This dissertation is dedicated to my father, Dr. Peter Curreri.
ACKNOWLEDGMENTS

This work was supported in part by the industry and university cooperative research (I/UCRC) program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by Aldec, Altera, GiDEL, Impulse Accelerated Technologies, SRC Computers, Inc. and XtremeData, Inc. The authors would also like to acknowledge University of Washington Adaptive Computing Machines and Emulators (ACME) Lab for an XD1000 version of the backprojection application that was ported to Novo-G.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>8</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>9</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>11</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>13</td>
</tr>
<tr>
<td>2 BACKGROUND AND RELATED RESEARCH</td>
<td>16</td>
</tr>
<tr>
<td>2.1 Reconfigurable Computing</td>
<td>16</td>
</tr>
<tr>
<td>2.2 High-Level Synthesis</td>
<td>17</td>
</tr>
<tr>
<td>2.3 Performance Analysis of Parallel Applications</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Field-Programmable Gate Array Performance Monitoring</td>
<td>18</td>
</tr>
<tr>
<td>2.5 Automatic Analysis and Visualization</td>
<td>20</td>
</tr>
<tr>
<td>2.6 Assertion-Based Verification</td>
<td>20</td>
</tr>
<tr>
<td>3 PERFORMANCE ANALYSIS FRAMEWORK FOR HIGH-LEVEL SYNTHESIS</td>
<td>23</td>
</tr>
<tr>
<td>3.1 High-Level Synthesis Performance Analysis Challenges</td>
<td>24</td>
</tr>
<tr>
<td>3.1.1 Instrumentation Challenges</td>
<td>25</td>
</tr>
<tr>
<td>3.1.1.1 Instrumentation levels</td>
<td>25</td>
</tr>
<tr>
<td>3.1.1.2 Instrumentation selection</td>
<td>26</td>
</tr>
<tr>
<td>3.1.2 Measurement Challenges</td>
<td>28</td>
</tr>
<tr>
<td>3.1.2.1 Measurement techniques</td>
<td>28</td>
</tr>
<tr>
<td>3.1.2.2 Measurement data extraction</td>
<td>29</td>
</tr>
<tr>
<td>3.1.3 Analysis Challenges</td>
<td>30</td>
</tr>
<tr>
<td>3.1.4 Visualization Challenges</td>
<td>31</td>
</tr>
<tr>
<td>3.2 High-Level Synthesis Performance Analysis Framework</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1 Instrumentation Automation</td>
<td>35</td>
</tr>
<tr>
<td>3.2.1.1 Automated C source instrumentation</td>
<td>35</td>
</tr>
<tr>
<td>3.2.1.2 Automated hardware description language instrumentation</td>
<td>36</td>
</tr>
<tr>
<td>3.2.2 Performance Analysis Tool Integration</td>
<td>39</td>
</tr>
<tr>
<td>3.2.2.1 Performance data extraction</td>
<td>39</td>
</tr>
<tr>
<td>3.2.2.2 Performance data visualization</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Molecular-Dynamics Case Study</td>
<td>42</td>
</tr>
<tr>
<td>3.4 Conclusions</td>
<td>46</td>
</tr>
<tr>
<td>4 COMMUNICATION BOTTLENECK DETECTION AND VISUALIZATION</td>
<td>50</td>
</tr>
<tr>
<td>4.1 Communication Performance Analysis</td>
<td>52</td>
</tr>
</tbody>
</table>
4.1.1 Instrumentation and Measurement ............................................. 52
4.1.2 Visualizations ........................................................................... 55
4.1.3 Analysis .................................................................................... 57
4.2 Experimental Framework ................................................................. 58
4.2.1 Instrumentation ........................................................................ 58
4.2.2 Visualization ............................................................................ 60
4.3 Experimental Results ..................................................................... 60
4.3.1 Triple Data Encryption Standard (DES) ...................................... 61
4.3.2 Molecular Dynamics .................................................................. 61
4.3.3 Backprojection .......................................................................... 63
4.4 Conclusions .................................................................................... 65

5 ASSERTION-BASED VERIFICATION FOR HIGH-LEVEL SYNTHESIS .... 67
5.1 Assertion Synthesis and Optimizations ............................................. 69
5.1.1 Assertion Parallelization .............................................................. 71
5.1.2 Resource Replication ................................................................. 72
5.1.3 Resource Sharing ...................................................................... 74
5.2 In-Circuit Timing-Analysis Assertions ............................................ 75
5.3 Hang-Detection Assertions ............................................................... 78
5.4 Assertion Framework ..................................................................... 82
5.4.1 Unoptimized Assertion Framework ........................................... 82
5.4.2 Assertion Framework Optimizations .......................................... 84
5.4.3 Timing-Analysis and Hang-Detection Extensions ....................... 85
5.4.4 High-Level Synthesis Tool and Platform .................................... 86
5.5 Experimental Results ..................................................................... 87
5.5.1 Detecting Simulation Inconsistencies .......................................... 87
5.5.2 Assertion Parallelization Optimization ....................................... 88
5.5.2.1 Triple Data Encryption Standard (DES) case study .............. 89
5.5.2.2 Edge-detection Case Study ................................................... 89
5.5.2.3 State Machine Overhead Analysis ....................................... 90
5.5.3 Resource Replication Optimization .......................................... 92
5.5.4 Resource Sharing Optimization ................................................ 93
5.5.5 In-circuit Timing Analysis ......................................................... 95
5.5.6 Hang Detection ......................................................................... 99
5.5.7 Assertion Limitations ............................................................... 100
5.6 Conclusions .................................................................................. 102

6 CONCLUSIONS ................................................................................ 104

APPENDIX

A TARGET SYSTEMS .......................................................................... 107
A.1 ProcStar-III .................................................................................. 107
A.2 XD1000 ....................................................................................... 107
A.3 SRC-7 .......................................................................................... 107
B APPLICATION CASE STUDIES .............................................. 108

B.1 Molecular Dynamics ......................................................... 108
B.2 Backprojection .................................................................. 108
B.3 Triple Data Encryption Standard (DES) ............................. 108
B.4 Edge Detection ................................................................. 108

REFERENCES .......................................................................... 109

BIOGRAPHICAL SKETCH .......................................................... 115
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>Performance analysis overhead</td>
<td>47</td>
</tr>
<tr>
<td>4-1</td>
<td>Triple Data Encryption Standard (DES) instrumentation overhead</td>
<td>64</td>
</tr>
<tr>
<td>5-1</td>
<td>Triple Data Encryption Standard (DES) assertion overhead</td>
<td>90</td>
</tr>
<tr>
<td>5-2</td>
<td>Edge-detection assertion overhead</td>
<td>91</td>
</tr>
<tr>
<td>5-3</td>
<td>Single-comparison assertion</td>
<td>93</td>
</tr>
<tr>
<td>5-4</td>
<td>Pipelined single-comparison assertion</td>
<td>93</td>
</tr>
<tr>
<td>5-5</td>
<td>Individual backprojection timing assertion overhead</td>
<td>98</td>
</tr>
<tr>
<td>5-6</td>
<td>Looped backprojection timing assertion overhead</td>
<td>99</td>
</tr>
<tr>
<td>5-7</td>
<td>Data Encryption Standard (DES) hang-detection overhead</td>
<td>101</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Simplified block diagram of a Field-Programmable Gate Array</td>
<td>17</td>
</tr>
<tr>
<td>2-2</td>
<td>Performance analysis steps</td>
<td>19</td>
</tr>
<tr>
<td>3-1</td>
<td>Example performance visualization</td>
<td>32</td>
</tr>
<tr>
<td>3-2</td>
<td>Hardware Measurement Module (HMM) addition and application instrumentation</td>
<td>36</td>
</tr>
<tr>
<td>3-3</td>
<td>Design flow for Impulse C performance analysis</td>
<td>37</td>
</tr>
<tr>
<td>3-4</td>
<td>Hardware Measurement Module (HMM)</td>
<td>38</td>
</tr>
<tr>
<td>3-5</td>
<td>Profile tree table visualization for Triple Data Encryption Standard (DES)</td>
<td>40</td>
</tr>
<tr>
<td>3-6</td>
<td>Molecular-Dynamics (MD) hardware subroutine</td>
<td>43</td>
</tr>
<tr>
<td>3-7</td>
<td>Accelerator process source with profiling percentages</td>
<td>44</td>
</tr>
<tr>
<td>3-8</td>
<td>Pie chart visualization of the states in Accelerator process. State b5s0 correspond to the pipeline and state b6s2 corresponds to the output stream.</td>
<td>45</td>
</tr>
<tr>
<td>3-9</td>
<td>Bar chart visualization depicting MD kernel performance improvement due to increased stream buffer size (128 to 4096 elements).</td>
<td>46</td>
</tr>
<tr>
<td>3-10</td>
<td>Output stream overhead for the Accelerator process.</td>
<td>47</td>
</tr>
<tr>
<td>4-1</td>
<td>Parallel Performance Wizard (PPW) data transfer visualization</td>
<td>51</td>
</tr>
<tr>
<td>4-2</td>
<td>Streaming-communication call visualization</td>
<td>56</td>
</tr>
<tr>
<td>4-3</td>
<td>Direct memory access (DMA) communication call visualization</td>
<td>56</td>
</tr>
<tr>
<td>4-4</td>
<td>Streaming buffer analysis</td>
<td>58</td>
</tr>
<tr>
<td>4-5</td>
<td>Toolflow for Impulse C visualization</td>
<td>59</td>
</tr>
<tr>
<td>4-6</td>
<td>Communication visualization of streaming DES</td>
<td>62</td>
</tr>
<tr>
<td>4-7</td>
<td>Communication visualization of DES using DMA transfer</td>
<td>63</td>
</tr>
<tr>
<td>4-8</td>
<td>Molecular-dynamics bandwidth visualization (half of FPGA cropped to enlarge image)</td>
<td>64</td>
</tr>
<tr>
<td>4-9</td>
<td>Communication visualization of Backprojection</td>
<td>65</td>
</tr>
<tr>
<td>5-1</td>
<td>Assertion framework</td>
<td>70</td>
</tr>
<tr>
<td>5-2</td>
<td>Application’s state machine</td>
<td>73</td>
</tr>
</tbody>
</table>
5-3 Using timing-analysis assertions with a filter application .......................... 77
5-4 Manually using American National Standards Institute C (ANSI-C) assertions for hang detection ................................................................. 79
5-5 Using hang-detection assertions with a filter application ......................... 80
5-6 High-Level Language (HLL) assertion instrumentation .......................... 83
5-7 In-circuit verification example .................................................................. 88
5-8 Simple streaming loopback ......................................................................... 94
5-9 Assertion frequency scalability .................................................................. 95
5-10 Optimized assertion resource scalability ................................................. 96
5-11 Adding timing assertions individually to backprojection ......................... 97
5-12 Adding timing assertions in a loop to backprojection .............................. 97
PERFORMANCE ANALYSIS AND VERIFICATION FOR HIGH-LEVEL SYNTHESIS

By

John A. Curreri

May 2011

Chair: Alan D. George and Greg Stitt
Major: Electrical and Computer Engineering

High-Level Synthesis (HLS) for Field-Programmable Gate Arrays (FPGAs) facilitates the use of Reconfigurable Computing (RC) resources for application developers by using familiar, higher-level syntax, semantics, and abstractions. Thus, HLS typically enables faster development times than traditional Hardware Description Languages (HDLs). However, this higher level of abstraction is typically not maintained throughout the design process. Once the program is translated from source code to an HDL, analysis tools provide results at the HDL level. The research performed in this document focus on providing higher-level analysis of application behavior throughout the design process to increase developer productivity. Just as knowledge of assembly code is no longer required for most programmers of microprocessors today, the goal of this work is to evaluate methods that can eliminate the need for programmers to understand HDLs in order to develop a high-performance error-free application for FPGA platforms.

This document is divided into three phases. Phase one addresses the challenges associated with prototyping an in-circuit performance analysis framework for HLS applications while leveraging existing visualizations. Phase two focuses on the challenges of detecting communication bottlenecks and creates a novel visualization designed specifically for HLS and FPGA systems. Phase three addresses the challenges associated with in-circuit integration of assertion-based verification into HLS. The performance analysis and verification frameworks presented in this document are believed to be
(after extensive literature review) the first of their kind for HLS. Case studies using various FPGA platforms and HLS tools show the utility and low overhead of these frameworks. The frameworks explored by this research will help lay the foundation for future performance analysis and verification research and tools targeting HLS.
CHAPTER 1
INTRODUCTION

Reconfigurable Computing (RC) which typically uses Field-Programmable Gate Arrays (FPGAs) as processors has shown favorable energy efficiency advantages as compared to other processors for High-Performance Computing (HPC) and High-Performance Embedded Computing (HPEC) [1]. However, the adoption of FPGAs for application acceleration has been hampered by design-productivity issues, since developing applications using a low-level Hardware Description Language (HDL) can be a time-consuming process. In order to address this problem, High-Level Synthesis (HLS) tools have been developed to allow application design using High-Level Languages (HLL) such as C. Unfortunately, many of the tools needed to develop a correctly functioning, high-performance application are absent from HLS tools. HLS tools lack performance analysis tools to measure the performance of computation and communication of the application at runtime. There is also a lack of verification tools to check the correctness, real-time constraints or the origin of hangs of the application at runtime. The frameworks and tools presented in this dissertation allow the user to perform performance analysis and verification on their application at runtime while having a low resource and performance overhead, thus boosting developer productivity and the competitiveness of FPGA-based solutions.

The research in this document is divided into three phases. In Phase 1, several challenges are explored and addressed when expanding upon an existing performance analysis framework (ReCAP) [2] to support HLS tools. New instrumentation techniques are explored for HLS code. Methods for performance data extraction are also explored as well as methods to present performance data in the context of C source code. Performance visualizations are leveraged from an existing performance analysis tool called Parallel Performance Wizard (PPW) [3] to provide seamless performance analysis across both microprocessors and FPGAs. The performance analysis framework for HLS is evaluated through a case study showing its ability to speedup applications while having
a low-resource overhead. The goal of Phase 1 is to address the challenges associated with prototyping an in-circuit performance analysis tool for HLS that is familiar to parallel programmers who typically target microprocessors.

Phase 2 focuses on the challenges of detecting communication bottlenecks and creating novel performance visualizations for HLS. New instrumentation is explored to measure the average bandwidth of communication channels in the FPGA. Research is performed to explore the techniques and methods needed to automatically determine the communication bottlenecks. A visualization in the form of a directed graph is prototyped for manual analysis. These new HLS bottleneck detection and visualization techniques are validated via case studies. The goal of Phase 2 is to further increase design productivity by evaluating novel HLS-specific visualization techniques that enable bottleneck detection.

In the third phase, the challenges associated with prototyping assertion-based verification for HLS designs are explored and addressed. Techniques needed to perform verification of HLS code are explored in circuit as opposed to using simulation. Tradeoffs are explored that reduce FPGA resource overhead and runtime performance overhead. The effectiveness of this framework is demonstrated by showing its ability to find verification failures not caught in HLS simulation. Novel enhancements to assertion-based verification are explored to allow real-time deadline checking and hang detection. The goal of Phase 3 is to address the challenges associated with prototyping a low overhead assertion-based verification tool for HLS that is familiar and useful to parallel programmers that have not used FPGAs.

The rest of this document is organized as follows. Chapter 2 discusses background and related research. Chapter 3 describes the performance analysis framework for HLS (Phase 1). Chapter 4 explains the extended performance analysis framework that incorporates visualizations for bottleneck detection (Phase 2). Chapter 5 gives details about the framework for in-circuit assertion-based verification of HLS (Phase 3).
Finally, Chapter 6 provides concluding remarks. A brief description of target systems and application case studies can be found in Appendix A and B, respectively.
CHAPTER 2
BACKGROUND AND RELATED RESEARCH

The background and related research in this chapter is divided into six sections. Section 2.1 provides background on reconfigurable computing. Section 2.2 explains the productivity gain from using High-Level Synthesis tools. Section 2.3 covers the steps used to perform performance analysis on a parallel application along with a brief survey of microprocessor-based parallel performance analysis tools. Section 2.4 discusses FPGA performance monitoring and the lack of in-circuit FPGA performance analysis tools. Section 2.5 describes related work in automatic analysis and visualization. Section 2.6 gives details on related work in assertion-based verification.

2.1 Reconfigurable Computing

This section provides a brief background for Reconfigurable Computing (RC). More detailed surveys of RC can be found in Compton and Hauck [4] or Hartenstein [5]. RC systems have the ability to change the organization of their hardware to best suit a particular application. RC systems are most commonly built with FPGAs. Unlike microprocessors which are programmed via instructions, FPGAs are programmed by connecting various hardware elements together. Figure 2-1 shows a simplified block diagram of the most common FPGA hardware elements. The look-up tables inside an FPGA can be programmed to provide algebraic or logic operations needed by the program while routing switches connect look-up tables together to create more complex operations. Flip-Flops are used to hold values during each clock cycle. FPGAs also contain memory in the form of block RAMs to store data too large to hold with registers. Input-output ports facilitate off-chip communication. For a more detailed description of an FPGA architecture, refer to Altera’s Stratix handbook [6] or Xilinx’s Virtex user guide [7].

