call), by embedding a special marker in the data stream and escaping that marker in any data sent by the application, or by setting a specific flag on the FPGA if such a feature were available and unused by the application. Despite these limitations, we have found that most platforms can be supported quickly (e.g., in a few hours). For example, ReCAP currently supports software and hardware APIs for a Nallatech PCI-X card [74], an XtremeData XD1000 system [12], and a GiDEL PROCStar III PCIe card [82]; Table 5-1 in Section 5.4 contains more information on these platforms.

5.1.2 Hardware

In order to determine when communication occurs in hardware, ReCAP must know what events and data are associated with a read or write. Thus, ReCAP requests information concerning the names of data and address signals for both incoming and outgoing transfers as well as HDL conditions that indicate when data is available for (or when data can be sent by) the user’s application. In addition, a user can provide an HDL code fragment to perform required actions when sending or receiving data (e.g., setting a valid or acknowledge flag high). Also, due to the wide variety of ways data can be transferred, ReCAP supports both memory-style access and block-transfer-style access (e.g., DMA transfers where the address and number of words are sent along with a request flag). For block transfers, the user must provide the signal name indicating the number of words and the condition indicating a transfer request (address signals
were already specified with the data signals earlier); the transfer ends when the number of words reaches 0. The user may also select whether the API will keep the address up-to-date on each cycle, or whether only a starting address is provided, in which case ReCAP will manage updating the address internally. Further, due to different latencies required by different platforms, the user may specify the number of cycles to delay outgoing data, including 0 for the same cycle. Finally, ReCAP requires the user to provide the top-level clock signal name (ReCAP currently only operates in one clock domain, although an extended implementation could support multiple clock domains) and reset condition.

As with software, we again assume an address-based scheme; if one is not present, techniques mentioned above are applicable here as well. We also note that our current framework only considers attachment to a CPU-FPGA communication port, whereas the FPGA may connect to another FPGA or external memory as well. We leave the extension of this framework to monitor these types of ports for future work, but this extension should be similar to the techniques presented here. Thankfully, this limitation only prevents automatic monitoring and bottleneck detection on these ports, as our current implementation can monitor any port via a few hardware pragmas in the top-level file; pragmas are discussed in Section 5.2.

### 5.2 Bottleneck Detection in RC Applications

As discussed at the beginning of this chapter, there are a number of factors beyond system and API diversity that complicate bottleneck detection in RC applications, including intricate hierarchies of interconnected components, component heterogeneity, and non-computational components. For example, Figure 5-1 depicts a simple mockup RC application that distributes sensor input between two application core pipelines, performs some computation using SRAM in that pipeline, collects results, and potentially offloads data for further processing to a dual-core CPU (with two software threads per core) before storing results in DDR memory. This application exhibits heterogeneity
since many different blocks exist (e.g., the P1, P2, and P3 pipeline stages are likely performing fairly different tasks), non-computational components (e.g., buffers, distributors, and collectors), and complex interaction amongst blocks (e.g., data may traverse through 6 or 10 blocks via several paths). Thus, these differences must be addressed when detecting bottlenecks in RC applications.

Figure 5-1. Directed graph of an RC application that takes input from a sensor, processes data using a two-core pipeline, potentially offloads data to threads on a multi-core CPU for further processing, and finally stores results in DDR memory.

We leverage our Phase 2 research in performance visualization and exploration for abstracting application behavior as a directed graph of blocks, where a “block” represents a software thread, a clocked VHDL “process” block, or a clocked Verilog “always” block (shown as black boxes in Figure 5-1). One key requirement of a block is that it operates in parallel with all other blocks, possibly with some dependencies due

---

3 Purely asynchronous logic will already have had timing optimized during synthesis and implementation and can be monitored on clock boundaries if necessary.
to interactions between blocks. VHDL “entity-architecture” pairs, Verilog “modules”, and CPU cores are called “components,” and may contain one or more blocks. While each block may perform an arbitrary amount of communication and computation (e.g., a single block may be simultaneously communicating with several blocks while computing a multiply-and-accumulate), we choose a block to be the fundamental unit of parallelism in our abstraction of an application. This choice represents a tradeoff between a desire for detailed recording and modeling of fine-grained parallelism and a desire to minimize extraneous detail recorded and visualized.

The performance visualization and exploration research discussed in Phase 2 also defines a pragma-based syntax, providing an application designer with a simple, unobtrusive methodology for specifying high-level information concerning application behavior; we extend this syntax here for the purpose of bottleneck detection. Figure 5-2 provides some examples of our extended syntax for software and hardware pragmas. Extensions to the syntax defined in our previous work include subdividing a “busy” category into “work” and internal “overhead” as well as the addition of a “reason” argument; for convenience, we briefly describe each part of a pragma’s syntax below, including aspects defined in our previous work as well as these extensions.

Each pragma in software or hardware defines a “state” that the given block is in when that pragma is reached in source code; these pragmas are placed either before an API call or before the first statement in an HDL branch. Pragmas can indicate a block is working, performing internal overhead, communicating, or waiting. Specifically, the following categories are permitted:

- **Work**: tasks directly associated with the objective of the application (e.g., matrix multiplication or convolution); this should be maximized
- **Overhead**: internal, auxiliary tasks that are artifacts of the implementation, such as bookkeeping or loop counter updates; this should be minimized
- **Send / Recv**: movement of application data to or from this block
#pragma recap writeX send(data, top.in, x1): words > 0
fpgaWrite(fpga, data, addr, words);
...
do {
  #pragma recap waitResult wait_recv(buffer_empty, top.out, r1)
  done = fpgaReadReg(fpga, addr2);
} while (done == 0);

A SW pragmas

case current_state is
  when recvXCoord =>
    --pragma recap ge
      --pragma recap getX recv(data, $CPU, x1)
  ...
  when compute =>
    --pragma recap mult work()
  ...
  when nextIter =>
    --pragma recap loopIncr overhead(update)
  ...
  when waitAck =>
    --pragma recap waitNext wait_recv(ack, top.out, w1): ack = ‘0’
    --pragma recap next recv(ctrl, top.out, w1): ack = ‘1’

B HW pragmas

Figure 5-2. Examples of user-defined pragmas.

- Wait_send / Wait_recv: synchronization with another block (e.g., waiting to send data to a locked resource or waiting to receive data from a block that is currently working)

Additionally, a condition can be provided to indicate when a pragma is active; thus multiple pragmas can be defined per API call or HDL branch with conditions indicating which pragma is active at any given time. For example, only one of the two hardware pragmas in the “waitAck” state in Figure 4-2B is active, depending on whether the “ack” signal is 0 or 1. All categories except for “work” (which has no arguments) accept an optional first argument representing the “reason” the application is performing the given operation (we will provide exact details for “reasons” later in this section). Finally, the communication and wait categories require a target and message ID to be provided (second and third arguments); a “target” indicates which block(s) the current block is
communicating with or waiting for while a “message ID” distinguishes between different communication involving the same blocks.

Figure 5-2A shows several examples of software pragmas for bottleneck detection. The first pragma is given a unique, user-defined name of “writeX” and is categorized as a “send.” The reason given is “data,” indicating the following API call is sending data, as opposed to control messages, to a process block labeled “in” within the “top” VHDL entity/architecture (targets are given using hierarchical references, or using specially named blocks such as $CPU or $MEM); a message ID of “x1” is given. Note that the condition indicates this pragma will be active only when “words” is greater than 0, as presumably the following API call does not represent communication if 0 words are sent. The second pragma, named “waitResult,” indicates the following API call is waiting to receive data because of an empty buffer; the API call is waiting to receive data which will have a message ID of “r1” from the “top.out” process block.

Figure 5-2B shows several examples of hardware pragmas. The first, named “getX,” is complementary to the first software pragma. The process block is described to be receiving data from the CPU with the same message ID; note that it is possible to receive multiple message IDs if needed (e.g., if different threads or different states within the same thread will interact with the same HDL pragma state). The next two pragmas indicate when the given block is working (e.g., performing some multiplication) and performing an internal overhead task with the “update” reason (e.g., updating a loop counter). The last two hardware pragmas demonstrate the use of conditions to differentiate behavior, even within the same state; in fact, no state machine needs to be present in hardware at all. The pragma named “waitNext” indicates the VHDL process block is waiting to receive an acknowledgment from another VHDL process block, “top.out,” when the “ack” signal is low. However, when the “ack” signal is high, the second pragma, named “next,” is active, indicating a control message (the acknowledge) is being received with the given message ID.
Bottlenecks can occur when one or more blocks are working slower than other blocks they interact with. In order to detect bottlenecks, we monitor time spent in each state defined by a pragma as well as transitions between these states, thus generating a Markov model of each block’s behavior. ReCAP employs three constructs in bottleneck detection: reasons, metrics, and bottleneck rules; the latter two are discussed below and are similar in notion to the rules, metrics, and parameters in [50] as well as to other techniques used in knowledge-based bottleneck tools [49], [33].

“Reasons” provide an explanation for why an application is performing a given task. In addition to just indicating that a given block is waiting to receive data, a user can now specify the block is waiting to receive because of an empty buffer or because of contention (or both – multiple “reasons” can be provided for a single state). These “reasons” are user-editable and are closely tied to bottleneck rules discussed below. For example, a contention bottleneck can be detected by searching for blocks that could achieve at least 1.05x speedup if all time spent in states with “contention” as a given “reason” were eliminated. Figure 5-3 shows the different “reasons” provided in ReCAP for each category.

The key concept behind “reasons” is that it is relatively easy for an application designer, if provided a list of “reasons,” to look at an API call or branch of HDL code and determine whether it is waiting for an acknowledge, sending control messages, or updating a loop counter, whereas it is relatively difficult for a tool to ascertain such high-level information automatically. Conversely, it is possible for a tool, given the above information for each API call or HDL branch of interest, to analyze the performance of each block for potential bottlenecks (accounting for block dependencies), whereas this task is fairly difficult for an application designer to perform manually.

“Metrics” are tool-provided measurements of runtime behavior (e.g., duration of an event, bytes transferred, bandwidth observed), which can be reported for any block or overall, and can be filtered by any combination of “reasons;” metrics are
used to define bottleneck rules, as discussed below. For example, a metric could return the minimum bandwidth for a software API function that was waiting to send due to a full buffer or waiting to receive due to an empty buffer, or a metric could return the total time a hardware block spent communicating or waiting. Metrics can be hardware- or software-specific, and currently include items such as total time spent; min/avg/max/total bytes transferred (or bandwidth observed); number of calls, call groups, min/avg/max consecutive calls, and call type (for software API calls); various statistics for microbenchmark data for a given API call (for comparing with actual bandwidth achieved); and miscellaneous metrics for computing formulas such as percentages and speedup. Unfortunately, adding, modifying, or removing metrics requires some detailed tool knowledge, and thus metrics are not user-customizable,
although we have localized where metric information is defined to facilitate the inclusion of additional metrics.

Since it is desirable to have a more flexible, user-defined metric structure, we have begun development of a hardware directive framework for adding user-defined hardware modules (with some constraints on port interfaces) that could extend ReCAP’s measurement capabilities, which we briefly mention here. These directives can be defined in terms of each other, and thus only a limited subset of hardware metrics must actually be defined in HDL; the remainder are formed by a simple macro syntax that can compose new metrics from other ones. The hardware subset includes directives such as conditional constructs (from basic to multi-cycle pattern-based conditions), sum, min/max, histogram, and several directives aiding in iteration which generate arrays of linear or exponential sequences; composite directives include items for correlation, additional multi-cycle pattern-based conditions, and even computing average and standard deviation. Software metrics could be similarly defined using C/C++ code modules.

“Bottleneck rules” include a description of the bottleneck, the bottleneck condition (any boolean C/C++ expression, typically employing one or more metrics), textual suggestions for resolving the bottleneck along with arguments to insert into the text (given in printf-style format), whether the bottleneck applies to individual blocks or whether it can be applied to the entire application (or both), any additional metrics the user may be interested in, and information concerning the original time and new time if the bottleneck were resolved (for computing speedup). Speedup is used to filter out bottlenecks that, if remedied, would not improve application performance by at least the user-defined speedup threshold. Bottleneck rules may be added or modified by the user through ReCAP’s GUI and are saved along with all “reasons” in a separate file to facilitate sharing of bottleneck detection strategies in a community; the file format is
currently that used by the standard Java “Properties” class, although other formats such as XML could also be of use.

ReCAP detects and produces reports concerning bottlenecks at runtime immediately after the user application has finished executing by testing all applicable bottleneck rules on all software and hardware blocks and for the application as a whole. We augmented ReCAP’s SVG-based visualization from our previous Phase 2 research with warning icons for blocks containing bottlenecks. These icons directly link to an HTML file containing bottlenecks detected in that block; internal named anchors are used to jump within a single bottleneck file, and thus all bottlenecks may be reviewed directly as well. All reported bottlenecks display information concerning the bottleneck type; potential speedup if the bottleneck were remedied; suggestions for remedying the bottleneck, which can include specific data from tool metrics; all values of metrics used to determine whether to display this bottleneck; and any other user-specified metrics of interest. Currently, speedup presented is ideal, assuming no other bottlenecks prevent the application from improving by the given amount; in reality, dependencies amongst blocks could result in considerably less speedup. Our prior performance exploration research in Phase 2 dealt directly with this estimation problem; thus, while not discussed or implemented in this work, integrating performance exploration would significantly improve the accuracy of speedup estimates. Finally, platform-specific suggestions and microbenchmark data are also included (the latter as a link to an HTML table). Figure 5-4 shows an example of ReCAP’s augmented visualization, which contains warning icons for blocks with bottlenecks detected, as well as the associated bottleneck information.