FPGAs can be integrated into computing systems at multiple levels. The XD1000 [8] integrates an FPGA in one of the two Opteron sockets on a dual-processor motherboard. The SRC-7 [9] uses a riser board plugged into the memory slot to communicate with
Figure 2-1. Simplified block diagram of a Field-Programmable Gate Array

the CPU. The GiDEL ProcStar-III [10] is a PCI Express card that allows FPGAs to
be accessed as a peripheral device. The Cray XD1 [11] uses a custom Rapid Array
interconnect to allow microprocessors, FPGAs and other nodes to communicate with each
other. The large variety of RC architectures and applications can pose a challenge for
general purpose verification and performance analysis tools.

2.2 High-Level Synthesis

This section provides a brief background for HLS. A survey of HLS tools can be found
in Holland et al. [12] and a productivity analysis of HLS tools can be found in El-Araby
et al. [13]. Traditionally, circuit designers program FPGAs using HDLs such as VHDL
or Verilog. HDLs are inherently able to express parallelism because the components that
make up a circuit work in parallel. However, software programmers tend to find HDLs
cumborsome because they use clock signals to control the timing of the application. HLS
tools allow applications to be designed using graphical techniques or with an HLL such
as C, Java, or Fortran. HLL-based HLS tools typically have Application Programming
Interface (API) calls to increase parallelism or add synchronization and communication to
an application. Although the higher level of abstraction provided by HLS tools increases
developer productivity when writing applications, productivity with current tools is then
lowered by the low-level abstraction of accurate verification and performance analysis tools for HLS-generated HDL code.

2.3 Performance Analysis of Parallel Applications

Performance analysis can be divided into five steps (as derived from Maloney’s work on the TAU performance analysis framework for traditional processors [14]) whose end goal is to produce an optimized application. These steps are Instrument, Measure, Analyze, Present, and Optimize (see Figure 2-2). The instrumentation step inserts the necessary code (i.e., additional hardware in the FPGA’s case) to access and record application data at runtime, such as variables or signals to capture performance indicators. Measurement is the process of recording and storing the performance data at runtime while the application is executing. After execution, analysis of performance data to identify potential bottlenecks can be performed. Some tools such as TAU can automatically analyze the measured data to help the user find potential bottlenecks, while other tools rely solely upon the developer to analyze the results. In either case, data is typically presented to the programmer via text, charts, or other visualizations to allow for further analysis. Finally, optimization is performed by modifying the application’s code or possibly changing the application’s platform based upon insights gained via the previous steps. Since automated optimization is an open area of research, optimization at present is typically a manual process. Finally, these steps may be repeated as many times as the developer deems necessary, resulting in an optimized application. This methodology is employed by a number of existing tools for software parallel performance analysis tools including PPW (Parallel Performance Wizard) [3], TAU (Tuning and Analysis Utilities) [14], KOJAK (Kit for Objective Judgment And Knowledge-based detection of performance bottlenecks) [15], HPCToolkit [16], and SvPablo [17].

2.4 Field-Programmable Gate Array Performance Monitoring

To the best of our knowledge after a comprehensive literature search, little previous work exists outside this research concerning runtime performance analysis for HLS
Figure 2-2. Performance analysis steps

applications. Hardware performance measurement modules have been integrated into FPGAs before; however, they were designed specifically for monitoring the execution of soft-core processors [18]. The Owl framework, which provides performance analysis of system interconnects, uses FPGAs for performance analysis, but does not actually monitor the performance of hardware inside the FPGA itself [19]. Range adaptive profiling has been prototyped on FPGAs but was not used for profiling an application executing on an FPGA [20]. Runtime debugging of an HLS tool, Sea Cucumber, has been developed by Hemmert et al. [21]. However, the Sea Cucumber debugger does not support performance analysis. Calvez et al. [22] describe performance analysis for Application-Specific Integrated Circuits (ASICs) while DeVille et al. [23] discuss performance monitoring probes for FPGA circuits; however, neither work targets HLS tools. This work significantly extends our previous work on performance analysis for HDL applications [2] by expanding this HDL framework to address the challenges of high-level synthesis tools.
2.5 Automatic Analysis and Visualization

Automatic analysis has been developed in performance analysis tools such as TAU [14], KOJAK [15], PPW [3], and Paradyn [24]. TAU uses ParaProf [25] for profile analysis. KOJAK and TAU use the Expert performance analysis tool [26] to perform trace pattern matching. PPW uses an automatic analysis system [27] developed by Su et al.

A summary of early visualizations from Kiviat diagrams to 3D isosurface displays is described by Heath et al. [28]. ParaGraph [29], a performance analysis tool for MPI programs, utilizes many of these early visualizations. Miller wrote an essay [30] on effective techniques and pitfalls of performance visualizations. A modern tool, TAU, uses Vampir [31], Jumpshot [32], and Paraver [33] for trace visualization. Knupfer et al. describe an extension to Vampir for visualization of patterns in trace data [34]. KOJAK has a 3D load-balancing case study of an n-body simulation [35]. Santamara et al. give an overview of graphing techniques for visualizing networks [36]. Haynes et al. [37] give an example of a visualization of network performance in a computing cluster. Little work on visualizations of FPGA performance was found in an extensive literature review. DRIVE [38] provides simulation and visualization for dynamic FPGA reconfiguration. The sample visualization presented in their paper shows blocks of an FPGA inactive, active, or reconfiguring. Their framework only provides simulation and does not provide in-circuit monitoring. Related work exists for performance analysis [2] and visualization [39] of HDL applications for FPGAs. However, that work is ill-suited for HLS applications due to lack of source-code correlation and bandwidth visualizations.

2.6 Assertion-Based Verification

Many languages and libraries enable assertions in HDLs during simulation, such as VHDL assertion statements, SystemVerilog Assertions (SVA) [40], the Open Verification Library (OVL) [41], and the Property Specification Language (PSL) [42]. Previous work has also introduced in-circuit assertions via hardware assertion checkers for each assertion in a design. Tools targeted at ASIC design provide assertion checkers using SVA [43],

20
PSL [44], and OVL [45]. Academic tools such as Camera’s debugging environment [46] and commercial tools such as Temento’s DiaLite also provide assertion checkers for HDL. Kakoe et al. show that in-circuit assertions [45] can also improve reliability, with a higher fault coverage than Triple Modular Redundancy (TMR) for a FIR filter and a Discrete Cosine Transform (DCT).

Logic analyzers such as Xilinx’s ChipScope [47] and Altera’s SignalTap [48] can also be used for in-circuit debugging. These tools can capture the values of HDL signals and extract the data using a JTAG cable. However, the results presented by these tools are not at the source level of HLS tools. A source-level debugger has been built for the Sea Cucumber synthesizing compiler [21] that enables breakpoints and monitoring of variables in FPGAs. Our work is complementary by enabling HLL assertions and can be potentially be used with any HLS tool.

Checking timing constraints of HDL applications can be performed with many of the methods mentioned above. SVA, PSL and OVL assertions can be used to check the timing relationship between expected values of signals in an HDL application [49]. A timed C-like language, TC (timed C), has been developed for checking OVL assertions inserted as C comments for use during modeling and simulation [50]. In-circuit logic analyzers such as ChipScope [47] and SignalTap [48] can also be used to trace application signals and check timing constraints for signal values. The HLS tool, Carte, provides timing macros [51] which return the value of a 64-bit counter that is set to zero upon FPGA reset. However, most HLS tools (including Impulse C) do not provide this functionality. In-circuit implementation of high-level assertions is a more general approach that potentially supports any HLS tool and enables designers to use ANSI-C assertions.

After a comprehensive literature search, we found no previous work related to hang detection of HLS applications. Hang detection for microprocessors has been implemented on FPGAs [52]. Nakka et al. [53] separates hang detection for microprocessors into three categories. First, Instruction-Count Heartbeat (ICH) detects a hung process not executing
any instructions. Second, Infinite-Loop Hang Detector (ILHD) detects a process which
never exits a loop. Finally, Sequential-Code Hang Detector (SCHD) detects a process that
never exits a loop because the target address for the completion of a loop is corrupted.
Although similar detection categories could be used for hardware processes generated
by HLS tools, the methods needed for hang detection are different; hardware processes
typically use state machines for control flow rather than using instructions. The related
work found for microprocessor hang detection is typically used to increase reliability of the
system by terminating the hung process rather than to help an application developer find
the problematic line of code.

Although HDL assertions could be integrated into HLS-generated HDL, such an
approach has several disadvantages. Any changes to the HLL source or a different version
of the HLS tool could cause changes to the generated HDL (e.g., reorganization of code
or renaming of signals), which requires the developer to manually reinsert the assertions
into the new HDL. It is also possible that the developer may not be able to program in
HDL or the HLS tool may encrypt or obfuscate generated HDL (e.g., LabVIEW-FPGA).
HLL assertions for HLS avoid these problems by adding assertions at the source level.
Specifically, ANSI-C [54] assertions were chosen to be synthesized to hardware since they
are a standard assertion widely used by software programmers. Synthesizing ANSI-C
assertions would allow existing assertions already written for software programs to be
checked while running in circuit.
CHAPTER 3
PERFORMANCE ANALYSIS FRAMEWORK FOR HIGH-LEVEL SYNTHESIS

High-level synthesis tools translate high-level languages (e.g., Impulse C [55] or Carte C [56]) to hardware configurations on FPGAs. Today's HLLs simplify software developers transition to reconfigurable computing and its performance advantages without the steep learning curve associated with traditional HDLs. While HDL developers have become accustomed to debugging code via simulators, software developers typically rely heavily upon debugging and performance analysis tools. In order to accommodate the typical software development process, HLS tools support debugging at the HLL source-code level on a traditional microprocessor without performing translation from source to HDL. Current commercial HLS tools provide few (if any) runtime tools (i.e., while the application is executing on one or more FPGAs) for debugging or performance analysis at the source-code level. In addition, research on runtime performance analysis for FPGAs is lacking with few exceptions and none of which is targeted towards HLS tools.

While it is possible for debugging and simulation techniques to estimate basic performance, many well-researched debugging techniques may not be suited for performance analysis. For example, halting an FPGA to read back its state will cause the FPGA to become temporarily inaccessible from the CPU, potentially resulting in performance problems that did not exist before. Thus, this approach is not viable due to the unacceptable level of disturbance caused to the application's behavior and timing. Alternatively, performance can be analyzed through simulation. However, cycle-accurate simulations of complex designs on an FPGA are slow and increase in complexity as additional system components are added to the simulation. Most (if not all) cycle-accurate simulators for FPGAs focus upon signal analysis and do not present the results at the source-code level to a software developer.

RC applications have potential for high performance, but HLS-based applications can fall far short of that potential due to the layer of abstraction hiding much of the
implementation (and thus performance) details. Performance analysis tools can aid the developer in understanding application behavior as well as in locating and removing performance bottlenecks. Due to the HLL abstraction layer, it is essential for performance analysis to provide performance data at that same level, allowing correlation between performance data and source line.

This work focuses upon performance analysis of an HLS-based application on a reconfigurable system by monitoring the application at runtime. The majority of insight gained was from performance analysis with high-level languages from experiences with Impulse C, a language designed by Impulse Accelerated Technologies, which maps a reduced set of C statements to HDL; however insights gained from Carte C, a language designed by SRC Computers, are also discussed to provide an alternate perspective of HLL performance analysis. A performance analysis framework was developed based on Impulse C and this framework was prototyped into an automated tool in order to demonstrate its effectiveness on a molecular-dynamics application.

This chapter is organized as follows. Section 3.1 covers the challenges of performance analysis for HLL. Next, Section 3.2 provides details about the performance analysis framework for Impulse C. Section 3.3 then presents a case study using a molecular-dynamics kernel written in Impulse C. Finally, Section 3.4 concludes and presents ideas for future work.

3.1 High-Level Synthesis Performance Analysis Challenges

While all stages of performance analysis mentioned above are of interest for HLS-based applications, discussion in this chapter is limited to the challenges of instrumentation, measurement, analysis, and visualization; optimization is beyond the scope of this work. Thus, Section 3.1.1 covers the challenges of instrumenting an application, Section 3.1.2 explains the challenges associated with measuring performance data from an application, Section 3.1.3 discusses the challenges of analyzing performance data, and Section 3.1.4 examines the challenges of visualizing performance data.
3.1.1 Instrumentation Challenges

Instrumentation, the first step of performance analysis, enables access to application data at runtime. For HLS-based applications, this step raises two key issues: at what level of abstraction should modifications be made, and how to best select what should be accessed to gain a clear yet unobtrusive view of the application’s performance. Tradeoffs concerning the level of abstraction are discussed in Section 3.1.1.1, while the selection of what to monitor is covered in Section 3.1.1.2.

3.1.1.1 Instrumentation levels

Three main instrumentation levels have been investigated: HLL software, HLL hardware and HDL. Each instrumentation level offers advantages to a performance analysis tool for HLS-based applications. For details on instrumentation levels below HDL, see Graham et al. [57].

The most obvious choice for instrumentation is to directly modify the HLL source code. Instrumentation can be added to the software code requiring timing to be handled by the CPU. Each timing call is sent from FPGA to CPU over a communication channel. This is currently used by Impulse C developers since no HLL hardware timing functions are available. For a small number of coarse-grained measurements (e.g., for phases of the hardware application), the communication overhead and timing granularity are acceptable.

Instrumentation can also be added to the HLL source code that describes FPGA hardware. Most high-level synthesis tools lack this feature. Carte C is an exception in that it allows the developer to manually control and retrieve cycle counters, which, along with the FPGA’s clock frequency and some adjustment for skew, provides accurate timing information between the CPU and FPGA. The main advantage of this method is simplicity; code is added to record data at runtime, and this data can be easily correlated with the source line that was modified. It is also possible that the HLL source code may be the only level that can be instrumented (e.g., if encrypted net-lists and bitstreams are employed).
Instrumentation can also be inserted after the application has been mapped from HLL to HDL. Instrumentation of VHDL or Verilog provides greater flexibility than instrumentation at the HLL level since measurement hardware can be fully customized to the application’s needs, rather than depending upon built-in HLL timing functions. Adding instrumentation after the HLL-to-HDL mapping guarantees that measurement hardware will run in parallel with the hardware being timed, minimizing the effect of measurement on the application’s performance and behavior. In contrast, Carte C introduces delays into a program when its timing functions are used. However, using instrumentation below the HLL source level does require additional effort to map information gathered at the HDL level back to the source level. This process is problematic due to the diversity of mapping schemes and translation techniques employed by various high-level synthesis tools and even among different versions of the same tool. For example, if a performance tool relies upon a textual relation between HLL variables and HDL signals, then the performance tool would fail if this naming scheme was modified in a subsequent release of the high-level synthesis tool.

While the simplicity of HLL instrumentation is desirable, the HDL level was instrumented in order to provide fine-grained performance analysis that would otherwise be impossible for HLS tools that lack hardware timing functions. Even for HLS tools that do provide hardware timing functions, HDL instrumentation may incur less overhead and generally provides greater flexibility than HLL instrumentation. HDL instrumentation is also utilized by FPGA logic analyzers such as Xilinx’s ChipScope [47] or Altera’s SignalTap [48].

3.1.1.2 Instrumentation selection

Application performance can generally be considered in terms of communication and computation. Many HLS tools, such as Impulse C and Carte C, have built-in functions for communication; these functions typically have associated status signals at the HDL level that can be instrumented to determine usage statistics such as transfer rate or idle time.
Instrumenting computation is more complex due to the various ways that computation can be mapped to hardware. However, these mappings are constrained by the fact that each high-level synthesis tool must preserve the semblance of program order and thus will require some control structure to manage this ordering. For example, Impulse C maps computation onto (possibly multi-level) state machines, using the top-level state machine to provide high-level ordering of program tasks. For Carte C, computation is mapped to code blocks that activate each other using completion signals. While these control structures are useful for coarse-grained timing information, additional information can be obtained from substates within a single state of the top-level state machine or signals within a code block, which are used, for example, to control a single loop that has been pipelined. In Impulse C, this pipeline would consist of the idle, initialize, run, and flush substates, where the initialize and flush substates indicate pipelining overhead and thus provide indication of lost performance. Additionally, signals such as stall, break, write, and continue can be instrumented on a per-pipeline-stage basis to obtain even more details if needed. For Carte C, less detail is available since pipelined loops are not broken up into explicit stages and state machines are not exposed for instrumentation. Nonetheless, intermediate signals connecting Carte C’s hardware macros inside a code block can be instrumented, which provide the necessary information to determine pipelined loop iterations and stalls. Overall, it is the control structures employed to maintain program order that provide key data for monitoring performance of these applications.