Although users are free to add or modify “reasons,” “bottleneck rules,” and “platform templates,” typical users of ReCAP need only add pragmas to their user source code to take full advantage of bottleneck detection; Section 5.3 will present our taxonomy of common RC bottlenecks (and associated “reasons”) that are provided by default
Late sender (FPGA)

Perform any other unrelated computation or communication during this waiting period (especially useful if non-blocking communication is available for your platform).

If the FPGA is not working at near-full capacity, a buffering scheme will likely improve performance.

- For streaming-based applications, employ a FIFO-style FPGA output buffer. The CPU should periodically check for data in (or be interrupted when data is in) the FPGA buffer. Ideally, the CPU will retrieve data when the FPGA output buffer is near full (so the CPU buffer should be as large as the FPGA output buffer), although, to prevent stalls due to a full FPGA buffer, it is better to retrieve data when the FPGA buffer is between 1/4 to 1/2 full.

- For block-memory-based applications, employ double-buffering on the FPGA's output buffer to overlap communication and work (if necessary to prevent contention, more than two buffers may be employed).

If the FPGA is working at near-full capacity, there is likely an imbalance between work done on the CPU and FPGA (the CPU has less load).

- Increase the production capabilities of the FPGA to match the CPUs processing rate (e.g., by increasing replication or clock frequency on the FPGA).
- Shift some work from the FPGA to CPU (e.g., have the CPU do some extra pre-/post-processing that eases the load on the FPGA).
- Decrease the retrieval rate on the CPU, spending time on other tasks.

1.17x
Based on reduction from 11.2426s to 0s out of 79.0223s.

RECAP_TIME_SW(wait_recv.wait_recv) 11.2426
RECAP_PERCENT_TIME_SW(RECAP_TIME_SW(wait_recv.wait_recv)) 14.2271

recvOutput_0 (Board::read) [RECV, RECV]
4.08898 s
100.0%
0.00330048 s
89.3%
0.0290843 s

queryDone_1 (Board::read) [WAIT_RECV, RECV, RECV]
11.2426 s

Figure 5-4. Example of bottleneck detection results, showing inclusion of warning icons to indicate blocks with bottlenecks and a portion of the detailed bottleneck report.

in ReCAP. In addition, while we deal with only CPUs or FPGAs here, our block-based approach could be applicable to a broad class of heterogeneous systems (e.g., GPUs, DSPs, Cell processors); only the methodology for instrumentation and measurement must change.

As mentioned at the beginning of this chapter, our current implementation relies solely on profile data for bottleneck detection, even though traditional HPC employs trace data for this purpose; to compensate, ReCAP does provide a number of useful time-dependent profile metrics, such as those reporting on consecutive API calls and transitions between different states in both software and hardware. While ReCAP supports tracing in both software and hardware, the largest FPGAs currently contain
less than 10MB of on-chip memory; in contrast, a 40-block, 100MHz design generating 64-bit trace records every 10 cycles would require 3.2GB/s. Further, future devices with larger memories are likely to employ larger applications, which will likely generate larger amounts of trace data. Thus, reducing the number of trace events along with compressing or otherwise pre-processing trace data to save storage could be of particular use. Also, unlike with CPUs, it can be difficult to pause a user application in an FPGA in order to write trace data to memory; the FPGA may interact with external hardware in a timing-dependent manner, allowing a pause to miss critical data or become unsynchronized with another device. If handshaking were required between the FPGA and all external devices, pausing the user’s application on the FPGA may be a viable method for recording trace data, albeit by incurring additional runtime overhead.

We finally note that ReCAP does currently employ a memory hierarchy to improve tracing capabilities; local trace buffers are constructed from on-chip memory and then fed into a single, per-chip trace collector using either internal or external memory, the latter of which is connected manually at this time. This approach allows high-bandwidth recording for small bursts of trace records for each block while providing storage for larger amounts of trace data. As tracing can provide a wealth of data useful for bottleneck detection, further research into efficient performance-based tracing methodologies for FPGAs could be of great use.

5.3 Common Bottlenecks in RC Applications

In this section we attempt to systematically explore and taxonomize potential RC bottlenecks, drawing upon our experience with RC applications as well as concepts and techniques from knowledge-based bottleneck detection in traditional HPC. Since an RC application may contain all the problems of a standard parallel application, traditional HPC bottleneck detection tools are quite beneficial in tuning the CPU portion of an RC application, allowing this work to be easily integrated with the significant amount of literature and tools already present for traditional HPC. In fact, since ReCAP builds upon
PPW, it inherits all of PPW’s software bottleneck analysis capabilities. Thus, we focus on bottlenecks that may occur due to CPU-FPGA communication and for bottlenecks within the FPGA, culminating in a taxonomy of potential RC application bottlenecks (including detection and optimization strategies for each bottleneck) that dovetails with traditional HPC knowledge-based bottleneck detection research. Figure 5-5 provides our taxonomy of possible bottlenecks, which will be discussed for the remainder of this section. Note that while this taxonomy contains common bottlenecks that apply to both software and hardware, these bottlenecks are defined separately in ReCAP to permit different detection and optimization strategies for the same bottleneck.

![Diagram of bottleneck taxonomy](image)

Figure 5-5. Taxonomy of common bottlenecks in an RC system.
We break our discussion into four general bottleneck categories: communication (Section 5.3.1), synchronization (Section 5.3.2), internal overhead (Section 5.3.3), and imbalances (Section 5.3.4). Since a number of bottlenecks below are detected simply by determining if a significant portion of time for a block was spent performing tasks with an associated reason (where “significant” implies a possible speedup of more than a user-defined threshold if the problem were remedied), we will only highlight detection strategies that require additional conditions to be tested.

5.3.1 Communication Bottlenecks

Bottlenecks may occur during communication between blocks (recall that a block can be a VHDL process, Verilog always block, or software thread, and thus this includes communication between CPUs and FPGAs, blocks within an FPGA, and even between FPGAs). If a block spends a significant portion of time communicating with other blocks, this constitutes a potential “general communication” bottleneck. However, some blocks may be solely purposed for communication (especially common in FPGAs); in hardware, this case could be detected by searching for blocks with no pragmas specifying the “work” category, although, in software, ReCAP currently considers all time spent between API calls to be “work,” making such detection more difficult. Suggestions for ameliorating this generic bottleneck include overlapping communication with other tasks if possible, employing bit-packing or compression, and repartitioning the algorithm to reduce the data transferred. Note that if more specific communication bottlenecks are detected in a block, basic bottleneck suggestions will still be displayed, thus eliminating the need to repeat these suggestions for more specific cases.