It may also be beneficial to monitor application data directly (i.e., an HLL variable) if such information provides a better understanding of application performance and behavior. For example, a loop control variable may be beneficial to monitor if it represents the progress of an algorithm. Unfortunately, selection of an application variable is, in general, not automatable due to the need for high-level, application-specific knowledge to understand the variable’s purpose and expected value.
Instrumentation of state machines was chosen for Impulse C and completion signals for Carte C since they provide timing data similar to software profilers. Since these control structures are needed to preserve the order of execution of the HLL, they should be targeted for automatic instrumentation.

When comparing Impulse C and Carte C, it is evident that instrumentation is the primary step of performance analysis that requires change for a new HLS tool. The remaining steps can remain basically unchanged as long as the designer is focused primarily on the timing of HLL source code and that reverse mapping is performed.

### 3.1.2 Measurement Challenges

After instrumentation code has been inserted into the developer’s application, monitored values must be recorded (measured) and sent back to the host processor. Section 3.1.2.1 presents standard techniques for measuring application data while Section 3.1.2.2 discusses the challenges of extracting measurement data.

#### 3.1.2.1 Measurement techniques

Regardless of the programming language used, the two common modes for measuring performance data are profiling and tracing. Profiling records the number of times that an event has occurred, often using simple counters. To conserve the logic resources of an FPGA, it is possible to store a larger number of counters in block RAM if it can be guaranteed that only one counter within a block RAM will be updated each cycle. This technique is useful for large state machines, since they can only be in one state at any given clock cycle. Profiling data can be collected either when the program is finished (post-mortem) or sampled (collected periodically) during execution. At the cost of communication overhead, sampling can provide snapshots of profile data at various stages of execution that would otherwise be lost by a post-mortem retrieval of performance data.

In contrast, tracing records timestamps indicating when individual events occurred and, optionally, any data associated with each event. Due to the potential for generating large amounts of data, trace records typically require a buffer for temporary storage (e.g.,
Block RAM) until they can be offloaded to a larger memory, such as the host processor’s main memory. While logic resources in the FPGA can also be used for trace data storage, this resource is scarce and of lower density than block RAM, making logic resources ill-suited for general trace data. If available, other memory resources such as larger, preferably on-board SRAM or DRAM can be used to store trace data as well before it is sent to the host processor. Tracing does provide a more complete picture of application behavior, capturing the sequence and timing of events. Thus, when needed, tracing can be justified despite the often high memory and communication overhead. The challenges and techniques associated with measurement for HLLs are similar to those of HDLs [2]. Therefore, their profiling and tracing measurement techniques are used in this framework.

3.1.2.2 Measurement data extraction

Measurement data gathered in the FPGA typically is transferred to permanent storage for analysis. This data is commonly first buffered in large, lower-latency memories while awaiting transfer. FPGA logic analyzers such as Xilinx’s ChipScope or Altera’s SignalTap use JTAG [58] as an interface for extracting measured data in order to debug hardware. However, while a JTAG interface is available on many FPGA computing platforms, it is not well suited towards data extraction for runtime performance analysis. JTAG is a low-bandwidth serial communication interface and, in an HPC environment, the setup of all required JTAG cables coupled with the possible need to add additional hardware in order to receive the JTAG data is cumbersome and scales poorly.

As an alternative to JTAG, many HLS tools use communication interfaces to transfer data between the CPU and FPGA. In order to extract measurement data, an additional communication channel can be added to the application’s source code. Using these built-in interfaces is advantageous since no change to the physical system is required to support performance analysis. Thus, data extraction was chosen using HLL communication channels since it is more portable and better suited for typical HPC environments.
However, since the HLL communication channel is shared with the application, care must be taken not to disturb the application’s original behavior.

Communication overhead can depend upon several factors. One major factor concerns how much data is generated. Profile counters and trace buffers should be sized according to the number of events expected (with some margin of safety). Events should also be defined frugally to minimize the amount of data recorded while still obtaining the information needed to analyze performance. For example, while it may be ideal to monitor the exact time and number of cycles for all writes, it may be sufficient to know the number of writes exceeding a certain number of cycles.

Another source of overhead comes from the HLL’s communication interface. The bandwidth of streaming and memory-mapped communication interfaces can vary significantly between HLS tools as well as between FPGA platforms using the same tool, depending upon implementation. Therefore, it is important for performance analysis tools to support as many communication interfaces (e.g., streaming, DMA) as possible to provide flexibility and reduce overhead.

3.1.3 Analysis Challenges

While analysis of performance data has historically been very difficult to automate, automatic analysis can improve developer productivity by quickly locating performance bottlenecks. Automatic analysis of HLS-based applications could focus upon recognizing common performance problems such as potentially slow communication functions or idle hardware process. For example, processes replicated to exploit special parallelism can be monitored to determine which are idle and for what length of time, giving pertinent load-balancing information to the developer. Processes can also be replicated temporally in the form of a pipeline of processes and monitored for bottlenecks. High-level synthesis tools can also pipeline loops inside of a process, either automatically (e.g., Carte C) or explicitly via directed pragmas (e.g., Impulse C). In this case, automatic analysis would
determine how many cycles in the pipeline were unproductive and the cause of these problems (e.g., data not available, flushing of pipeline).

Automatic analysis can also be useful in determining communication characteristics that may cause bottlenecks, such as the rate or change in rate of communication. For example, streams that receive communication bursts may require larger buffers, or an application may be ill-suited for a specific platform due to a lack of bandwidth. The timing of communication can also be important; shared communication resources such as SRAMs often experience contention and should, in general, be monitored. Monitoring these communication characteristics can aid in the design of a network that keeps pipelines at peak performance. Integration of automatic analysis into the framework will be saved for future work. Further study on common reconfigurable computing bottlenecks in applications is needed in order to develop a general-purpose automatic analysis framework.

3.1.4 Visualization Challenges

One of the strengths of reconfigurable computing is that it allows the programmer to implement application-specific hardware. However, visualizations for high-performance computing are typically designed to show computation on general-purpose processors and communication on networks that allow all-to-all communication. Thus, these visualizations are ill-suited for HLS-based applications, treating heterogeneous components and communication architectures as homogeneous.

Koehler et al. [2] presented a mockup visualization for HDL performance analysis. This visualization was organized in the format of the system architecture and provides details on CPU usage, interconnect bandwidths, and FPGA state machine percentages. The visualization concepts presented by Koehler can be extended and presented in greater detail for HLS-based applications. HDL code generated by high-level synthesis tools has a predefined structure, making it more feasible to automatically generate meaningful visualizations for HLS-based applications.
Visualizations for HLS-based applications can show performance data in the context of the application’s architecture, as specified by the programmer. A potential (mockup) visualization is shown in Figure 3-1. In this example, profile data representing the time the application spent in various states is presented using pie charts. The communication architecture and its associated performance is also shown, connecting the processes together with streaming or DMA communication channels (Profile View and Default Profile Key in Figure 3-1). This type of representation allows all of the application profile data to be displayed in a single visualization while capturing the application’s architecture.

Figure 3-1. Example performance visualization

Rather than presenting each state used by an HLL for DMA or streaming communication, these states can be assigned to one of three categories: transferring, idle, or blocking. For example, blocking can occur if a stream FIFO becomes full and a streaming call is made;
the function will then block, preventing further execution until an element is removed from the FIFO. However, blocking can also occur when DMA communication requires a shared resource that is currently servicing another request. By categorizing any communication channel state into one of three categories, the visualization provides better scalability (states are effectively summarized) and is more readily understood (all communication channels use the same categories).

A similar categorization is used for hardware processes (where there can be hundreds of states, making categorization essential in order to obtain meaningful visualizations). States of a hardware process are assigned to one of three categories: active communication, active computation, and a miscellaneous category. Active communication can be defined as time spent for streaming or DMA calls that are non-blocking. Active computation can include the use of variables and states corresponding to an active pipeline. In the case where both computation and communication are taking place simultaneously in a process (e.g., a Carte C parallel section) time should only be added to the computation category since the overhead of communication is being hidden. The miscellaneous category acts as a catch-all for initialization and overhead such as pipeline flushing. However, the definition of overhead can vary depending on the application and programmer.

A further complication exists due to the lack of a one-to-one mapping between HLL code and hardware states. In order to help the programmer link HLL source code to the above categories, each line in the source code can be color-coded to match the corresponding category’s color (Source View in Figure 3-1). Note that some lines of code may receive no color at all if these lines of code do not require any execution time in hardware (e.g., defining a variable creates a signal in the HDL but does not consume cycles). In the case where a single line of code contains multiple states, commented lines of code can be automatically added beneath that line of code to allow the programmer to make a more fine-grained selection between states that could fall into different categories.
For example, the four states of an Impulse C pipeline (run, flush, initialize, and idle) can be added as comments below a CO PIPELINE pragma. The run state would be considered active computation whereas the initialize, flush and idle states would fall under the miscellaneous category. If multiple lines of code are grouped into a single state, then those lines of code can only be color-coded as a whole.

While presenting a breakdown of time spent in processes can provide a good indication of application performance, pinpointing the cause of bottlenecks may require more detailed trace-based data. Since local storage for trace data is likely to be limited, and since communication channels are likely to be shared with the application, it is important to define efficient triggers to minimize the trace data generated. Thresholds for trace triggers can be set after examining the Profile View for bottlenecks. As an example, the programmer may want to trigger a trace event when a stream buffer becomes full (Timeline View in Figure 3-1) or when a pipeline is stalled for 100 cycles. The programmer can iteratively refine thresholds that control trace-event triggers, if necessary, to reduce bandwidth needed for performance data.

In order to provide a scalable visualization for user-specified trace data, trace events can be displayed in a single timeline view for the entire application. This one-dimensional view will allow the programmer to quickly scan a timeline and find trace events that indicate a potential bottleneck. In contrast, traditional performance analysis visualizations such as Jumpshot [32] would dedicate a row to each parallel node (in this case, each process shown in the Profile View of Figure 3-1 since they are all operating in parallel). This two-dimensional view forces the programmer to scan a potentially large number of rows on one axis over a large period of time on the second axis in order to find a bottleneck.

3.2 High-Level Synthesis Performance Analysis Framework

A performance analysis tool for Impulse C was developed in order to illustrate techniques that address many of the major challenges described in Section 3.1. Impulse C
was selected due to its support for a variety of platforms. In order to improve the usability of the tool first described in Curreri et al. [59], two main issues needed to be addressed: automation of instrumentation and integration with an existing software performance analysis tool. In Section 3.2.1, the methods needed to automate instrumentation are discussed. Section 3.2.2 then covers the steps taken to add performance analysis for Impulse C to an existing software performance analysis tool in order to create a unified, dual-paradigm performance analysis tool.

### 3.2.1 Instrumentation Automation

Section 3.1.1 concluded that while instrumentation could be inserted at a number of levels ranging from HLL source code to binary, instrumenting at the HDL level provided the best tradeoff between flexibility and portability. Additionally, Section 3.1.2.1 concluded that extracting measurement data using HLL communication channels offers the greatest portability and is often better suited to an HPC environment than JTAG or other communication interfaces. Thus, instrumentation must be performed in two stages. Section 3.2.1.1 describes automated HLL instrumentation that inserts communication channels for measured performance data. Section 3.2.1.2 then presents an automated tool for HDL instrumentation in order to record and store application behavior at runtime.

#### 3.2.1.1 Automated C source instrumentation

In order to communicate measured performance data from hardware back to software, the framework first instruments Impulse C source code. Before instrumentation is added, the Impulse C application consists of software processes running on the host processors, hardware processes running on the FPGAs and communication channels to connect them (shown with white boxes and arrows in Figure 3-2 on the left-hand side). Automatic instrumentation modifies this structure by inserting separate definitions for a hardware process, a software process, and communication channels (shown with dark boxes and arrows in Figure 3-2 on the right-hand side). The software process can be declared as an “extern” function to be added later. Since the hardware process will be overwritten during
HDL instrumentation, a simple loopback process suffices, depicted by the cross-hatched arrow in Figure 3-2.

![Figure 3-2. Hardware Measurement Module (HMM) addition and application instrumentation](image)

Figure 3-2. Hardware Measurement Module (HMM) addition and application instrumentation

Figure 3-3 provides a flowchart illustrating the typical design flow (lightly shaded steps) as well as the changes made to that flow (darkly shaded steps) when instrumenting code for performance analysis. Instrumentation of Impulse C source code is handled automatically by the HLL Instrumenter, as shown in step two of Figure 3-3; for Impulse C, the HLL Instrumenter consists of a Perl script driven by a Java GUI frontend. The HLL Instrumenter modifies only the Impulse C hardware file; the software file remains unchanged. Since Impulse C hardware files must include a configure function to setup Impulse C processes and communication channels, the HLL Instrumenter searches for the definition of the configure function, adding the loopback hardware process code and “extern” software process declaration at the beginning of this function. The configure function will also be modified so that it declares the new software and hardware process and provides the necessary communication channels between them.

### 3.2.1.2 Automated hardware description language instrumentation

Once the application is mapped to HDL code (step 3 in Figure 3-3), the HDL Instrumenter is employed, providing the application developer the choice between default
and custom instrumentation (step 4 in Figure 3-3). Since signals that correspond to state machines in Impulse C are prime candidates for instrumentation, the default option simply monitors all state machines. Impulse C relies upon state machines in the generated HDL code to preserve the structure of the original C code. The state machine structure is primarily determined by statements that represent a branch in execution, such as if, while, for, etc. Impulse C handles C statements within a branch by placing them either in a single state or in multiple sequential states depending upon their aggregated delay. However, a loop that is pipelined is always represented as one state within this state machine. Instrumenting state machines aids the user in better understanding where time is being spent inside their hardware processes.

Custom instrumentation allows the application developer to instrument Impulse C variables. Due to the fact that Impulse C variable names are used within the names of corresponding HDL signals, the HDL signals can often be identified easily by the programmer. Variables can be instrumented by counting each time the variable is above or below some threshold, although more advanced instrumentation, such as histograms can be constructed if desired. Currently, custom instrumentation typically involves specifying thresholds via concurrent conditional VHDL statements (i.e., VHDL “when/else” statements). These conditional statements convert the instrumented signal value from
the hardware process to a one or zero value that will then control a profile counter or trace buffer. However, this process could be simplified (e.g., histogram generation only requires a value range and bin size to be specified by the user).

Once the signals to be instrumented have been selected, they are routed into the hardware loopback process (thin black arrow in Figure 3-2). The loopback process is then replaced by the Hardware Measurement Module (HMM) shown in the lower dark box in Figure 3-2. The HMM contains customized hardware with profiling and tracing capabilities (see Figure 3-4) and was originally designed for HDL performance analysis [2]. The HMM allows HDL signals to be used in arbitrary expressions that define events such as “buffer is full” or “component is idle.” These events are used to trigger custom profile counters or trace buffers depending upon the type and level of detail of performance data required. A cycle counter is also provided for synchronization and timing information. The module control provides the interface to software for transferring data back to the host processor at runtime as well as clearing or stopping the module during execution.

Figure 3-4. Hardware Measurement Module (HMM)

Once instrumentation is complete, the HDL is ready to be converted to a bitstream (step 5 in Figure 3-3) and programmed into the FPGA. In general, the techniques used in
Section 3.2.1 and Figure 3-2 should be valid for any HLS tool that employs communication channels and generates unencrypted and non-obfuscated HDL.

3.2.2 Performance Analysis Tool Integration

In order to provide the programmer with traditional microprocessor performance data (as well as a GUI frontend for viewing performance data) with minimal design effort, the Impulse C hardware performance analysis framework was integrated into an existing performance analysis tool, Parallel Performance Wizard (PPW) [3], which is designed to provide performance analysis for several parallel programming languages and models including UPC (Unified Parallel C) and MPI (Message Passing Interface). Section 3.2.2.1 explains how PPW+RC (i.e., PPW augmented with the framework for RC performance analysis) extracts performance data from the HMM. Section 3.2.2.2 then describes how measurement data for Impulse C hardware processes are visualized by PPW+RC.

3.2.2.1 Performance data extraction

The software process that extracts measurement data from the HMM was originally inserted into the application software. For Impulse C, the measurement extraction process now resides in the PPW+RC backend. For other HLLs, nameshifting could be used to intercept function calls for measurement data extraction. Integrating the measurement extraction process with PPW+RC allows PPW+RC to automatically gather performance data from Impulse C hardware processes as well as to convert received data to match the format of and mesh well with data being collected on the microprocessor (e.g., all times measured on the FPGA must be converted from cycle counts to nanoseconds in order to allow easy comparison of the time spent in hardware and software processes). Once execution of the application is finished, performance data from both hardware and software is stored in a single file for review.

3.2.2.2 Performance data visualization

The PPW+RC frontend is a Java GUI that is used to visualize performance data recorded and stored by the PPW+RC backend. Assuming the programmer has selected
the default instrumentation mode (see Section 3.2.1.2), the performance data file will contain information on all state machines (which are employed by the hardware processes); this information can easily be viewed in an expandable table or via pie charts or other graphical views. State machines associated with pipelines will also be displayed if pipelining was used. Figure 3-5 shows the PPW+RC frontend displaying timing for software processes, hardware processes, and pipelines for a DES encryption application [60]. PPW+RC can also compare multiple executions of an application to allow careful analysis of the benefits and effects of various modifications and optimizations, or possibly just to study non-deterministic behavior between executions of the same version of the application.

Figure 3-5. Profile tree table visualization for Triple Data Encryption Standard (DES)
Currently, the HDL names for signals and states are presented in visualizations. Techniques for fully automating the reverse-mapping of HDL code states to HLL source lines of code are complex and may not work for all cases. One possible technique is to perform a graph analysis comparison on the hardware control-flow graph (e.g., state machine graph) and the control-flow graph produced by a software compiler (e.g., one obtained by compiling with gcc after removing non-ANSI-C-compliant HLL statements). This could allow loops or branches in the state machine graph to be matched with loop or branch source code. Another reverse-mapping technique involves matching HLL-specific code statements with corresponding hardware. For example, HLL pipelined loops can be matched with pipeline hardware. Additional techniques such as variable name-matching (e.g., via matching similar names in both the HLL source code and the generated HDL in Impulse C to match variables to signals) can aid in matching states to source code line numbers.

Ideally, HLS tool vendors would provide support for reverse mapping, greatly simplifying the above process. For example, the data-flow graph file generated by Carte C links hardware code blocks to source lines of code, allowing the tool to perform automatic reverse mapping. For Impulse C, reverse mapping is currently tool-assisted. Impulse Accelerated Technologies, creators of Impulse C, has expressed interest in adding comments to their HDL output to make the reverse-mapping process fully automated [61]. Reverse-mapping would allow PPW+RC to provide source line correlation for both hardware and software processes, allowing the application developer to easily locate the line(s) of code associated with measured performance data. With fully automated reverse-mapping support, the unfamiliar concept of hardware states can be abstracted away allowing the software application developer to see similar performance analysis visualizations for both software and hardware.
3.3 Molecular-Dynamics Case Study

To demonstrate the benefits of HLS performance analysis and explore its associated overhead, a molecular-dynamics (MD) kernel written in Impulse C was analyzed. MD simulates interactions between atoms and molecules over discrete time intervals. MD simulations take into account standard physics, Van Der Walls forces, and other interactions to calculate the movement of molecules over time. Alam et al. [62] provides a more in-depth overview of MD simulations. This simulation keeps track of 16,384 molecules, each of which uses 36 bytes (4 bytes to store its position, velocity, and acceleration in each of the X, Y, and Z directions). Analysis is focused on the kernel of the MD application that computes distances between atoms and molecules.

Serial MD code optimized for traditional microprocessors was obtained from Oak Ridge National Lab (ORNL). The MD code was redesigned in Impulse C using an XD1000 [8] as the target platform. The XD1000 is a reconfigurable system from XtremeData Inc. containing a dual-processor motherboard with an Altera Stratix-II EP2S180 FPGA on a module in one of the two Opteron sockets. The HyperTransport interconnect provides a sustained bandwidth of about 500 MB/s between the FPGA and host processor with Impulse C. Using this platform, a speedup of 6.2 times was obtained versus the serial baseline running on the 2.2 GHz Opteron processor in the same XD1000 server. Using the prototype performance analysis tool, the performance of the MD code was analyzed to determine if further speedup could be obtained.

There are three hardware processes defined in the MD hardware subroutine (Figure 3-6). The two processes named Collector and Distributor are used to transfer data to and from SRAM, respectively, in order to provide a stream of data running through the third process, Accelerator. Accelerator calculates the position values of molecules and is pipelined using Impulse C pragmas. The process is then replicated 16 times, so that FPGA resources are nearly exhausted, so as to increase performance.
The MD kernel was instrumented and analyzed, with a focus on understanding the behavior of the state machine inside of each Accelerator process (Figure 3-7). The number of cycles spent in each state was recorded by the HMM and sent back to the host processor post-mortem. Upon examination, three groups of states in the main loop of the Accelerator process were of particular interest. The first group keeps track of the total number of cycles used by the input stream (arrows pointing to Accelerator in Figure 3-6) of the Accelerator process. The second group of states keeps track of the total number of cycles used by the pipeline inside of the Accelerator process. Finally, the third group of states keeps track of the total number of cycles used by the output stream (arrows pointing to the Collector in Figure 3-6) in the Accelerator process. Tracing was used to find the start and stop times of the FPGA and all Accelerator processes. The cycle counts from these three groups were then converted into a percentage of the Accelerator runtime (Figure 3-7) by dividing by the total number of cycles used by the MD hardware subroutine (i.e. FPGA runtime). Since the state groups vary by less than one-third of a percent when compared across all 16 Accelerators, data was only presented data from one of the Accelerator processes.

The performance analysis tool successfully identified a bottleneck in the MD hardware subroutine. In the Accelerator processes, almost half of the execution time was used by the output stream to send data to the Collector process (state b6s2 in Figures 3-7).
Figure 3-7. Accelerator process source with profiling percentages

and 3-8). An optimal communication network would allow the pipeline performing MD operations to execute for nearly 100% of the FPGA runtime minimizing the number of cycles spent blocking for a transfer to complete. This trait is an indicator that the stream buffers which hold 32-bit integers are becoming full and causing the pipeline to stall. Increasing the buffer size of the streams by 32 times only required a change of one constant in the program. This increase changes the stream buffer size to 4096 bytes for all 16 input and output streams of the Accelerator processes. Since the Impulse C compiler can only increase the stream buffer size by a power of 2, a buffer size of 4096 bytes is the maximum size that will pass place and route. Figure 3-9 shows the tradeoff between application runtime and various stream buffer sizes. The larger stream buffers reduced the number of idle cycles generated by the output stream (top bar in Figure 3-9) while the pipeline’s runtime (bottom bar in Figure 3-9) remained the same thus reducing the MD kernel’s runtime. This simple change increased the speedup of the application from 6.2 to 7.8 versus the serial baseline running on the 2.2 GHz Opteron processor.
Although per-loop analysis is currently not supported by automatic instrumentation and visualization, more detail analysis of the blocking stream calls can be performed by examining the stream transfer time for each loop iteration. Since these times are so small, they will be presented in cycles. The outer loop of the Accelerator process (Figure 3-7) is performed once per molecule (16384 times). Only one of the three output transfer states in the loop generates idle cycles (Figure 3-9); thus only that transfer state needs to be monitored. After each loop iteration, the number of cycles required by the output stream to transfer data is counted. The cycle count range is segmented into sub-ranges or bins, each 256 cycles wide. A counter is used for each range to keep track of the number of times the transfer count falls in that range. Figure 3-10 shows the stream transfer cycle count segmented into bins. Per-loop analysis of the output stream provides additional insight into the bottleneck and the effect of the buffer size on the loop iteration cycle count. As the buffer size increases, longer cycle counts become less frequent and cluster into different regions. The region corresponding to a one-cycle stream transfer represents the case where no idle cycles are generated. Even with a stream buffer size of 4096 bytes, less than 30% of the stream transfers are ideal.
The overhead caused by instrumentation and measurement of the Accelerator process with a stream buffer size of 4096 bytes on the XD1000 is shown in Table 1. The instrumented version in Table 3-1 includes all additional hardware for performance analysis (i.e., the HMM and additions to the Impulse C communication wrapper). Instrumentation and measurement hardware increased total FPGA logic utilization by 3.90%. Profile counters and timers used an additional 3.70% of the FPGA’s logic registers, whereas tracing buffers required 1.27% additional block memory implementation bits. An additional 2.73% of combinational Adaptive Look-Up Tables (ALUTs) were also needed. For routing, instrumentation increased block interconnect usage by 2.56%. Finally, the FPGA experienced a slight frequency reduction of 2.64% due to instrumentation. Overall, the overhead for performance analysis was found to be quite modest.

### 3.4 Conclusions

High-level languages have the potential to make reconfigurable computing more productive and easier to use for application developers. While this higher-abstraction
Figure 3-10. Output stream overhead for the Accelerator process.

Table 3-1. Performance analysis overhead

<table>
<thead>
<tr>
<th>EP2S180</th>
<th>Original</th>
<th>Instrumented</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic used</td>
<td>126252</td>
<td>131851</td>
<td>+5599</td>
</tr>
<tr>
<td>(143520)</td>
<td>(87.97%)</td>
<td>(91.87%)</td>
<td>(+3.90%)</td>
</tr>
<tr>
<td>Comb. ALUT</td>
<td>100344</td>
<td>104262</td>
<td>+3918</td>
</tr>
<tr>
<td>(143520)</td>
<td>(69.92%)</td>
<td>(72.65%)</td>
<td>(+2.73%)</td>
</tr>
<tr>
<td>Registers</td>
<td>104882</td>
<td>110188</td>
<td>+5306</td>
</tr>
<tr>
<td>(143520)</td>
<td>(73.08%)</td>
<td>(76.78%)</td>
<td>(+3.70%)</td>
</tr>
<tr>
<td>Block memory</td>
<td>3437568</td>
<td>3557376</td>
<td>+119808</td>
</tr>
<tr>
<td>(9383040 bits)</td>
<td>(36.64%)</td>
<td>(37.91%)</td>
<td>(+1.27%)</td>
</tr>
<tr>
<td>Block Interconnect</td>
<td>288877</td>
<td>300987</td>
<td>+12110</td>
</tr>
<tr>
<td>(536440)</td>
<td>(53.85%)</td>
<td>(56.11%)</td>
<td>(+2.56%)</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>80.57</td>
<td>78.44</td>
<td>-2.13</td>
</tr>
</tbody>
</table>

level allows the high-level synthesis tools to implement many of the design details, this higher-abstraction can also make it easier for the developer to introduce bottlenecks into their application. Performance analysis tools allow the application developer
to understand where time is spent in their application so that the best strategy for application optimization can be taken.

Many challenges for performance analysis of HLS-based FPGA applications have been identified in this chapter. A number of instrumentation levels and associated challenges were discussed; in the end, instrumentation at the HDL level was chosen for its flexibility and portability between high-level synthesis tools and platforms. In addition, many different HLL structures have the potential to be instrumented once mapped to an HDL. Instrumenting control hardware employed to maintain program order, pipelines, and communication channels is discussed. These structures are amenable to automated instrumentation and can provide performance data relevant to a wide range of applications. Communication of measured data via JTAG and HLL channels at runtime was also discussed. HLL communication channels were selected due to their portability between platforms and minimal external hardware requirements. The use of measured performance data is explained for automatic bottleneck detection at the HLL source-code level to increase developer productivity. An HLS-specific visualization displaying performance information in the context of the application’s architecture was presented, providing the programmer with an overview of application performance. The visualization also displayed bottleneck-specific tracing.

An automated framework for performance analysis of Impulse C was also presented that implements many of the techniques needed to address the challenges above. The framework incorporates automatic instrumentation of Impulse C hardware processes. State machines in each process can be instrumented to provide an execution time breakdown. Timing data gathered from these state machines are collected and visualized using a performance analysis tool, Parallel Performance Wizard. Since PPW was originally designed for performance analysis of parallel computing languages (e.g., MPI and UPC), the extension to PPW allows it to provide performance analysis of both hardware and software simultaneously.
A case study was presented to demonstrate the utility of profiling and tracing application behavior in hardware, allowing the developer to gain an understanding of where time was spent on the reconfigurable processor. Low overhead was observed (in terms of FPGA resources) when adding instrumentation and measurement hardware, demonstrating the ability to analyze applications that use a large portion of the FPGA. In addition, a slight reduction in frequency (less than 3%) resulted from instrumentation. Since data was gathered after execution completed, there was no communication overhead.

Although the mapping between HDL and HLL code is not currently presented to the programmer, future plans include linking all HDL-based data back to the HLL source, permitting the programmer to remain fully unaware of the HDL generated by a high-level synthesis tool. Additional future work includes developing more advanced visualizations for HLS-based applications and expanding the tool to support performance analysis for Carte C.
CHAPTER 4
COMMUNICATION BOTTLENECK DETECTION AND VISUALIZATION

High-level synthesis (HLS) tools such as Impulse C [55] and Carte [56] increase developer productivity by allowing developers to program Field-Programmable Gate Arrays (FPGAs) at a higher abstraction level. This higher abstraction level is achieved by using the C programming language rather than programming at the register-transfer level (RTL) using hardware description languages (HDLs). However, programming at a higher abstraction level can also lead to a loss in performance that can be difficult for a developer to optimize due to a lack of visibility into the synthesized circuit structure and runtime behavior.

Performance analysis tools are commonly used to identify performance bottlenecks in software development, but there is a lack for such tools designed specifically for HLS. Existing simulators and debugging tools provide some performance-analysis capabilities by enabling developers to monitor signals and state machines. However, these approaches are lacking in one or more of several areas: speed or accuracy, source-code correlation, high-level metrics and visualizations. HLS simulators run C source on the microprocessor, which is fast and provides source-code correlation, but is inaccurate because the timing and synchronization of hardware is not taken into account. Thus, HLS simulators are ill-suited for performance analysis. Cycle-accurate HDL simulators can provide accurate monitoring of HDL structures generated by HLS tools, but simulating large designs can be time consuming and current HDL simulators do not provide correlation back to HLS source code. Logic analyzers such as ChipScope [47] and Signal Tap [48] provide fast access to HDL signal values but do not provide correlation to C source code.

One of the key capabilities missing from existing performance-analysis approaches is visualization of high-level metrics such as bandwidth and visualizations of the application’s communication architecture. Such visualizations are common in the high-performance computing (HPC) community for parallel programming languages and libraries such as
MPI and UPC. For example, Figure 4-1 shows the amount of bytes transferred between initiating threads and data affinity threads for an application executing on 32 nodes. This visualization was generated by the Parallel Performance Wizard (PPW), a performance analysis tool which was able to get over a 14% performance improvement after optimizing the FT benchmark (fast Fourier transform) [3].

Figure 4-1. Parallel Performance Wizard (PPW) data transfer visualization

This chapter presents a communication-bottleneck detection and visualization tool for HLS. Our tool provides the developer with a high-level visualization of the bandwidth of all communication between processes of an application for both the CPU and FPGA and graphically identifies bottlenecks via color coding. We evaluated our techniques using Impulse C, but the techniques also apply to other HLS tools, such as Carte. A prototype tool was developed for Impulse C due to its support for multiple platforms. To measure bandwidth, the tool automatically adds hardware counters and software timers to the application code and then uses the measurements to generate a
communication visualization of the application. Optimizing the bottlenecks shown by the communication visualization required only several minutes of developer effort and enabled a 2.18x speedup of a Triple DES application on an XD1000 platform by XtremeData. Bottlenecks were also detected on a Molecular Dynamics application on an XD1000 platform, resulting in a speedup of 1.25x, in addition to a Backprojection application on the Novo-G supercomputing platform (24 nodes each with two GiDEL PROCStar III boards for a total of 192 FPGAs). The measured overhead of our tool was less than 2 percent resource overhead and 3 percent frequency overhead.

The remainder of the chapter is presented as follows. Section 2.6 provides related research. Section 4.1 presents the challenges for the HLS performance-analysis tool for communication-bottleneck detection. Section 4.2 presents the framework developed for Impulse C. Section 4.3 provides results from the application case studies. Section 4.4 discusses conclusions from the presented work.

4.1 Communication Performance Analysis

The focus of this work is on techniques to automatically visualize the efficiency of inter-process communication for both microprocessors and FPGAs in applications using high-level synthesis. To create the visualization, the performance analysis tool instruments application code to measure bandwidth of all communication channels during execution. A developer can then analyze the visualization to identify bottlenecks and determine appropriate optimizations to increase performance.

The remainder of this section presents techniques for visualizing and analyzing bandwidth. Section 4.1.1 presents instrumentation and measurement challenges. Section 4.1.2 describes the visualization techniques. Section 4.1.3 discusses analysis techniques for identifying bottlenecks in the visualization.

4.1.1 Instrumentation and Measurement

Communication channels between processes in high-level synthesis tools normally fall into two categories: streaming and DMA transfers. Streaming transfers use buffers to
temporarily store data between two communicating processes, where one process writes to the buffer and the other process reads from the buffer. Direct memory access (DMA) transfers move data to/from dedicated memory such as SRAM.

The performance analysis tool instruments communication channels by adding monitoring circuits connected to the synthesized communication calls in the source code of each process on the FPGA. During execution, the monitoring circuits store measured bandwidths locally in registers, which are extracted by the microprocessor after the application has finished executing to avoid communication overhead. The tool then combines the measured bandwidths for both the FPGA and CPU to form a visualization of communication channel efficiency.