Another common scenario in communication is that transfer rates can vary drastically with transfer size and type depending on the protocols and interconnects used; see Figure 5-6 for a subset of transfer sizes and types for an XtremeData XD1000 platform, a Pentium-4 Xeon system equipped with a Nallatech H101-X accelerator, and a quad-core Xeon E5520 system equipped with two quad-FPGA GiDEL PROCStar III
cards. Thus, using an inefficient transfer type or size constitutes a potential bottleneck. ReCAP currently only detects this bottleneck in software, since hardware communication is often not packetized and thus incurs no overhead beyond the actual data transfer, although more complicated protocols could be used in hardware for inter-FPGA communication if the FPGAs were not tightly coupled. For example, Figures 5-6A and 5-6B demonstrate the potential for large differences between read and write performance, whereas Figure 5-6C shows how different transfer types may perform better for different transfer sizes. Figure 5-6B also shows that non-overlapped transfers, where multiple simultaneous transfers can occur via either non-blocking communication or collective functions (e.g., broadcast, scatter, gather, reduce), can be a potential bottleneck in RC applications; this is also true in hardware where it is ideal for a block to perform as much communication as possible in parallel with other tasks, and thus communication should be overlapped wherever possible. These phenomena have been well-researched in non-RC contexts [83].

ReCAP detects such bottlenecks by comparing actual bandwidth recorded with the best microbenchmark bandwidths from the platform template; as mentioned in Section 5.1, ReCAP could also automatically perform microbenchmarks or detect performance deviations between similar API calls. From this data, ReCAP can make specific suggestions on the best type to use if transfer size was held constant, the best size to use if transfer type was held constant, and the best overall size and type to use for the platform. The best size is given as a range (e.g., indicating transfer sizes that achieve 95% of maximum bandwidth for that transfer type), with a link to microbenchmark data provided for further investigation. This type of specific suggestion, along with potential speedups, is just one example of where ReCAP’s suggestions can be useful, even for expert RC designers. Suggestions for increasing transfer size include unifying different transfers together (either by placing different items consecutively in the memory map or by embedding control, masking, or address information in the
Figure 5-6. Platform transfer rate vs. transfer size across various RC systems and communication types, demonstrating common communication problems. The non-blocking transfer rate was computed as the sustained cumulative transfer rate of eight concurrent non-blocking transfers.

communication stream) and creating buffers, or increasing buffer sizes, to handle larger transfers (if the data cannot be processed in real-time). Transfer size may be decreased simply by breaking a large transfer into pieces, in which case buffers may be able to be reduced in size to save resources. As an example, transferring thirty-two

---

4 ReCAP’s actual suggestions often contain additional description and examples not included here for brevity, but which are quite useful for less-experienced RC designers.
128KB packets on the Nallatech platform is more efficient than transferring a single 4MB packet (360MB/s vs. 291MB/s).

Control messages deserve specific mention as these are commonly employed in RC applications and typically incur significant overhead. ReCAP detects control messages directly from pragmas that specify communication to be control oriented. Suggestions for addressing control bottlenecks include moving as much control logic onto the FPGA as possible (e.g., instead of reading from an FPGA register, testing a value, and then possibly starting some action on the FPGA, the register test can be moved onto the FPGA), employing unused address bits in a read to carry control information, or increasing the size/duration of the work being controlled (e.g., loop unrolling), thus reducing the amount of control needed.

In software and hardware, it may be possible for both the application and interface to stall a transfer (e.g., due to no data being available or preemption by another transfer), potentially causing a stalling bottleneck on behalf of either the application or interface. These bottlenecks can be detected in hardware by monitoring the appropriate signals on the interface port; however, it is rare that an API would surface such information in software, and stalling could occur in some intermediate buffer not easily monitored from either software or hardware. These bottlenecks could be remedied by the addition of buffers to prevent stalling or, in the case of the application, by increasing the rate at which the application can accept data (e.g., via replication of application components); while a similar suggestion could be made for improving an API, this is typically not possible for an application designer, who often does not have the source code for an API available, and thus reducing the application’s data processing speed is suggested as this is unlikely to affect performance and may save resources or power.

5.3.2 Synchronization Bottlenecks

Synchronization bottlenecks indicate a block is waiting on one or more other blocks, either to transfer data or to reach a certain point in execution before continuing.
If two blocks must communicate synchronously, one block may arrive to its side of communication later than another, permitting a “late sender” or “late receiver” bottleneck (common terminology in traditional HPC bottleneck detection). Suggestions for remediying this situation include performing unrelated computation or communication while waiting, buffering or double-buffering (depending on whether the application is streaming-based or block-memory-based), or remediying an imbalance between blocks (either by increasing the efficiency of the slower block or decreasing the efficiency of faster blocks, the latter of which could reduce power or resources used). If employing a buffer in the streaming case, ReCAP suggests a transmission size that is at most half the total buffer size (to ensure the buffer can be refilled before it drains) and at least a quarter of the total buffer size (to maximize communication bandwidth).

As buffering is a common technique for addressing synchronization bottlenecks, additional common bottlenecks include a full-buffer or empty-buffer bottleneck. ReCAP suggests an imbalance may exist, as mentioned above, or that burst traffic may be occurring if the buffer is full and empty often, in which case the buffer size should be increased to handle larger bursts or the transmission size should be decreased and the transmission frequency increased to smooth out the bursts, such as by having a separate thread handle transmission periodically.

Several additional types of synchronization bottlenecks are also handled in ReCAP. A polling bottleneck can occur when status of another block is repeatedly queried until some condition is met. In hardware, this situation is extremely common and is similar to a normal synchronization bottleneck; however, in software, this behavior is problematic, wasting communication bandwidth and processor time. ReCAP detects this bottleneck by searching for any API call that is waiting to receive data and is, on average, executed three or more times consecutively. Suggestions include using interrupts if supported by the system and API, moving any unrelated tasks between polling calls, and using a separate thread to perform the poll, potentially adding an indicator into data returned
from the poll, such as “percent done,” to estimate time the thread should sleep before
polling again.

An acknowledge bottleneck can occur when waiting for a block to acknowledge
some event. In this case, ReCAP suggests speculatively continuing without the
acknowledge if an acknowledge is expected, storing any checkpoint or state information
needed to restart or retry a task if the acknowledge does not arrive so long as this
storage is not prohibitive in terms of memory requirements. A barrier bottleneck
indicates that a block is waiting for other blocks to reach a given point of execution
before continuing. In this case, decoupling these blocks via buffering may be possible
if no feedback or resource contention exists that would prevent blocks from continuing.
For example, to average the results from several blocks, a running average could be
computed or a buffer could be added to the output of each block, with the average beingcomputed from the buffer output.

A contention bottleneck detects when blocks are spending a significant time waiting
on a shared resource. There are a number of suggestions for alleviating this type of
bottleneck. One suggestion involves reducing time needed to acquire or release a lock,
usually by adding an arbiter that manages all requests to the shared device. Another
suggestion includes increasing the efficiency or replication of the shared resource
itself can be useful, such as by placing the memory in a faster clock-domain that can
handle two transactions per application cycle or by either replicating a shared memory in
hardware or using an additional memory bank from software (issuing reads to different
copies while writes are issued to all for consistency). Additional suggestions include
forcing a staggered ordering to reduce or eliminate locking, increasing (or decreasing)
granularity of tasks performed between lock and release if the cost for locking is high (or
low), ensuring no block holds a shared resource any longer than necessary, ensuring
the minimum locking is performed to still ensure consistency, and finally reducing the
efficiency of other blocks or the number of blocks accessing the shared resource to potentially save resources with little performance loss.

5.3.3 Internal Overhead Bottlenecks

A block can spend a significant amount of time performing bookkeeping or other internal overhead tasks, thus allowing an internal-overhead bottleneck to occur. As with other generic bottlenecks, this bottleneck may be addressed by parallelizing, pipelining, or otherwise overlapping these tasks with others or by increasing the size/duration of the work associated with internal overhead so that fewer overhead tasks are performed (e.g., loop unrolling). However, many specific variants of this bottleneck are also detected. Initialization, finalization, and update bottlenecks occur when significant time is spent in a block performing initialization, update, or finalization tasks. Suggestions include reducing or eliminating this overhead; for example, if a histogram is to be accumulated in memory and thus needs to be cleared for each new dataset, it may be possible to instead maintain a bit-vector that indicates which memory locations have been accessed since the last dataset and thus determine whether to store or add a given value to the current memory location. A multiple-initialization or multiple-finalization bottleneck indicates that software has not only spent a significant amount of time configuring or releasing the FPGA, but that software performed this task several times; this typically indicates several different configuration files have been loaded during runtime to accelerate different phases of an application. These bottlenecks are detected by ensuring the API call is of the appropriate type (e.g., configure, release) and called at least twice. Suggestions for optimization include adding functionality to, or generalizing functionality in, each configuration file to reduce the number of configurations, rescheduling the CPU's work if the same configuration file is loaded multiple times so that overhead from reprogramming the same file is reduced, and repartitioning the algorithm to minimize the amount of functionality needed by the
FPGA, such as by moving some pre- or post-processing tasks from the FPGA to the CPU.

An interrupt-processing bottleneck may occur if a block is interrupted too often by another block. Note that while the physical interrupt is communication, this refers to interrupt handling, and thus is an internal task. In this case, the number of exceptional circumstances that cause interrupts should be reduced, such as by increasing precision to reduce overflow interrupts or by resolving the most common interrupts locally, if possible. A delay bottleneck occurs when a block must delay for some internal reason, such as when waiting for an extra-long combinatorial path or internal pipeline, the latter of which is handled specifically in the “pipeline fill/drain” and “pipeline flush/stall” bottlenecks. Delay bottleneck suggestions focus on reducing the latency causing the delay or overlapping these latencies with other delays or useful work. Suggestions for addressing a “pipeline fill/drain” bottleneck include filling a pipeline with the next dataset while still processing or draining the previous dataset. Suggestions for addressing a “pipeline flush/stall” bottleneck include decreasing latency or pipeline stages, although this must be balanced with the effect on the FPGA’s maximum frequency; moving detection of flush conditions earlier to minimize stages flushed; and having pipelines with a large number of stalls process data from multiple streams, interleaving independent data into the pipeline to minimize dependency stalls.

5.3.4 Imbalance Bottlenecks

An imbalance of computation between two or more related blocks is also considered a potential bottleneck. A stage imbalance refers to an imbalance where a block depends on other blocks, such as in a pipeline. Optimization of this bottleneck involves improving the performance of slower blocks (e.g., through additional stage division or replication) or by reducing the performance of faster blocks to potentially save resources. A load imbalance indicates an imbalance between parallel blocks that receive data, potentially indirectly, from the same source block, such as is common with replicated cores.
Possible optimizations for this bottleneck include improving the data distribution scheme (e.g., employing a look-ahead round-robin scheme that determines whether any of the next four or eight blocks are idle, skipping them all if so, or employing a priority-based selection scheme such as least-recently used), adding input buffers, or changing the replication factor to possibly distribute data more evenly (e.g., simply using a prime replication factor may distribute data more evenly than a heavily composite replication factor).

ReCAP detects both bottlenecks by searching through the block dependency graph given by user pragmas to determine where potential bottlenecks are located; a number of metrics are provided that return statistics over these block groups to facilitate such bottleneck detection. Within the provided suggestions, specific data about the best and worst performing blocks as well as the average performance of all related blocks are also given to allow a designer to determine how severe the imbalance is and what techniques may be best in addressing the bottleneck.

5.4 Case Studies

To demonstrate the utility of RC bottleneck detection and platform templates, our extended ReCAP tool was employed on two different applications on a total of three diverse RC platforms: a time-domain finite impulse response benchmark [84] on a GiDEL PROCStar III [82] and a 2-D probability density function estimator application [85] on both a Nallatech H101-PCIXM card [74] and an XtremeData XD1000 system [12]. Table 5-1 provides details for these RC systems.

5.4.1 Time-Domain Finite Impulse Response

The time-domain finite impulse response (TDFIR) benchmark is part of the HPEC challenge benchmark suite [84] and has been accelerated on GPUs as well [86]. For an FPGA-accelerated version, we implemented convolution for real numbers rather than complex; however, for consistency of results, we report all numbers in GFLOPS, thus accounting for the fact that each basic computation involves only two floating-point
Table 5-1. RC platforms employed during case studies.

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU(s)</th>
<th>FPGA(s)</th>
<th>API type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nallatech H101-PCIXM</td>
<td>Pentium 4 Xeon 3.2GHz</td>
<td>One Virtex 4 LX100 (PCI-X card)</td>
<td>C, simple memory</td>
</tr>
<tr>
<td>XtremeData XD1000</td>
<td>Dual-core Opteron 285 2.6GHz</td>
<td>One Stratix II S180 (HyperTransport socket)</td>
<td>C++, DMA</td>
</tr>
<tr>
<td>GiDEL PROCStar III</td>
<td>Quad-core Xeon 5520 2.26GHz</td>
<td>Four Stratix III E260s (PCIe card)</td>
<td>C++, simple memory</td>
</tr>
</tbody>
</table>

operations rather than eight for the complex versions. This case study was performed on the GiDEL system (Table 5-1) using Quartus 9.1SP2 and GCC 4.4.3 with -O3 optimization; all times given are the average of three executions. The TDFIR benchmark was able to execute at 125MHz on the FPGA for both the original and optimized versions, and thus all executions were performed at this frequency for uniformity of results. All FPGA benchmark execution times include all data transfer times between the CPU and FPGA as well as any other needed CPU tasks such as data movement; we only exclude the FPGA initialization/finalization time, since the configuration file could be preloaded once and then used indefinitely (e.g., for streaming large amounts of data through).

Three datasets were used for evaluation, which consisted of random data with the same kernel size, input size, and iterations (i.e., the number of different sub-datasets with the given kernel and input size that must be computed). Dataset A and B are the standard datasets given in the HPEC challenge, while dataset C is the largest dataset from [86] (Table 5-2). We compare FPGA results from a Stratix III E260 to both the Intel Xeon E5520 processor (the host processor in the GiDEL system) and the results given in [86] for an NVIDIA 8800GTX.