Bandwidth (or transfer rate) can be measured using Equation 4–1 where $B$ is bandwidth, $D$ is data transferred and $T$ is transfer time. For all measurements in this chapter, time is the amount of time taken by a communication API call to perform a transfer including idle time as a result of blocking. For example, a stream write call will block if the streaming buffer being written to is full. This measurement of time does not penalize applications that perform mostly computation with little communication since only the communication performed by the application is being timed. In software, time is measured by adding a wrapper around each communication call type (e.g., stream read) in the source code. The wrapper function call measures the time and records the transfer size passed as a parameter for each communication call.

$$B = \frac{D}{T}$$

(4–1)

For communication API calls performed in hardware, bandwidth measurement can become more complex. The invocation of a communication API call is determined by which state is active in a state machine (e.g., in Impulse C) or handshaking signals (e.g., in Carte). Counters are used to monitor how long signals are active that are associated with a particular communication API call. Since the counters measure time in cycles
instead of seconds, the frequency of the FPGA must be determined in order to convert cycles to seconds as shown in Equation 4-2 where $C$ is cycles and $F$ is Frequency.

$$T = \frac{C}{F}$$  \hspace{1cm} (4-2)

Some communication calls have a static transfer size while other can change dynamically. We have observed that streaming transfers typically have a set width fixed during hardware generation that cannot be changed at runtime. DMA transfer sizes can be static or dynamic during program execution depending on how the application is designed. For a static transfer size, only the invocation of a communication API call must be monitored. The fixed size of each transfer can be parsed from the source code. The data transferred, $D$, can be determined by Equation 4-3, where $S$ is the fixed size of each transfer and $I$ is number of invocations. Communication invocations are detected by monitoring signals or states corresponding to communication API calls. For each invocation, a counter is incremented for only the first cycle that the signal or state for the communication API call is active. Additional cycles may be needed to finish the data transfer of the communication call. For dynamic transfer sizes, a counter is used to sum the total bytes transferred each time the API call is invoked. Equation 4-4 is used to compute the bandwidth of a communication channel that uses API calls with static and dynamic transfer sizes. Equation 4-4 is derived by substituting Equations 4-2 and 4-3 with Equation 4-1 while adding together the static and dynamic versions of $D$ and $C$. In Equation 4-4, $n$ and $m$ represent the number of API calls with static and dynamic transfer sizes, respectively, for a given communication channel.

$$D = SI$$  \hspace{1cm} (4-3)
\[
B = \frac{F\left(\sum_{i=1}^{n}(S_i I_i) + \sum_{j=1}^{m} D_j\right)}{\sum_{i=1}^{n} C_i + \sum_{j=1}^{m} C_j} \quad (4-4)
\]

It is also important to supply the visualization with the maximum bandwidth for a particular communication type. The maximum bandwidths can be measured using our tool by running a benchmark application. A typical bandwidth benchmark would need to perform four types of transfers between processes for both streaming and DMA communication: CPU to CPU, from CPU to FPGA, from FPGA to CPU and FPGA to FPGA. Alternately, vendor provided maximum bandwidths can be used. The maximum bandwidths can then be given to the visualization generator in comma separate value (CSV) format to enable color coding.

4.1.2 Visualizations

Software parallel performance analysis tools generate their visualizations in a format that assumes a homogeneous network between nodes with the capability of all to all communication as shown in Figure 4-1. HLS tools, in contrast, can generate a heterogeneous partially connected network with multiple modes of communication between processes. Furthermore, bandwidths between processes can change depending on the end points (e.g. FPGA, CPU, or memory) and type of communication. Therefore, a visualization tailored to the application’s communication architecture between processes is needed to represent the bandwidth between each node.

The visualization tool constructs a directed graph based on the source code of the application. The nodes of the graph correspond to application processes and data buffers while the edges of the graph correspond to direction that data is transferred between the nodes. In each visualization, the visualization tool draws boxes corresponding to one or more CPUs, FPGAs and external memories. The tool represents processes as ovals that are placed into their respective processor. The communication API
calls inside each process are used to create the edges that connect processes. The tool also visualizes memories in the directed graph. Streaming buffers are shown as diamonds. Figure 4-2 shows an example of streaming communication (Figure 4-2A) with a corresponding visualization of a process writing to a stream buffer (Figure 4-2B). SRAM used for DMA is shown as a separate box from the CPU or FPGA. Each DMA buffer is displayed as a separate box inside of the corresponding SRAM box. Figure 4-3 shows a DMA-communication call inside a function (Figure 4-3A), and the corresponding visualization of a process writing to a SRAM buffer (Figure 4-3B).

```
void Top(co_stream pl) {
    co_stream_write(pl, ...);
}
```

A Source code

```
void Top(co_memory datamemx) {
    co_memory_write(datamemx, ...);
}
```

A Source code

Figure 4-2. Streaming-communication call visualization

Figure 4-3. Direct memory access (DMA) communication call visualization

The tool annotates bandwidths measured for each communication API call next to each edge connecting a process and buffer. To make it easier to find bottlenecks, the edges of the graph corresponding to data transfers are color coded depending on the ratio of
measured bandwidth to the maximum bandwidth of the data transfer where shades of red are used for below 50% bandwidth utilization and shades of green are used for above 50% bandwidth utilization. Percentages of maximum bandwidth are also provided next to the measured bandwidth for each edge of the graph. As stated before, the applications that perform mostly computation with little communication will not be penalized in the visualization, since bandwidth is only measured while communication is being performed by the application.

### 4.1.3 Analysis

This section describes analysis techniques to detect communication bottlenecks. All bottlenecks (i.e., communication calls with low bandwidth) are shown in a shade of red in the visualization. By analyzing the data flow and changes in bandwidth in the visualization of the application, developers can determine potential remedies for each bottleneck.

The following analysis techniques can be used manually or automated to suggest potential optimizations to increase bandwidth utilization and speedup an application. For streaming transfers, it is important to note the ratio of input and output bandwidths to a streaming buffer. One potential bottleneck may occur when a streaming buffer is full, in which case there will be a lower input bandwidth than output bandwidth, as shown in Figure 4-4A. Lower input bandwidth occurs because the writing process must block when the buffer is full. To optimize this bottleneck, the buffer size can be increased to increase bandwidth. Another bottleneck is caused by empty stream buffers, represented with the opposite ratio, which causes the reading process to block as shown in Figure 4-4B. To optimize this bottleneck, data rates upstream should be increased to prevent the buffer from becoming empty by changing streaming widths, pipelining or switching to a different communication method. An additional bottleneck can be caused by low bandwidths on both sides of the streaming buffer, which can result from streaming bursts of data into streaming buffers that become full and empty at different times during the execution of
A Buffer full  B Buffer empty

Figure 4-4. Streaming buffer analysis

the application. To reduce this bottleneck, a combination of both optimization methods can be used.

SRAM buffers, unlike streaming buffers, must share ports between multiple processes that may try to access the buffer simultaneously. When multiple processes make simultaneous transfers, processes making DMA calls must block, resulting in a potential bottleneck. To reduce this bottleneck, synchronization can be added between processes to more effectively share bandwidth. Using small DMA transfer sizes can also cause bottlenecks, which can be optimized by sending larger chunks of data to increase bandwidth.

4.2 Experimental Framework

This section presents the framework designed to visualize communication bottlenecks in Impulse C. Section 4.2.1 gives details on the extra steps added to the Impulse C design process for instrumentation and source-code correlation. Section 4.2.2 describes which software package was used to generate the visualization.

4.2.1 Instrumentation

Since Impulse C is not open-source, instrumentation is added to the source code and hardware files. Perl scripts are used to parse and modify these files. A Java GUI front end is used to select files and instrumentation features.
The following steps are used to instrument the application and generate performance analysis data files as shown in Figure 4-5. In step 1, the developer selects the source-code files to be instrumented. In step 2, the tool adds an extra communication channel to the application that is used to transfer bandwidth measurement data after the application is finished executing. After the instrumented source code is used to generate hardware HDL files in step 3, the tool allows HDL files to be selected in step 4 to add the instrumentation described in Section 4.1.1. By default all communication states in all processes are instrumented, however, the GUI for the hardware instrumenter allows selection of which processes or individual communication states should be instrumented to reduce resource usage of the FPGA. The Impulse C XHW file is used for source-code correlation. Each communication state is annotated with its communication type and direction (e.g., stream write) and corresponding source line of code. Once the hardware is instrumented, the tool adds a custom software process to the software source code to gather the bandwidth measurement data and write the measurements to a file. Currently, only static transfer sizes are instrumented by the tool as described in Section 4.1.1. Dynamic transfer size instrumentation was not required for any of the case studies in Section 4.3.

Figure 4-5. Toolflow for Impulse C visualization
4.2.2 Visualization

After the bitfile is generated in Step 5 and the application is executed on the target platform in Step 6, the bandwidth measurement data file can be used to generate the bandwidth visualization in Step 7. The tool uses the source-code files to reconstruct the communication architecture as described in Section 4.1.2. The tool uses Graphviz [63] to generate a multi-level visualization using SVG output and linking between multiple SVG files. The generated HDL files and XHW file are used by the tool to visualize the state machine of each hardware process along with source-code correlation for each state. Currently, the analysis must be performed manually but could be automated in the future. For example, input-output bandwidth ratios of streaming buffers could be used to determine if the buffers are becoming full or empty. Optimization suggestions could be automatically provided as described in Section 4.1.3.

4.3 Experimental Results

This section presents case studies used to evaluate the communication visualization for Impulse C applications. Section 4.3.1 gives details on how the visualization was used to speedup a DES application. Section 4.3.2 demonstrates multiple communication bottlenecks in a Molecular Dynamics application, in addition to corresponding optimizations to reduce the bottlenecks. Section 4.3.3 shows a communication visualization for multiple FPGAs on a Backprojection application.

The framework currently uses Impulse C and Quartus. The target platforms are the XtremeData XD1000 [8] containing a dual-processor motherboard with an Altera Stratix-II EP2S180 FPGA in one of the Opteron sockets and the Novo-G supercomputer [64] at University of Florida containing 48 GiDEL PROCStar III [10] cards each with four Stratix-III EP3SE260. Impulse C 3.3 is used for the XD1000 while Impulse C 3.6 with an in-house platform support package is used for Novo-G. Quartus 9 was used for Triple DES and Backprojection. Molecular Dynamics would not fit using Quartus 9 and required Quartus 8.1 to fit due to the high resource utilization of the Molecular Dynamics cores.
4.3.1 Triple Data Encryption Standard (DES)

Triple DES [60] is a block cipher used for encryption. The application consists of a modified version of the Triple DES code provided by Impulse C, which sends text files to the FPGA to be encoded and then decoded again before being sent back and compared. We evaluated this application on the XD1000 platform. Figure 4-6 shows the resulting visualization of the DES application, which solely uses streaming communication. By analyzing the visualization, we identified that transfers between the CPU and FPGA were a bottleneck because of their low bandwidth relative to the FPGA internal bandwidth. The stream buffer `blocks_decrypted_ic` has a higher input bandwidth than output bandwidth indicating that streaming communication is being blocked because the buffer is empty. Since the XD1000 platform supports DMA transfers and the application is not utilizing DMA for the FPGA, we used DMA transfers to increase the transfer rates between the CPU and FPGA as shown in Figure 4-7. Large black arrows have been added to Figures 4-6 and 4-7 to point out the bandwidth increase. The change from streaming to DMA communication was a simple modification and took about half an hour to complete. This modification enabled a 2.18x speedup and greatly increased bandwidth usage. The resource overhead of the instrumentation and additional Impulse C communication channels used for measurement and data extraction was quite modest at less than 2% resource overhead and 4% frequency overhead as shown in Table 5-1.

4.3.2 Molecular Dynamics

Molecular Dynamics (MD) simulates interactions between atoms and molecules over discrete time intervals. Alam et al. [62] provides a more in-depth overview of MD simulations. For our experiments, the simulation keeps track of 16,384 molecules, each of which uses 36 bytes (4 bytes to store its position, velocity, and acceleration in each of the X, Y, and Z directions). The kernel of the MD application computes distances between atoms and molecules. Serial C MD code was obtained from Oak Ridge National
Figure 4-6. Communication visualization of streaming DES Lab (ORNL) and optimized to run on the FPGA using Impulse C. We evaluated this application on the XD1000 platform.

This case study provides a comparison between the modified PPW tool presented in Section 3.3 and the bandwidth visualization tool presented in this chapter. By analyzing Figure 4-8, we identified a bottleneck resulting from all of the streaming buffers with the letter p becoming full since the input bandwidth is lower than output bandwidth. The streaming buffers with the letter a have low bandwidth both on the inputs and outputs. The low bandwidths measured are caused by the streaming buffer becoming full and the bottom process blocking during stream reads. The bottom process requires data from all a streams to be ready simultaneously for its stream read state. If one a buffer becomes empty, then the other can become full while the bottom process waits for data. To reduce the bottleneck, we increased the stream buffer size. The stream buffer
size can be increased by changing a constant in the application. Finding the maximum buffer size that would fit on the FPGA only required several minutes of developer effort (not including the time for Quartus to implement the designs). Maximizing the stream buffer size achieved a speedup of 1.25x compared to the original FPGA implementation. The speedup compared to the serial baseline running on the 2.2 GHz Opteron processor changed from 6.2x to 7.8x after optimizing the detected bottlenecks.

4.3.3 Backprojection

Backprojection is a DSP algorithm for tomographic reconstruction of data via image transformation. We evaluated this application on all four FPGAs of the ProcStar-III board in the Novo-G supercomputer. Since the in-house Impulse C platform support package for Novo-G currently requires the use of GiDEL software API calls, some manual instrumentation and file generation was required. Only one FPGA was instrumented although its data is representative of all other FPGAs. Figure 4-9 shows the resulting visualization. One obvious bottleneck is that PCI data transfers are 8-25% of peak speeds.
Table 4-1. Triple Data Encryption Standard (DES) instrumentation overhead

<table>
<thead>
<tr>
<th>EP2S180</th>
<th>Original</th>
<th>Inst.</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Used</td>
<td>21192</td>
<td>24032</td>
<td>+2840</td>
</tr>
<tr>
<td>(out of 143520)</td>
<td>(14.77%)</td>
<td>(16.75%)</td>
<td>(+1.98%)</td>
</tr>
<tr>
<td>Comb. ALUT</td>
<td>12972</td>
<td>14935</td>
<td>+1963</td>
</tr>
<tr>
<td>(out of 143520)</td>
<td>(9.04%)</td>
<td>(10.41%)</td>
<td>(+1.37%)</td>
</tr>
<tr>
<td>Registers</td>
<td>13896</td>
<td>16242</td>
<td>+2346</td>
</tr>
<tr>
<td>(out of 143520)</td>
<td>(9.68%)</td>
<td>(11.32%)</td>
<td>(+1.64%)</td>
</tr>
<tr>
<td>Block RAM</td>
<td>143520</td>
<td>158400</td>
<td>+9216</td>
</tr>
<tr>
<td>(9383040 bits)</td>
<td>(1.59%)</td>
<td>(1.69%)</td>
<td>(+0.10%)</td>
</tr>
<tr>
<td>Block Interconnect</td>
<td>39067</td>
<td>42969</td>
<td>+3162</td>
</tr>
<tr>
<td>(out of 536440)</td>
<td>(7.28%)</td>
<td>(7.87%)</td>
<td>(+0.59%)</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>77.02</td>
<td>74.47</td>
<td>-2.55</td>
</tr>
</tbody>
</table>

Figure 4-8. Molecular-dynamics bandwidth visualization (half of FPGA cropped to enlarge image)

The streaming buffers *sino_in* show signs of becoming full since their input bandwidth is lower than their output bandwidth. This bottleneck could potentially be reduced by increasing the streaming buffer size. Unfortunately, we were not able to evaluate this optimization due to limitations of the current platform support package for Novo-G,
which only supports fixed sizes of streaming buffers. Alternatively, bandwidths could be increased by using DMA transfer instead of streaming transfers, but DMA transfers are not yet supported by the platform support package. Although the potential speedup is not known, the visualization allows a developer to easily identify the communication bottlenecks, which could be reduced in just several minutes for a system with a complete platform support package.