Upon executing the FPGA version of TDFIR, two problems were observed. First, the single-FPGA version was slower than either the CPU or GPU version for the first two datasets, although the FPGA version did achieve a 9.1x speedup over the
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Kernel Size</th>
<th>Input Size</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>12</td>
<td>1024</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>128</td>
<td>4096</td>
<td>64</td>
</tr>
<tr>
<td>C</td>
<td>4096</td>
<td>32768</td>
<td>128</td>
</tr>
</tbody>
</table>

Table 5-2. Datasets evaluated for TDFIR benchmark.

CPU and a 3.0x speedup over the GPU on dataset C (Figure 5-7A). Second, when scaling up from 1 to 4 FPGAs, performance scaled poorly, resulting in less than 1.06x speedup for datasets A and B and less than 2.1x speedup for dataset C, even though no communication or synchronization was needed between FPGAs; different iterations were simply scattered to the different FPGAs (Figure 5-7B).

We then employed ReCAP’s automatic bottleneck detection on the FPGA version of TDFIR, using dataset C for execution. Overhead for obtaining performance data was acceptable, incurring an additional 5.1% in software time, 1.8% of FPGA logic resources, 0.8% of FPGA register resources, and a negligible (less than 1%) decrease in frequency; the benchmark still met the required 125MHz. ReCAP detected several communication bottlenecks in software including “inefficient communication size and type” (ideal 11.63x speedup), “control overhead” (ideal 3.60x speedup), a “late sender” bottleneck for the FPGA (the CPU is ready to receive data but the FPGA is late in sending that data, ideal 2.52x speedup), and a “late sender” bottleneck for the CPU (the FPGA is ready to receive data but the CPU is late in sending that data, ideal 1.63x speedup). As mentioned earlier, the speedup numbers are ideal, and thus our performance exploration framework in Phase 2 should be used for more accurate predictions. For example, while ReCAP suggests above that an 11.63x speedup is possible if the associated bottleneck is remedied, ReCAP’s visualization shows the FPGA is working 54.1% of the time when processing dataset C, thus limiting speedup to at most $\frac{1}{0.541} = 1.85x$ (the FPGA is working much less of the time for datasets A and B and thus better speedup is possible). Nonetheless, the ideal speedup numbers given
Figure 5-7. TDFIR performance on various devices (including both the original and optimized FPGA performance after bottleneck detection. are still useful for providing an upper bound and serve as a severity indicator for a given bottleneck and thus are often a good ranking of which bottlenecks to address first.

We first chose to address the “inefficient communication size and type” bottleneck. For one API call, bottleneck detection indicated a 2.52x speedup was possible if the transfer size were increased to between 32MB and 64MB (or higher for asynchronous communication) while a 2.39x speedup was possible by switching to low-latency transfers. As changing the transfer size would be more difficult and yet not result in much additional performance beyond that gained by switching the transfer type, we chose the latter approach for this API call. However, for several other API calls, solely switching to a different communication type was not recommended by ReCAP, and thus
we increased the transfer size by a factor of 8 to 10 via buffers and logic to handle batch transfers\(^5\). These changes alone resulted in a 1.47x improvement in performance for dataset C on one FPGA and a 1.68x performance improvement for 4 FPGAs. However, with only a 2.4x speedup between the 1 and 4 FPGA versions, scalability was still low.

We next addressed the “late sender” bottlenecks, which suggested overlapping the waiting period with other computation or communication, using asynchronous transfers if available, as well as employing double or higher-order buffering. Thus, we overlapped communication using asynchronous transfers and quadruple buffering with three different external memories as well as internal memory, providing an additional 1.12x improvement in performance for dataset C on one FPGA and an additional 1.73x performance improvement for 4 FPGAs; the 1-FPGA version experienced less performance increase due to heavy use of its computational units. As shown in Figure 5-7C, the second optimization resulted in a scalability of over 90% of the ideal (3.61/4.00) for dataset C and noticeable improvement in scalability for dataset B.

Thus, by employing these two optimizations, performance was improved by 1.65x for the single FPGA version (achieving 27.3 GFLOPS) and 2.9x for the 4 FPGA version (achieving 98.9 GFLOPS) on dataset C, with the 4-FPGA version achieving a total of 54.4x speedup over the software baseline; final performance for each dataset is shown as black bars in Figure 5-7A and with circle markers in Figure 5-7D. Further, the single FPGA version now performed 1.15x better than the CPU on dataset B, whereas the original version had performed 6.7 times slower than the CPU. While performance on dataset A was increased by 5.7x, the FPGA continued to perform poorly

\(^5\) ReCAP suggested increasing the transfer size much further for optimal transmission, which was infeasible due to memory limitations. Thus, we manually determined a balance between significantly better bandwidth from the microbenchmark table, memory overhead, and achieving even divisibility into the number of iterations for each dataset; the last requirement resulted in a batch size of 10 for dataset A since 8 does not divide evenly into the stipulated 20 iterations for that dataset.
due to communication overhead; the Nvidia 8800GTX experienced a similar, albeit less pronounced effect for this dataset, as seen in Figure 5-7A. Further improvements suggested by ReCAP, such as moving a memory clear operation from software into FPGA logic, would likely achieve some additional speedup, although the FPGA cores were observed to be working 68.7% of the time in the 4-FPGA version, limiting the amount of further speedup possible without employing additional resources or a better algorithm. It is noteworthy that given the reported bottlenecks and optimization suggestions, actual optimizations were made within two days, resulting in higher performance as well as improved productivity compared to manual, ad-hoc bottleneck location and optimization.

5.4.2 2-D Probability Density Function Estimation

The 2-D probability density function (2DPDF) estimation application is used in various engineering, financial, and scientific fields where non-parametric probabilistic approaches are required; the application is computationally intensive, involving $O(m \times n^2)$ operations where $m$ is the number of sample points and $n$ is the number of bins per dimension. Our implementation uses the Parzen-window algorithm and a fixed-point format ([18,9] external precision, [48,18] internal precision, given in [total, fractional] format) [85].

Our experimental setup consisted of the Nallatech system (Table 5-1), using GCC 4.4.3 and Xilinx ISE 11.5 to compile all C and VHDL files, respectively. A software baseline was written in C using only integer arithmetic to better compare to the FPGA’s fixed-point format, as the CPU’s floating-point version was slower. This baseline was compiled with -O3 optimization and executed on the attached Pentium 4 Xeon 3.2GHz processor; all execution times were computed from the average of 3 executions. The 2DPDF application was capable of operation at 100MHz or higher for all design variants, and thus 100MHz was used for uniformity of performance results. Similar to the convolution case study, all FPGA application execution times include all data transfer
times between the CPU and FPGA as well as any other needed CPU tasks such as
data movement; we only exclude the FPGA initialization/finalization time, since the
configuration file could be preloaded once and then used indefinitely (e.g., for streaming
large amounts of data through).

Initially, we attempted to gain speedup by extending the 2DPDF application to
a multi-core design within the FPGA. Figure 5-8 shows the initial execution times for
several multi-core variants (circle markers). While our software baseline required
250.5 seconds to process 1,024,000 points, the 20-core FPGA design required only
64.5 seconds for the same dataset, resulting in a 3.9x speedup. However, Figure
5-8 demonstrates that these additional cores provided diminishing performance
improvements.

![Figure 5-8. Speedup of initial and improved versions of the 2DPDF application when
compared to a CPU baseline.](image)

We then employed ReCAP’s automatic bottleneck detection on the 20-core design.
Overhead for obtaining performance data was acceptable, incurring an additional
14.2% in software time (due to millions of API calls during execution), 4.3% of FPGA
logic resources, 2.3% of FPGA register resources, and a maximum 2.2% decrease in
frequency. ReCAP identified a number of potential bottlenecks including a “CPU late
sender” bottleneck with ideal 16.21x speedup, a “control” overhead bottleneck with ideal
1.55x speedup, an “inefficient communication size/type” bottleneck with ideal 9.83x speedup, a “control” overhead bottleneck with ideal 1.38x speedup, and an “FPGA late sender” bottleneck with ideal 1.32x speedup. Specifically, bottlenecks were also detected in five of the twelve individual API calls in software involving clearing memory, starting and stopping cores, checking to see if cores were complete, and reading the output; each API call’s potential speedup ranged from 1.15x to 1.34x. The generic “all overhead” bottleneck on the FPGA indicated that, if the FPGA were fully utilized, a possible 16.81x speedup could be achieved.

Based on the optimizations suggested above, we focused on consolidating the large number of small transfers performed and on moving control logic onto the FPGA. Specifically, several input buffers were increased in size from 2KB to 32KB, the minimum ideal transfer size suggested by ReCAP; output buffers were placed consecutively in the memory map, permitting a single larger read rather than several smaller reads; register data was consolidated to reduce the amount of data polled; and control logic was moved onto the FPGA, performing tasks such as automatically clearing intermediate FPGA buffers when receiving new data rather than relying on software to manually control this process.

Speedup for the improved 2DPDF application increased from 3.9x to 44.4x when compared with the software baseline, resulting in an 11.4x performance improvement between the unoptimized and optimized versions (square markers in Figure 5-8) and demonstrating far more linear speedup with respect to the number of cores employed. Interestingly, we also discovered that the new version incurred slightly less rounding error since FPGA data is rounded when transferred to the CPU and fewer transfers were employed in the optimized version, although the error incurred by the original version was deemed acceptable. Again, it is noteworthy that given the reported bottlenecks and optimization suggestions, actual optimizations were made within a day, resulting in a
significant performance increase as well as improved productivity compared to manual approaches for bottleneck detection and optimization.

As a final attempt to improve performance, we ported the optimized 2DPDF application to the XD1000 platform (Table 5-1), which allowed us to increase computational resources by a factor of 2.4x. Software and hardware were compiled with GCC 4.3.2 with -O3 optimization and Quartus 9.1SP2, respectively. We obtained an additional 2.5x speedup over the Nallatech implementation; note that a speedup greater than 2.4x is possible due to faster transfers afforded by the HyperTransport interconnect on the XD1000. We used ReCAP’s automatic bottleneck detection on the ported application, incurring an additional 5.2% of software runtime overhead, 4.7% of FPGA logic resources, 1.8% of FPGA register resources, and a 15.3% frequency degradation (due to the fact that the application filled 87% of the device before instrumentation, and thus 92% with instrumentation); the 100MHz requirement was still met by the instrumented version.

ReCAP’s bottleneck detection returned several potential bottlenecks, but the ideal speedups were much lower than in the previous cases. For example, in software, all speedups were less than 4.2x; in hardware the cores showed speedup potential of at most 1.31x (note that a stage imbalance bottleneck was the exception, showing a potential of up to 5.5x; however, the stage imbalance it referred to involved a basic distribution core that performs very little communication and no work; with the integration of the aforementioned performance exploration framework, such false positives could be significantly reduced). Thus, given lackluster performance improvements predicted, we chose not to optimize further, although there are scenarios where a user may believe the potential speedup warrants additional effort, such as if the application must be executed many times or if runtime were significantly longer. This result underscores an often overlooked benefit of bottleneck detection; bottleneck detection can often be as useful in indicating what not to optimize as it is in what to optimize. Table 5-3 gives execution
times for the 2DPDF application on both platforms, showing that the XD1000 version achieved a 112.9x speedup over the original Pentium 4 Xeon 3.2GHz software baseline.

<table>
<thead>
<tr>
<th>Application Version</th>
<th>Runtime (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4 Xeon CPU</td>
<td>250.515</td>
<td>1.0</td>
</tr>
<tr>
<td>Nallatech (FPGA original)</td>
<td>64.506</td>
<td>3.9</td>
</tr>
<tr>
<td>Nallatech (FPGA optimized)</td>
<td>5.648</td>
<td>44.4</td>
</tr>
<tr>
<td>XD1000 (FPGA)</td>
<td>2.218</td>
<td>112.9</td>
</tr>
</tbody>
</table>

Table 5-3. 2DPDF results for both the Nallatech and XD1000 platforms. Speedup is given with respect to the software baseline executed on a Pentium 4 Xeon 3.2GHz CPU.

5.5 Conclusions

In this phase of research, we presented what we believe to be the first automatic bottleneck detection framework and tool for RC applications, including a framework for platform templates that permits more accurate, platform-aware bottleneck detection as well as tool portability across diverse RC systems. These templates are easily created by end-users, typically in a few hours provided the platform fits within the generic platform-template model. In addition, we formulated what we believe to be the first taxonomy of common bottlenecks for RC applications, along with associated detection and optimization strategies for each of these bottlenecks, to populate ReCAP’s knowledge base for bottleneck detection. Our bottleneck knowledge base is extensible, providing for user detection of bottlenecks not envisioned by the authors. The knowledge-based bottleneck detection framework and platform-template system were implemented by extending our work from Phase 1 and 2, providing users with a full-featured RC performance analysis framework and tool for a diverse set of RC systems that can significantly accelerate the optimization process.

We then demonstrated bottleneck detection in ReCAP via two case studies involving a time-domain finite impulse response (TDFIR) benchmark from the HPEC challenge and a 2-D probability density function (2DPDF) estimation application on
a total of three diverse platforms. ReCAP reported a number of bottleneck types discovered in both software and hardware along with optimization suggestions and potential speedup for each bottleneck type. Several of these optimization suggestions were employed to achieve an additional 2.9x speedup for TDFIR, resulting in a total 54.4x speedup over the CPU baseline, and an additional 11.4x speedup for 2DPDF estimation, resulting in a 44.4x speedup over the CPU baseline. We ported 2DPDF to the XD1000 platform (which provided an additional 2.5x speedup due to increased computational resources) and, based on lackluster potential speedup reported by bottleneck detection, did not further optimize the application, resulting in a total 112.9x speedup over the Pentium 4 Xeon 3.2GHz CPU baseline.
CHAPTER 6
CONCLUSIONS