Figure 4-9. Communication visualization of Backprojection

4.4 Conclusions

In this chapter, we introduced a communication visualization tool for high-level synthesis that allows a developer to quickly locate communication bottlenecks. The application’s communication API calls are visualized as a directed graph of communication links between the CPU, FPGA and communication buffers. Bandwidths are color coded to allow the developer to quickly locate inefficient data transfers. By analyzing the graph for bandwidth distribution and ratios, optimizations can be made by increasing streaming buffer sizes or switching to different communication types. Case studies were provided to show how the bandwidth visualization can be used to detect and eliminate bottlenecks related to communication with little effort, which resulted in FPGA application speedups ranging from 1.25x to 2.18x. In addition, the tool provides source-code correlation, which hides the HLS-generated HDL from the application developer. Future work includes
automating analysis and providing suggestions for optimization. The visualization can also be expanded by including performance analysis of computation for each process.
CHAPTER 5
ASSERTION-BASED VERIFICATION FOR HIGH-LEVEL SYNTHESIS

Field-programmable gate arrays (FPGAs) show significant power and performance advantages as compared to microprocessors [1], but have not gained widespread acceptance largely due to prohibitive application design complexity. High-level synthesis (HLS) significantly reduces application design complexity by enabling applications written in a high-level language (HLL) such as C to be executed on FPGAs. However, limited HLS support for verification, debugging, and timing analysis has contributed to limited usage of such tools.

For verification, designers using HLS can use assertion-based verification (ABV), a widely used technique in electronic design automation (EDA) tools [65], to verify runtime behavior by executing an application that contains assertions against a testbench. However, assertion-based verification of programs written in C using HLS tools, such as Impulse C [55] and Carte [56], is often limited to software simulation of the FPGA’s portion of the code, which can be problematic due to common inconsistencies between simulated behavior and actual circuit behavior. Such inconsistencies most commonly result from timing differences between the software thread-based simulation of the circuit and the actual FPGA execution [66]. In some cases, these inconsistencies may cause an application that behaves normally in software simulation to never complete (i.e., hang) when executing on the FPGA. Debugging an HLS-generated circuit to identify the cause of such hangs is a significant challenge that currently requires excessive designer effort.

Timing analysis, a procedure which determines if performance constraints are met, is an additional limitation of many HLS tools. Although timing analysis is widely used in physical design tools, in many cases HLS tools do not consider timing constraints. Even worse, designers are unaware of the performance of different regions of an HLS-generated circuit, which makes optimization more difficult. Although timing measurements can be
taken during high-level simulation, such measurements are based on software simulation and do not reflect actual circuit performance [51].

One potential solution to these verification, debugging, and timing-analysis problems is for designers using HLS to use post-synthesis register-transfer-level (RTL) simulation. However, such an approach requires a designer to manually add assertions to HLS-generated hardware-description-language (HDL) code, which is a cumbersome process (as compared to adding assertions at the source level) and there are numerous situations where such simulations may be infeasible or undesirable. For example, a designer may use HLS to create a custom core that is part of a larger multiprocessor system that may be too complex to model with cycle accuracy. Even if such modeling was realized, slow simulation speeds can make such verification prohibitive to many designers.

Ideally, designers could overcome these limitations by specifying assertions in high-level code, which the HLS tool could integrate into generated circuits to verify behavior and timing, while also assisting with debugging. To achieve this goal, we present HLS techniques to efficiently support in-circuit assertions. These techniques enable a designer to use assertions at the source level while checking the behavior and timing of the application. Furthermore, we leverage such assertions to enable a debugging technique referred to as hang detection that reports the specific high-level regions of code where a hang occurs. To realize these in-circuit-assertion-based techniques, this chapter addresses several key challenges: scalability, transparency, and portability. Scalability (large numbers of assertions) and transparency (low overhead) are interrelated challenges that are necessary to enable thorough in-circuit assertions while minimizing effects on program behavior. We address these challenges by introducing optimizations to minimize performance and area overhead, which could potentially be integrated into any HLS tool. Portability of in-circuit assertion synthesis, for verification or timing analysis, is critical because HLS tools can target numerous platforms and must therefore avoid platform-specific implementations. The presented techniques achieve portability by
communicating all assertion failures over the HLS-provided communication channels.

Using a semi-automated framework that implements the presented HLS techniques, we show that in-circuit assertions can be used to rapidly identify bugs and violations of timing constraints that do not occur during software simulation, while only introducing a small overhead (e.g., reduction in frequency on the order of less than 3% and increase in FPGA resource utilization of 0.7% or less have been observed with several application case studies on an Altera Stratix-II EP2S180 and Stratix-III EP3SE260). Various case studies with optimized assertions have shown a 3x reduction in resource usage and improved assertion performance by as much as 100% compared to unoptimized assertion synthesis. Such work has the potential to improve designer productivity and to enable the use of FPGAs by non-experts who may otherwise lack the skills required to verify and optimize HLS-generated circuits.

This chapter is presented as follows. Section 2.6 discusses related work. Assertion-synthesis techniques and optimizations are explained in Section 5.1. Section 5.2 discusses timing analysis. Hang detection is described in Section 5.3. Section 5.4 describes the experimental setup and framework used to evaluate the presented techniques. Section 5.5 presents experimental results. Section 5.6 provides conclusions.

### 5.1 Assertion Synthesis and Optimizations

American National Standards Institute C (ANSI-C) assertions, when combined with a testbench, can be used as a verification methodology to define and test the behavior of an application. Each individual assertion is used to check a specific run-time Boolean expression that should evaluate to true for a properly functioning application. If the expression evaluates to false, the assertion prints failure information to the standard error stream including the file name, line number, function name, and expression that failed; after this information is displayed, the program aborts.

The presented HLS optimizations for in-circuit assertions assume a system architecture consisting of at least one microprocessor and FPGA, and an application modeled as a task
graph. These assumptions are common to existing HLS approaches [55]. Therefore, the discussed techniques are potentially widely applicable with minor changes for different languages or tools.

In-circuit assertions are integrated into the application by generating a single assertion checker for each assertion and an assertion notification function, as shown in the top right hand side of Figure 5-1. The assertion checker implements the corresponding Boolean assertion condition by fetching all data, computing all intermediate values, and signaling the assertion notification function upon failure. The assertion notification function is responsible for printing information regarding all assertion failures and halting the application.

![Assertion framework](image)

**Figure 5-1. Assertion framework**

The assertion notification function can run simultaneously with the application as a task waiting for failure messages from the assertion checkers. The task is defined essentially as a large switch statement per communication channel that implements one case for each hardware-mapped assertion. Although a hardware/software partitioning algorithm could potentially map the assertion notification function task to either hardware or software, typically the assertion notification function will be implemented in software due to the need to communicate with standard error. Although the added HLS communication channels in the task graph could greatly increase the I/O requirements for
hardware/software communication, such a situation is avoided by time multiplexing all communication over a single physical I/O channel (e.g., PCIe bus, single pin).

Performance overhead due to this time-multiplexing should be minimal or even nonexistent (depending on the HLS tool) since ANSI-C assertions only send messages upon failure and halt the program after the first failed assertion.

One potential method to synthesize assertion checkers into circuits is described as follows. Semantically, an assert is similar to an if statement. Thus, assertions could be synthesized by converting each assertion into an if statement, where the condition for the if statement is the complemented assertion condition and the body of the if statement transfers all failure information to the assertion notification function. Although such a straightforward conversion of assert statements may be appropriate for some applications, in general this conversion will result in significant area and performance overhead. To deal with this overhead, we present three categories of optimizations that improve the scalability and transparency of in-circuit assertions, which are described in the following sections.

### 5.1.1 Assertion Parallelization

To maximize transparency of in-circuit assertions, the circuit for the assertion checker should have a minimal effect on the performance of the original application. However, by synthesizing assertions via direct conversion to if statements, the synthesis tool modifies the application’s control-flow graph and resulting state machine, which adds an arbitrarily long delay depending on the complexity of the assertion statement. For Impulse C, the delay of the assertion `assert((j <= 0 || a[0] == i) && (b[0] == 2 || i > 0))` can be shown by comparing the corresponding subset of the application’s state machine before (Figure 5-2A) and after (Figure 5-2B) the assertion is added. For this example, the assertion can add up to seven cycles of delay to the original application for each execution of the assertion. While seven cycles may be acceptable for some applications, if this assertion occurred in a performance-critical loop, the assertion could potentially reduce the loop’s
rate (i.e., the reciprocal of throughput) to 12.5% of its original single-cycle performance, which could significantly affect how application components interact with each other.

HLS tools can minimize the effect of assertions on the application’s control-flow graph by executing the assertions in parallel with the original application. To perform this optimization, HLS can convert each assertion statement into a separate task (e.g., a process in Impulse C) that enables the original application task to continue execution while the assertion is evaluated. Instead of waiting for the assertion, the application simply transfers data needed by the assertion task, and then proceeds.

For the previous assertion example, the optimization reduced the overhead from seven cycles to a single cycle as shown in Figure 5-2C. The optimization was unable to completely eliminate overhead due to resource contention for shared block RAMs. Such overhead is incurred when the assertion task and the application task simultaneously require access to a shared resource.

5.1.2 Resource Replication

As mentioned in the previous section, resource contention between assertions and the application can lead to performance overhead. Contention between assertions can happen even when assertions are executed in parallel. To minimize this overhead, HLS can perform resource replication by duplicating shared resources.

For example, arrays in C can be synthesized into block RAMs. A common source of overhead is due to the limited number of ports on block RAMs that are simultaneously used by both the application tasks and assertion tasks. When accessing different locations of the block RAM, the circuit must time-multiplex the data to appropriate tasks, which causes performance overhead. HLS can effectively increase the number of ports by replicating the shared block RAMs, such that all replicated instances are updated simultaneously by a single task. This optimization ensures that all replicated instances contain the same data, while enabling an arbitrary number of tasks to access data from the shared resource without delay.
Resource replication provides the ability to reduce performance overhead at the cost of increased area overhead. Such tradeoffs are common to HLS optimizations and are typically enabled by user-specified optimization strategies (i.e., optimize for performance as opposed to area). One potential limitation of resource replication is that for a large
number of replicated resources, the increased area overhead could eventually reduce the clock speed, which may outweigh the reduced cycle delays. However, for the case study in Section 5.5.2.3, resource replication improved performance by 33% allowing the application’s pipeline rate to remain the same.

5.1.3 Resource Sharing

Whereas the previous two optimizations dealt with performance overhead, in-circuit assertions can also have a large area overhead. Although an assertion checker circuit will generally cause some overhead due to the need to evaluate the assertion condition, HLS can minimize the overhead by sharing resources between assertions. For example, if a particular task has ten assertions with a multiplication in the condition, resource sharing could potentially share a single multiplier among all the assertions.

Although resource sharing is a common HLS optimization [67] for individual tasks, sharing resources across assertions adds several challenges due to the requirement that all statements sharing resources must be guaranteed to not require the resources at the same time. For task-graph-based applications, assertions may occur in different tasks at different times. This uncertainty prevents a HLS tool from statically detecting mutually exclusive execution of all assertions.

Due to this limitation, HLS can potentially apply existing resource-sharing techniques to assertions within non-pipelined regions of individual tasks, because those assertions are guaranteed to not start at the same time. However, due to the assertion parallelization optimization, different starting times for two assertions do not guarantee that their execution does not overlap. For example, an assertion with a complex condition may not complete execution before a later assertion requires a shared resource. To deal with this situation, HLS can implement all assertions that share resources as a pipeline that can start a new assertion every cycle. Although this pipeline will add latency to all assertions in the same task that require access to the shared resources, such latency does not affect
the application, and only delays the notification of program failure. This technique of pipeline assertion checking is evaluated in Section 5.5.2.1.

Resource sharing could potentially be extended to support an arbitrary number of simultaneous assertions in multiple tasks by synthesizing a pipelined assertion checker circuit that implements a group of simultaneous assertions. To prevent simultaneous access to shared resources, the circuit could buffer data from different assertions using FIFOs (e.g., one buffer per assertion) and then process the data from the FIFOs in a round-robin manner. This extension requires additional consideration of appropriate buffer sizes to avoid having to stall the application tasks, and an appropriate partitioning of assertions into assertion checker circuits, which we leave as future work.

In some cases, resource sharing may improve performance in addition to reducing area overhead. This benefit is made possible by enabling placement and routing to achieve a faster clock due to fewer resources. However, resource sharing will at some point experience diminishing returns, and may eventually increase clock frequency due to a large increase in multiplexers and other steering logic.

5.2 In-Circuit Timing-Analysis Assertions

For applications with real-time requirements, particularly in embedded systems, verification must guarantee that all timing constraints are met (a process referred to as timing analysis) in addition to checking the correctness of application behavior. If an HLS-generated application does not meet timing constraints during execution, then it would be helpful to know the location of the section of code that is violating constraints in order to focus optimization effort. However, determining the performance of an HLS-generated application can be difficult. HLS tools, such as Impulse C and Carte, provide some compile-time feedback about the rate and latency of a pipelined loop, but it is largely unknown how many cycles a particular line of code will require. While it is possible to determine the number of cycles a line (or lines) of code will take by examining the HDL generated by the tool, delay can be data-dependent, as shown in the possible
traversals of the state machine generated by the evaluation of the conditional statement 
\( i f((j <= 0 || a[0] == i) && (b[0] == 2 || i > 0)) \) in Figure 5-2B. However, such a process
requires significant designer effort and requires the designer to have knowledge of the
HLS-generated code. While a delay range for the computation in each line of code could
be provided by the HLS tool via static analysis, the delay of communication calls cannot
be determined by static analysis. Software simulation cannot provide accurate timing due
to timing differences between thread execution on the microprocessor and execution on the
FPGA. In this section, we describe the additional concepts and methods needed to extend
in-circuit assertions to perform timing analysis for applications built with HLS tools.

Figure 5-3 illustrates usage of timing-analysis assertions for an audio filtering
application designed with a HLS tool. In this example, the application designer has
determined that the filter takes too long to execute on the FPGA by measuring the
time to run the application on the FPGA. However, the application designer is unsure
of which part of the application in the FPGA is not meeting timing constraints. Using
timing-analysis assertions, the application designer can check the timing of different
application regions in the FPGA, as shown in the figure in addition to the case study in
Section 5.5.5. Data-dependent delays can be checked to see if they are within bounds for
each loop iteration. Although not shown in the figure, the same method can be used to
check streaming communication calls for delays caused by buffers becoming full or empty.

In order to enable ANSI-C assertions to check the timing of an application, time
must be accessible via a variable. In C, time is typically determined via a function call.
In Figure 5-3, the ANSI-C function, clock, is used to return the current time in cycles.
To measure the time of a section of code, the clock function should be called before
and after that section of code, with the difference between the two times providing the
execution time (in cycles). To perform timing analysis, an assertion can be used to check
a comparison between the expected time and the measured time. For example, in Figure
5-3, the code in the loop for each filter is expected to take less than 100 cycles.
Figure 5-3. Using timing-analysis assertions with a filter application

For timing-analysis assertions, time can potentially be represented in many different formats. However, returning time in terms of cycles will require the least amount of overhead. The ANSI-C library provides the clock timing function that returns the number of clock ticks that have elapsed since the program started. However, for C programmers who may want to express time in terms of seconds rather than cycles, the ANSI-C constant expression CLOCKS_PER_SEC can be used to convert clock ticks to time in seconds. The clock frequency of the FPGA could be determined by comparison with timestamps sent from the CPU. However, an assertion may need to be checked on the first cycle after an FPGA restart. Since determining the frequency of the FPGA automatically
could take too long, a preprocessor constant FPGA_FREQ is used to define the FPGA frequency in Hz.

The type defined for representing clock ticks in ANSI-C is clock_t that typically corresponds to a long integer. For added flexibility when used in hardware, time can be returned and stored as a 32-bit or 64-bit value. A 64-bit value is used by default. To select a 32-bit value, the preprocessor constant CLOCK_T_32 must be defined in the code. A 32-bit value can be used to reduce overhead but will overflow after 43 seconds for a clock speed of 100MHz. During software simulation, the assertions using timing information are ignored, which allows simulation to check correctness of the application while ignoring the timing of the microprocessor.

To enable synthesis of timing assertions, a counter, which is set to zero upon reset, is added in each hardware process that contains a clock statement. The value returned by the clock statement is generated by latching the counter signal for each transition of the state machine. Use of a latched counter signal ensures that the timer value is consistently taken at the beginning of each state transition for states that execute more than one cycle.

One potential problem with this approach is that HLS tools often reorder statements to maximize parallelism. Therefore, clock statements could potentially be reordered leading to incorrect timing results. However, such a problem is easily addressed by making a synthesis tool aware of clock statements. In this chapter, we alternatively evaluated the techniques using instrumentation due to the inability to modify commercial HLS tools. Although instrumentation could experience reordering problems, for the evaluated examples, reordering of clock statements did not occur.

5.3 Hang-Detection Assertions

A common problem with FPGA applications is a failure to finish execution, which is often referred to as hanging. Common causes of hanging include infinite loops, synchronization deadlock, blocking communication calls that wait indefinitely to send or receive data, etc. Determining the cause of a hanging application, referred to as hang
detection, is difficult for HLS-generated FPGA designs. While a debugger could be used to trace down the problem during software simulation, the inaccuracies of software simulation can miss hangs that occur during FPGA execution. To deal with this problem, we extend in-circuit assertions to enable hang detection for HLS-generated application.