In this research, the first framework and tool for general-purpose performance analysis of RC applications (per an extensive literature review), called ReCAP (reconfigurable computing application performance), was presented. Background and related work were presented in performance analysis for traditional high-performance computing (HPC) systems (including visualization and bottleneck detection), reconfigurable computing (RC), RC performance modeling and simulation, and finally RC debugging, performance analysis, and tool portability. An initial framework was defined in Phase 1 for instrumentation, measurement, and basic presentation of performance data, extending a current HPC performance analysis tool (PPW) to support performance analysis for multi-CPU/FPGA systems. In Phase 2, an extension to Phase 1 was presented that provides full-fledged performance visualization capabilities in ReCAP to aid the user in rapidly understanding application behavior. In addition, a methodology for performance exploration using runtime performance data was also presented, providing a framework for accurately predicting changes to application performance if optimizations were implemented. Finally, in Phase 3, ReCAP was further extended to provide automatic detection of bottlenecks (including potential optimization strategies and expected speedup). This phase of research included an enumeration and categorization of common RC-application bottlenecks as well as a platform-template system permitting more accurate, platform-aware bottleneck detection across diverse systems and tool portability. Case studies were performed across three systems to demonstrate the utility of ReCAP’s bottleneck detection and platform-template frameworks.

This work contributes the ReCAP performance framework and tool along with related research and findings, where such research and framework was lacking. The ReCAP framework and tool has been demonstrated to be applicable to a diverse set
of RC systems and applications as well as shown to significantly aid designers in quickly understanding and remedying performance bottlenecks in their RC applications. This framework contributes to all five areas of performance analysis as mentioned in Section 2.2; instrumentation, measurement, automated analysis, visualization, and optimization (via performance exploration in Phase 2 as well as optimization suggestions given for addressing bottlenecks in Phase 3). In addition, this framework has also contributed to ongoing research in performance analysis for HLL-based RC applications as demonstrated in Curreri et al. [20].

Future areas for research include expanding the flexibility and utility of bottleneck detection through user-defined metrics, integration and refinement of the performance exploration framework, trace-based RC bottleneck detection, and even employing data-mining for bottleneck detection. In addition, integration of this research into DSE tools could benefit both tools, providing a comprehensive environment for studying application behavior from conceptual design through optimization. Further research is also warranted in RC tool portability, such as exploring a generic framework for monitoring FPGA-FPGA and FPGA-memory interfaces, as well as further study and refinement of visualization techniques for large-scale heterogeneous applications, such as including communication channel usage and efficiency. Finally, research on the difficult task of automatic optimization could be of significant use, reducing the burden on the user to manually optimize their RC applications.
APPENDIX A
SYSTEMS SUPPORTED BY RECAP TOOL

A.1 Nallatech H101 Cluster

This cluster consists of 16 nodes connected by DDR InfiniBand and Gigabit Ethernet. Each node contains the following hardware:

- 3.2GHz Pentium 4 Intel Xeon EM64T Hyper-threading processor with 2MB L2 cache
- 2GB of registered-ECC DDR333 RAM
- Mellanox InfiniHost III Lx (MHGS18-XTC) PCI-E x8 DDR Infiniband card
- Nallatech H101-PCIXM Application Accelerator [74]
  - 133MHz PCI-X card with Xilinx XC4VLX100 user FPGA
  - 512MB DDR2 SDRAM (single bank)
  - 16MB DDR-II SRAM (across 4 banks)
  - 4 high-speed Serial I/O channels (2.5Gb/s each)
  - 25W power consumption (typical)

A.2 XtremeData XD1000

This system consists of a dual-processor motherboard containing a dual-core Opteron and an in-socket FPGA module [12].

- 2.6GHz Opteron 285
- 4GB DDR400 RAM
- XD1000 in-socket FPGA module (connected via HyperTransport)
  - Altera Stratix II EP2S180 FPGA
  - 4GB DDR RAM (128-bit, 333MHz)
  - 4MB ZBT SRAM (32-bit, 200MHz)
  - 32MB FLASH
A.3 GiDEL PROCStar III Cluster

This system consists of 24 compute nodes connected by DDR InfiniBand and Gigabit Ethernet. While this system includes a host CPU, the included GiDEL PROCStar III board [82] contains 4 tightly-coupled FPGAs and can be seen as representative of an embedded environment for this research, as the board can stand-alone without any CPU if needed. Each node contains the following hardware:

- Intel E5520 2.26GHz quad-core Xeon with QP
- 6GB DDR3 1333 RAM
- Two GiDEL PROCStar III boards, each containing...
  - Four Altera Stratix III E260 FPGAs
  - 4.25GB DDR memory per FPGA (three banks per FPGA: 256MB, 2GB, and 2GB)
  - High-speed direct-connection between neighboring FPGAs
  - PCIe interconnect (or card may stand alone if supplied external power)
APPENDIX B
APPLICATION CASE STUDIES

B.1 N-Queens

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. The RC benchmark was written by Vikas Aggarwal (HCS Lab, University of Florida), adapted by Gabe Barfield (HCS Lab, University of Florida), and then further adapted by John Curreri (CHREC, University of Florida). [75]

B.2 Time-Domain Finite Impulse Response

The time-domain finite impulse response (TDFIR) benchmark is part of the HPEC challenge benchmark suite [84]. The RC benchmark performs convolution for real numbers rather than complex. The initial RC implementation of this benchmark was written by Dr. Gregory Stitt (CHREC, University of Florida). [84]

B.3 Collatz Conjecture

The Collatz conjecture is a number theoretic conjecture involving sequences of integers. This application searches large ranges of natural numbers in an attempt to find record-length sequences of numbers that do not terminate at the trivial cycle containing the number 1. The algorithm employs parallel kernels which process 96-bit integers, computing sequences using heavily pipelined custom logic as well as logic to skip the initial part of most sequences. While this application is not readily used for practical purposes, it contains patterns that closely resemble cryptanalysis (i.e., a large number space is pre-filtered by a CPU, the FPGA performs a large parallel search, and finally the CPU collects and performs post-processing on returned numbers) [78], [79]. This application is also a good example of a non-deterministic RC application that makes heavy use of both CPUs and FPGAs. This application was written by the author. [77]
B.4 2-D Probability Density Function Estimation

The 2-D probability density function (2DPDF) estimation algorithm is employed in various engineering, financial, and scientific fields where non-parametric probabilistic approaches are required. The application uses a Parzen-window algorithm to estimate two-dimensional probability density functions via an exhaustive permutation of data between two vectors in a replicated, pipelined design. The complexity of this algorithm is $O(m \times n^2)$ where $m$ is the number of sample points and $n$ is the number of bins per dimension. The initial RC implementation of this application was written by Dr. Karthik Nagarajan (CHREC, University of Florida). [85]
REFERENCES


BIOGRAPHICAL SKETCH

Seth Koehler is a Ph.D. graduate from the Department of Computer Information Science and Engineering at the University of Florida. He received two B.S. degrees in computer engineering and mathematics from the University of Florida in 2003. His research focuses on the development of performance analysis and verification concepts and tools for reconfigurable systems. Additional interests include algorithms, number theory, numerical analysis, and game theory.