One challenge of hang detection using assertions is that it is assumed that the assertion will eventually be checked. If the application waits indefinitely for a line of code to finish (e.g., an infinitely blocking communication call) then a different detection method is needed, since the assertion after the hung line will never be executed as shown in Figure 5-4A. Without some mechanism to alert the developer to the current state of the program, it will be difficult to pinpoint the problem. For example, in the filter application (see Figure 5-5), the source of the problem that is causing the application to hang could be in any of the software or hardware processes.

![Timing Assertion](image)

A Timing Assertion

![Assertions set to fail](image)

B Assertions set to fail

Figure 5-4. Manually using American National Standards Institute C (ANSI-C) assertions for hang detection

One potential solution is to use assertions in a counterintuitive way by adding assertions periodically throughout the code that are designed to fail (i.e., `assert(0)`). By also defining the NABORT flag, failed assertions will not cause the application to abort, which allows the developer to manually create an application heartbeat (i.e., a signal sent as a notification that the process is alive) that traces the execution of the application on the FPGA as shown in Figure 5-4B. In the filter application example, multiple assertions would need to be placed in strategic locations in each FPGA process to determine the events that take place before the application hangs. The resolution (in terms of lines of
Figure 5-5. Using hang-detection assertions with a filter application

code) would be determined by how many assertions are used. Unfortunately, if a large number of assertions are used, then large amounts of communication and FPGA resources could be used by the assertions. Although this approach works, it requires significant designer effort and has large overhead.

To reduce effort and overhead, we present a more automated method of hang detection that does not require user instrumentation and instead uses watchdog timers to monitor the time between changes of the signals that represent the state of the hardware process. The monitoring circuit has software-accessible registers that contains the current state of all hardware process and the state of any hardware process that it has detected as hung. Hang detection is triggered using a watchdog timer for a hardware process that
signals when a state takes longer than a user-defined number of cycles; the assertion pragma, `#pragma assert_FPGA_watch_dog`, sets this timeout period, which is reset anytime a state transition occurs. The watchdog timer is sized to be just large enough to hold the cycle count given in the pragma to reduce FPGA resource and frequency overhead. In software, a separate thread is spawned to monitor the hardware hang detector to check for hung states (i.e., expired watchdog timers). If a hardware process has hung then the state in the registers is matched to the corresponding line of code via a lookup table generated by parsing an intermediate translation file (both Impulse C and Carte create these files). The state of all other hardware processes are given for reference.

In software, many HLS applications will wait indefinitely at some point in its execution for the FPGA to respond with some form of communication or synchronization. For those applications, hangs caused in the FPGA hardware will also cause the software to hang on the communication or synchronization API call for the FPGA. Although traditional debugging tools can be used to detect these hangs in software, software hang detection is provided to monitor the HLS API calls for convenience. A thread is spawned for all API calls of the HLS tool. The thread will check if the API call finishes within a time period set by the assertion pragma, `#pragma assert_API_watch_dog`. If the API call takes longer than the time out period then the current line of code for the API call and all hardware processes will be printed to standard output and the program will abort.

This automated approach simplifies the addition of hang detection to an application, as shown for the filter application in Figure 5-5 and case study in Section 5.5.6, compared to manually adding `assert(0)` statements. Two assertion pragmas are added to the application before instrumentation to set the watchdog timeout periods in hardware and software. Although hangs can be caused by the interaction between two or more (hardware or software) processes, providing the state of the hung process along with the current state of all other hardware processes can greatly narrow down the source of problem.
Several improvements can be added to further enhance hang detection of HLS applications. The feedback given to the application developer can be increased by reporting more than the last state of each process in the FPGA. For example, a trace buffer could be added of a user-defined size that would capture the sequence of state that occurred before the hardware process hung. Also, infinite loops in a hardware process will only trigger software API hang detection. Since infinite loops will not stay in a single state to trigger the hang-detection method mentioned above, detection of infinite loops in hardware could also be incorporated by adding a second counter for each process that is dedicated to counting the number of cycles spent in states that are known to be inside one or more loops. The overhead of hang detection could be reduced by allowing the user to select which processes to monitor. The hang detection counters could be removed for some or all processes while still allowing the current state of the process to be periodically retrieved or retrieved by software API hang detection. This approach would give the user the option to customize hang detection to fit for designs that nearly fill the FPGA.

5.4 Assertion Framework

To evaluate the assertion-synthesis techniques, we created a prototype tool framework for Impulse C that implements the techniques via instrumentation of HLL and HDL code. It should be noted that we use instrumentation because we are unable to modify the proprietary Impulse C tool. All of the techniques are fully automatable and ideally would be directly integrated into an HLS tool.

5.4.1 Unoptimized Assertion Framework

To implement basic in-circuit assertion functionality, the framework uses HLL instrumentation to convert assert statements into HLS-compliant code in three main stages. First, the C code for the FPGA is parsed to find functions containing assertion statements, converting any assertion statements to an equivalent if statement. A false evaluation produces a message that will be retrieved from the FPGA by the CPU, uniquely identifying the assertion. Next, communication channels are generated to transfer
these messages from the FPGA to the CPU. Finally, the assertion notification function is defined as a software function executing on the CPU to receive, decode, and display failed assertions using the ANSI-C output format. An example of this automated code instrumentation is shown in Figure 5-6.

![Figure 5-6. High-Level Language (HLL) assertion instrumentation](image)

To notify the user of an assertion failure, the framework uses an error code that uniquely identifies the failed assertion based on the line number and file name of the assertion. Once the assertion notification function decodes the assertion identifier, the user is notified by printing to the standard error stream by the CPU for the current framework. The framework could be extended to work without a CPU by having the assertion identifier stored to memory, displayed on a LCD, or even flashed as a sequence on an LED by the FPGA. Alternatively, an FPGA could potentially use a soft-core processor.

Note that other changes are needed to route the stream to the CPU, such as API calls to create and maintain the stream. The stream must also be added as a parameter to the function. The output of the framework is valid Impulse C code, allowing further modifications to the source code with no other changes to the Impulse C tool flow. Once verification of the application is finished, the constant NDEBUG can be used to disable all
assertions and reduce the FPGA resource overhead for the final application. An additional nonstandard constant NABORT can be used to allow the application to continue instead of aborting due to an assertion failure.

5.4.2 Assertion Framework Optimizations

In order to evaluate the optimizations presented in Section 5.1, a hybrid mix of manual HLL and HDL instrumentation was used. To enable assertion parallelization (Section 5.1.1), the framework modifies the HLL code to move assertions into a separate Impulse C process. The framework introduces temporary variables to extract data needed by the assertion. HDL instrumentation then connects the temporary variables and trigger conditions between processes. The results of this optimization can be found in Section 5.5.2.

Resource replication was performed using manual HLL instrumentation. Resource replication is described in Section 5.1.2. An extra array was added to the source code that performed the same writes as the original array but reads were only performed by the assertion, as shown in Section 5.5.3.

The following manual hybrid instrumentation was used to evaluate resource sharing as described in Section 5.1.3. Although resource sharing could potentially be applied to any shared resource, we evaluate the optimization for shared communication channels, which are common to all Impulse C applications. HLL instrumentation creates a streaming communication channel per Impulse C process and sends the identifier of the assertion upon assertion failure. Creating a streaming communication channel per Impulse C process can become expensive in terms of resources if a large number of Impulse C processes contain assertions. To reduce the number of streams created for each process, a single bit of the stream is used per assertion to indicate if an assertion has failed. This technique allows Impulse C processes to more efficiently utilize the streaming communication channels. When streaming communication resources are shared, a separate process is created via HLL instrumentation that can handle failure signals from up to
32 assertions per process if a 32-bit communication channel is used. For example, if all 32 assertions fail simultaneously then all 32 bits of the communication channel will simultaneously be asserted. The failure signals are connected to assertions using HDL instrumentation for efficiency. The overhead reduction associated with using this technique is explored in the case study that is presented in Section 5.5.4.

5.4.3 Timing-Analysis and Hang-Detection Extensions

Semiautomatic hybrid instrumentation was used to support timing functions presented in Section 5.2. Impulse C does not support ANSI-C library calls so the clock function calls must be removed. A placeholder variable is declared and used in place of the clock statement in the source code. After hardware generation, a Perl script is used to instrument the HDL. A counter is added in each hardware process that contains a clock statement, which is set to zero upon reset. A second signal is added to the process that latches the counter signal upon transition of the state machine. The placeholder variable, synthesized into a signal with a similar name in HDL, is replaced with the latched counter signal.

Semiautomatic hybrid instrumentation was used for hang detection in Section 5.3. For software hang detection, a wrapper was added around each of the Impulse C library API calls which added the threaded hang detection. The modified software API calls required extra parameters for access to the hardware hang-detection registers. Automatic parsing of the xhw file generated by Impulse C allows states to be converted to line numbers. For hardware hang detection, a hardware process supporting register transfer to software is automatically added to the source code. After Impulse C generates the HDL, the state machine signals of all other hardware processes are automatically routed into the hang-detection process. The hang-detection circuit is then manually added by overwriting part of the register transfer process.

Although many of the steps for adding timing-analysis and hang-detection instrumentation were manual, all of the steps could be automated via Perl scripts. Ideally, modification to
the Impulse C tool would be made instead of instrumenting source and intermediate code. However, because Impulse C is proprietary, such modification was not possible for this work.

5.4.4 High-Level Synthesis Tool and Platform

The framework currently uses Impulse C. Impulse C is a high-level synthesis tool to convert a program written in a subset of ANSI-C to hardware in an FPGA. Impulse C is primarily designed for streaming applications based upon the communicating sequential process model but also supports shared memory communication. Speedups can be achieved in Impulse C applications by running multiple sequential processes in parallel, pipelining loops and adding custom HDL coded functions calls.

Quartus 9 was used for synthesis and implementation of the Impulse C-generated circuits. The target platforms are the XtremeData XD1000 [8] containing a dual-processor motherboard with an Altera Stratix-II EP2S180 FPGA in one of the Opteron sockets and the Novo-G supercomputer [64] at University of Florida containing 48 GiDEL PROCStar III [10] cards each with four Stratix-III EP3SE260. Impulse C 3.3 is used for the XD1000 while Impulse C 3.6 with an in-house platform support package is used for Novo-G. Although the XD1000 and Novo-G are high-performance computing platforms, Impulse C also supports embedded PowerPC and MicroBlaze processors [66]. Furthermore, Novo-G and the XD1000 are representative of FPGA-based embedded systems that combine CPUs with one or more FPGAs. The presented overhead results would likely be similar for other embedded platforms, assuming similar Impulse C wrapper implementations.

Although we currently evaluate HLS assertions using Impulse C, the techniques are easily extended to support other languages. For example, in Carte, Impulse C’s streaming transfers would be replaced with DMA transfers. The software-based assertion notification function (see Figure 5-1) would then need to monitor Carte’s FPGA function calls for failed assertions as opposed to monitoring Impulse C’s FPGA processes.
5.5 Experimental Results

This section presents experimental results that evaluate the utility and overhead of the presented assertion synthesis, timing analysis and hang detection. Section 5.5.1 motivates the need for in-circuit assertions by illustrating a case study where assertions pass during simulation but fail during FPGA execution. Section 5.5.2 illustrates the performance and overhead improvements of the assertion parallelization optimization. Section 5.5.3 evaluates performance benefits of resource replication. Section 5.5.4 evaluates the scalability of assertions in terms of resource and frequency overhead by applying resource sharing optimizations to the communication channels. Section 5.5.5 presents the overhead of using assertions for timing analysis. Section 5.5.6 evaluates two hang-detection methods used on an application that fails to complete.

The designs used in the case studies occupy a relatively small part of the FPGA (24% of logic used in Section 5.5.5). Designs with higher resource utilization may lead to greater performance degradation and resource overhead of assertions due to increased difficulty in placement and routing for example. In addition, resource replication might not be applicable for designs that are almost full.

5.5.1 Detecting Simulation Inconsistencies

In this section, we illustrate how assertions can be used for in-circuit verification and debugging to catch inconsistencies between software simulation and FPGA execution of an application. The code in Figure 5-7 shows how assertion statements can be used for in-circuit verification by identifying bugs not found using software simulation. The first assertion is used to detect a translation mistake from source code to hardware.\(^1\) The assertion statement (line 6) never fails in simulation but fails when executed on the XD1000 platform. Upon inspection of the generated HDL, it is observed that Impulse C performs an erroneous 5-bit comparison of c2 and c1 (line 4). The 64-bit comparison of

\(^1\) It is possible for a translation mistake to also have an effect on an assertion
4294967286 > 4294967296 (which evaluates to false) becomes a 5-bit comparison of 22
> 0 (which evaluates to true), allowing the array address to become negative (line 4). In
contrast, the simulator executing the source code on the CPU sets the address to zero (line 5). Impulse C will generate a correct comparison when c1 and c2 are 32-bit variables.

The second assertion (line 8) is used to check the output of an external HDL function (line 7), which is used to gain extra performance over HLS generated HDL. When an
external HDL function is used, the developer must provide a C source equivalent for
software simulation. However, the behavior and timing of the C source for simulation may
differ from the behavior of the external HDL function during hardware execution, again
demonstrating a need for in-circuit verification.

```c
1 co_uint64 c2, c1;
2 co_int32 address, array[20], out;
3 c2 = 4294967286; c1 = 4294967296;
4 if (c2 > c1) address = c2 - c1;
5 else address = 0;
6 assert(address >= 0);
7 out = user(address);
8 assert((30 > out) && (out > 20));
9 array[address] = out;
```

For demonstration purposes, this example case is intentionally simplistic. Similar
conclusions could be drawn using a cycle-accurate HDL simulator. However, in practice,
inconsistencies caused by the timing of interaction between the CPU and FPGA would be
very difficult to model in a cycle-accurate simulator.

5.5.2 Assertion Parallelization Optimization

This section provides results for the parallelization optimization of assertions. Section
5.5.2.1 shows improvements from optimization for Triple-DES encryption. Section 5.5.2.2
shows optimization improvements for edge-detection. While the applications in the
previous sections evaluate frequency overhead, Section 5.5.2.3 evaluates state machine
performance overhead (in terms of additional cycles) and optimization improvements.
5.5.2.1 Triple Data Encryption Standard (DES) case study

The first application case study shows the area and clock frequency overhead associated with adding performance optimized assertion statements to a Triple-DES [60] application provided by Impulse C, which sends encrypted text files to the FPGA to be decoded. Two assertion statements were added in a performance critical region of the application to verify that the decrypted characters are within the normal bounds of an ASCII text file. Table 5-1 shows all sources of overhead, including the streaming communication channels generated by Impulse C for sending failed assertions back to the CPU. The overhead numbers were found to be quite modest, with resource usage increasing by at most 0.12% of the device and the maximum clock frequency dropping by less than 4 MHz.

For this case study, the optimized assertions were checked in a separate pipeline process to reduce the overhead generated by the assertion comparison. Assertion failures are sent by another process to ensure that assertions can be checked each cycle. The state machine of the application remained unchanged because the optimized assertions were checked in a separate task working in parallel with the application. Since the application’s state machine remained the same, the only performance overhead comes from the maximum clock frequency reduction. The resource overhead for optimized assertions actually decreased as compared to unoptimized assertions. The ALUT (Adaptive Look-Up Table) and routing resources needed by Quartus to achieve a maximum frequency of 144.7 MHz for unoptimized assertions was 0.06% greater than the ALUT and routing resources need for optimized assertions that achieved a maximum frequency of 142 MHz.

5.5.2.2 Edge-detection Case Study

The following case study integrates performance optimized assertions into an edge-detection application. The edge-detection application, provided by Impulse C, reads a 16-bit grayscale bitmap file on the microprocessor, processes it with pipelined $5 \times 5$ image kernels on the FPGA, and streams the image containing edge-detection information back.
Since the FPGA is programmed to process an image of a specific size, two assertions were added to check that the image size (height and width) received by the FPGA matches the hardware configuration. The assertions were added in a region of the application that was not performance critical. As shown in Table 5-2, the overhead numbers for this case study were also modest, with resource usage increasing by at most 0.06% on the EP2S180.

For the edge-detection case study, the optimized assertions were checked in a separate process to reduce the overhead generated by the assertion comparison. Since the applications state machine remained the same and maximum clock frequency did not reduce, the application did not incur any performance overhead due to the addition of the assertions. The frequency increase is likely due to randomness in placement and routing results of similar designs. The performance optimization of the assertions increased ALUT resource utilization from 0.03% to 0.06% on the EP2S180.

### 5.5.2.3 State Machine Overhead Analysis

This section presents a generalized analysis of performance overhead caused by adding assertions with a single comparison and the performance improvement via
optimizations. The results in this section present overhead in terms of cycles, and exclude changes to clock frequency, which was discussed in the previous section. We evaluate single comparison assertions to determine a lower bound on the optimization improvements. To measure the performance overhead of adding assertions, we examine the state machines and pipelines generated by Impulse C. Impulse C allows loops (e.g., for loops or while loops) to be pipelined. Assertions added to a pipeline can modify the pipeline’s characteristics. Each pipeline generated by Impulse C has a latency (time in cycles for one iteration of a loop to complete) and rate (time in cycles needed to finish the next loop iteration). Assertions that are not in a pipelined loop will add latency (i.e., one or more additional states) to the state machine that preserves the control flow of the application. As stated in Section 5.4.2, assertions can be optimized to reduce or eliminate the overhead of assertions in terms of additional clock cycles required to finish application execution. These optimizations move the comparisons to a separate Impulse C process so that they can be checked in parallel with the application. Any remaining clock cycle overhead after optimization comes from the data movement needed for assertion checking.

<table>
<thead>
<tr>
<th>EP2S180</th>
<th>Original</th>
<th>Assert</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Used</td>
<td>12250</td>
<td>12273</td>
<td>+23</td>
</tr>
<tr>
<td>(out of 143520)</td>
<td>(8.54%)</td>
<td>(8.56%)</td>
<td>(+0.02%)</td>
</tr>
<tr>
<td>Comb. ALUT</td>
<td>6726</td>
<td>6809</td>
<td>+83</td>
</tr>
<tr>
<td>(out of 143520)</td>
<td>(4.69%)</td>
<td>(4.75%)</td>
<td>(+0.06%)</td>
</tr>
<tr>
<td>Registers</td>
<td>9371</td>
<td>9417</td>
<td>+46</td>
</tr>
<tr>
<td>(out of 143520)</td>
<td>(6.53%)</td>
<td>(6.56%)</td>
<td>(+0.03%)</td>
</tr>
<tr>
<td>Block RAM</td>
<td>141120</td>
<td>141696</td>
<td>+576</td>
</tr>
<tr>
<td>(9383040 bits)</td>
<td>(1.50%)</td>
<td>(1.51%)</td>
<td>(+0.01%)</td>
</tr>
<tr>
<td>Block Interconnect</td>
<td>19904</td>
<td>19994</td>
<td>+90</td>
</tr>
<tr>
<td>(out of 536440)</td>
<td>(3.71%)</td>
<td>(3.73%)</td>
<td>(+0.02%)</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>77.5</td>
<td>79.3</td>
<td>+1.8</td>
</tr>
<tr>
<td></td>
<td>(+2.32%)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5-3 shows the latency overhead for non-pipelined, single comparison assertions. In most cases, assertions with these comparisons will increase latency by one cycle. With optimizations, this latency overhead is reduced to zero since extracting data in most cases will not add latency to the application. In the case where an array is consecutively accessed temporally by the application and an assertion, an unoptimized assertion will have a latency overhead of two cycles because of block RAM port limitations. With optimizations, this latency overhead is reduced to one cycle to extract data from the array or block RAM. For more complex assertions, the latency will increase for unoptimized assertions while the latency for optimized assertions will remain the same, as seen when comparing Figure 5-2B and Figure 5-2C. Even with the multiple array accesses in \( \text{assert}(\langle j <= 0 \parallel a[0] == i \rangle \& \& \langle b[0] == 2 \parallel i > 0 \rangle) \), only one cycle is needed to retrieve the array data.

Table 5-4 shows pipeline latency and rate overhead observed for a single comparison. Adding an unoptimized assertion using a scalar variable to a pipelined loop increased the latency from 2 to 3, resulting in an overhead of one cycle, and degraded the rate from 1 to 2 for the pipeline. Although the rate overhead was a single cycle, this corresponds to a 2x slowdown in performance because the throughput is reduced to half of the original loop. This overhead comes from adding a streaming communication call. For the optimized assertion, the streaming communication call was moved to a separate process that reduced the latency and rate overhead to zero, resulting in a 2x speedup compared to the unoptimized assertions. For assertions using arrays in pipelined loops, adding an assertion caused a 2 cycle latency overhead that increased the latency from 2 to 4. The assertion reduced the rate from 2 to 3, which is a one cycle rate overhead that corresponds to a 50% reduction in performance.

5.5.3 Resource Replication Optimization

As mentioned in Section 5.5.2.3, Table 5-4 shows pipeline latency and rate overhead observed for a single comparison. For assertions used in pipelined loops checking an array
Table 5-3. Single-comparison assertion

<table>
<thead>
<tr>
<th>Assertion data structure</th>
<th>Latency Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unoptimized</td>
</tr>
<tr>
<td>Scalar variable</td>
<td>1</td>
</tr>
<tr>
<td>Array (non-consecutive)</td>
<td>1</td>
</tr>
<tr>
<td>Array (consecutive)</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5-4. Pipelined single-comparison assertion

<table>
<thead>
<tr>
<th>Assertion data structure</th>
<th>Latency Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unoptimized</td>
</tr>
<tr>
<td>Scalar variable</td>
<td>1</td>
</tr>
<tr>
<td>Array</td>
<td>2</td>
</tr>
</tbody>
</table>

data structure, the assertion overhead was reduced via resource replication by adding an additional array to the process dedicated read access to the assertion as described in Section 5.4.2. With a duplicate array, only the latency increased from 2 to 3 and the rate remained the same which corresponds to a 33% rate improvement over the non-optimized version. A similar improvement could be gained for a non-pipelined assertion that checks multiple indexes to the same array.

5.5.4 Resource Sharing Optimization

This section demonstrates the improvement in scalability from resource sharing optimization techniques. We evaluate scalability by measuring the resource and clock frequency overhead incurred by adding assertions to a large number of Impulse C processes, providing an extremely pessimistic scenario in terms of overhead. A single assertion is added per process which results in a separate streaming communication channel for each process. A single greater than comparison is made per process, generally requiring only minor changes to the process state machine. In this study, the application consists of a simple streaming loopback as shown in Figure reffig:loopback. The loopback also stores the value and retrieves the value at each stage. Each process added to the
application adds an extra stage in the loopback (e.g., for 4 FPGA processes shown as L in Figure 5-8, incoming data would be passed from the input to the FPGA, passing through each of the processes before being returned to the CPU). The assertion in each process ensures the number being passed is greater than zero. Each process adds overhead in terms of an assertion shown as A in Figure 5-8 and an extra Impulse C streaming communication channel shown as C in Figure 5-8 to notify the CPU of failed assertions.

For a 32-bit stream, up to 32 assertions can be connected to the streaming communication channel before a new streaming communication channel is needed.

Figure 5-8. Simple streaming loopback

The previously discussed straightforward conversion of assert statements to if statements was used. The unoptimized assertions with 128 processes (128 assertions) had a resource overhead on the EP2S180 of 4.07% ALUTs (the highest resource percentage overhead). However, the maximum frequency decreased from 190 MHz for the 128-process original application to 154 MHz or an 18.8% overhead as shown in Figure 5-9 for the 128-process application with unoptimized assertions.

By applying the resource sharing optimization only to the communication channels so that only a single bit of the stream is used per assertion as described in Section 5.4.2 (and not the assertion resources), the resource overhead was decreased. The resource overhead on the EP2S180, as shown in Figure 5-10, was reduced to 1.34% of ALUTs or over a 3x improvement for the 128-process application with assertions. Assertion
optimizations increased the maximum frequency for the 128-process application to 189 MHz, as shown in Figure 5-9, which represents over an 18% improvement. The frequency of the application with assertion optimizations (189.3 MHz) was very close to the original application’s frequency of 190.6 MHz. While the resource usage increased consistently for all three tests (original, unoptimized, and optimized) from 1 to 128 processes, the maximum frequencies reported by Quartus did not consistently decrease as the number of processes increased until 32 processes were added. The frequency overhead decreased from 32 to 128 processes with optimized assertions because the application added one stream per process while the assertions only added one stream per 32 processes since 32-bit streaming communication was used. This demonstrates the benefits of the resource sharing optimization for streaming communication channels.

![Figure 5-9. Assertion frequency scalability](image)

### 5.5.5 In-circuit Timing Analysis

This section provides a case study showing the utility and overhead of adding assertions with timing statements to a backprojection application. Backprojection is a DSP algorithm for tomographic reconstruction of data via image transformation. For the
backprojection application, instrumentation was added into a nested loop (see Figure 5-11). Two 32-bit timing calls were added around the inner-pipelined loop to measure the time required for the pipelined loop to finish generating 512 pixels. After the timing calls, ten assertions were added to find the maximum time required for the pipelined loop to finish for all outer-loop iterations. Since the inner loop has 512 iterations, a minimum of 512 cycles should be needed to complete the loop, however, more cycles could be required for stalls and flushing of the pipeline. To test these assumptions, ten assertions were added to check the timing of the loop with exponentially increasing maximum times and NABORT was defined to stop the application from aborting. After execution, only the first assertion passed evaluation, which means that the maximum time for the inner loop is between 640 and 1023 cycles.

This technique allows the application designer to quickly check timing in multiple regions of the application with minimal disturbance to the application in terms of resource and communication overhead. After evaluating the feedback from the assertions, the application designer can modify the application to stream back the exact timing values for
problematic regions of code. In addition, the assertion feedback provided before modifying the application can be used to make sure that the timing values streamed back are valid. It is possible that the addition of large data transfers could change the timing of the application.

```c
for(y=0;y<512;y++)
{
    time1=clock();
    for(x=0;x<512;x++)
    {//compute pixel
        ...
    }
    time2=clock();
    assert((time2-time1)<1024));
    assert((time2-time1)<640));
    assert((time2-time1)<576));
    assert((time2-time1)<544));
    assert((time2-time1)<528));
    assert((time2-time1)<520));
    assert((time2-time1)<516));
    assert((time2-time1)<514));
    assert((time2-time1)<513));
    assert((time2-time1)<512));
    ...
}
```

Figure 5-11. Adding timing assertions individually to backprojection

```c
int32 constraint[]={1024,640,576,544,528,520,516,514,513,512};
...
for(y=0;y<512;y++)
{
    time1=clock();
    for(x=0;x<512;x++)
    {//compute pixel
        ...
    }
    time2=clock();
    for (i=0; i<10; i++) { 
        assert(time2-time1< constraint[i]);
    }
    ...
}
```

Figure 5-12. Adding timing assertions in a loop to backprojection
Table 5-5. Individual backprojection timing assertion overhead

<table>
<thead>
<tr>
<th>EP2S180</th>
<th>Original</th>
<th>Assert</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Used</td>
<td>48285</td>
<td>49702</td>
<td>+1417</td>
</tr>
<tr>
<td>(out of 203520)</td>
<td>(23.72%)</td>
<td>(24.42%)</td>
<td>(+0.70%)</td>
</tr>
<tr>
<td>Comb. ALUT</td>
<td>32962</td>
<td>33132</td>
<td>+170</td>
</tr>
<tr>
<td>(out of 203520)</td>
<td>(16.20%)</td>
<td>(16.28%)</td>
<td>(+0.08%)</td>
</tr>
<tr>
<td>Registers</td>
<td>44098</td>
<td>44595</td>
<td>+497</td>
</tr>
<tr>
<td>(out of 203520)</td>
<td>(21.67%)</td>
<td>(21.91%)</td>
<td>(+0.24%)</td>
</tr>
<tr>
<td>Block RAM</td>
<td>7114752</td>
<td>7114752</td>
<td>0</td>
</tr>
<tr>
<td>(15040512 bits)</td>
<td>(47.30%)</td>
<td>(47.30%)</td>
<td>(0%)</td>
</tr>
<tr>
<td>Block Interconnect</td>
<td>101317</td>
<td>102740</td>
<td>+1423</td>
</tr>
<tr>
<td>(out of 694728)</td>
<td>(14.58%)</td>
<td>(14.79%)</td>
<td>(+0.20%)</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>131.9</td>
<td>132.5</td>
<td>+0.6</td>
</tr>
</tbody>
</table>

The backprojection application runs on all four Stratix-III EP3SE260 FPGAs on the GiDEL PROCStar III [10] card. Overhead is only given for one FPGA since the image is split between all four FPGAs. Ideally, a single assertion could check an array of values in a loop for more compact code (see Figure 5-12). However, that approach increases overhead when synthesized with Impulse C as shown in Table 5-6 as compared to using individual assertions as shown in Table 5-5. For individual assertions, no additional block RAM was used since assertion failures were transferred via registers rather than using streaming communication on the PROCStar III. The logic overhead of 0.7% is the highest of all the application case studies but is reasonable given that timing calls and multiple assertions were used. The maximum FPGA frequency stayed about the same with an insignificant increase of 0.6 MHz. For a single assertion in a loop, the overhead increased in all categories except for routing. The additional overhead is likely caused by additional complexity of the state machine and the usage of block RAM. The lower routing overhead is probably due to only having to make connections to a single assertion.
### Table 5-6. Looped backprojection timing assertion overhead

<table>
<thead>
<tr>
<th>EP2S180</th>
<th>Original</th>
<th>Assert</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Used</td>
<td>48285</td>
<td>50169</td>
<td>+1884</td>
</tr>
<tr>
<td>(out of 203520)</td>
<td>(23.72%)</td>
<td>(24.65%)</td>
<td>(+0.93%)</td>
</tr>
<tr>
<td>Comb. ALUT</td>
<td>32962</td>
<td>33459</td>
<td>+497</td>
</tr>
<tr>
<td>(out of 203520)</td>
<td>(16.20%)</td>
<td>(16.44%)</td>
<td>(+0.24%)</td>
</tr>
<tr>
<td>Registers</td>
<td>44098</td>
<td>44657</td>
<td>+559</td>
</tr>
<tr>
<td>(out of 203520)</td>
<td>(21.67%)</td>
<td>(21.94%)</td>
<td>(+0.27%)</td>
</tr>
<tr>
<td>Block RAM</td>
<td>7114752</td>
<td>7123968</td>
<td>9216</td>
</tr>
<tr>
<td>(15040512 bits)</td>
<td>(47.30%)</td>
<td>(47.37%)</td>
<td>(0.07%)</td>
</tr>
<tr>
<td>Block Interconnect</td>
<td>101317</td>
<td>102621</td>
<td>+1304</td>
</tr>
<tr>
<td>(out of 694728)</td>
<td>(14.58%)</td>
<td>(14.77%)</td>
<td>(+0.19%)</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>131.9</td>
<td>131.3</td>
<td>-0.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(-0.45%)</td>
</tr>
</tbody>
</table>

### 5.5.6 Hang Detection

This section shows how in-circuit assertions can be used to detect when an application fails to complete (i.e., hangs), even when software simulation runs to completion. In an effort to speedup a decoder and encoder version of the DES application described in Section 5.5.2.1, modifications were made that caused the application to complete in software simulation and yet hang on the XD1000. Since Impulse C does not support printf in hardware, assertions were used to provide a heartbeat and “trace” the execution of process on the FPGA. Although this is not a common use of assertions in software, it can be useful to use assertions as a positive indicator rather than a negative indicator when an application is known to crash or hang. Assert(0) statements were placed at important points in the code for each FPGA process and NABORT was defined to stop the application from aborting. The new code with assertions added was executed via both software simulation and execution on the target platform. After comparing the line numbers of the failed assertions of both runs, it was found that the hang occurred at a memory read, which was causing the process to hang instead of exiting a loop. By
identifying the problematic line of code using in-circuit assertions, we were able to debug the application and determined that the memory read should have been a memory write. This correction allowed the process to complete execution.

Next, automated hang detection was used on the same problematic DES application. The software hang detector was triggered by the timeout of a communication call. The line number of the software API call was reported back along with the line number (taken before the API call was made) that the hardware process was currently executing. Although hardware hang detection was working correctly in the FPGA, the hardware hang detector was not able to notify the application designer of the problematic line of code since the software API call in conjunction with the erroneous line in the hardware process caused all communication between the CPU and FPGA to stop. To solve this problem, a sleep of one second was place above the software API call that was notified as being hung in previous run. The addition of the sleep allowed the hardware hang detector to report back the exact line number for the memory read that should have been a memory write.

The resource overhead of using automatic hang detection on the Triple-DES application is shown in Table 5-7. Hang detection had the highest, but still reasonable, percentage of ALUT (0.32%) and routing (0.25%) overhead because of the comparisons and connections made to the state machine of the encoder and decoder hardware process. The assertion pragma, #pragma assert_FPGA_watch_dog, was set to a timeout of a hundred million cycles which needed a 30-bit timing register. When using a 64-bit register, the frequency overhead increased to 5.7%. However, such overhead is very pessimistic because even with a 10 GHz clock speed, a 64-bit register supports a maximum timeout of about 58 years. For more typical cases, the frequency overhead should be less than 5.7%.

5.5.7 Assertion Limitations

The main limitation of in-circuit assertions is that overhead is dependent on the complexity of the assertion statements. For example, a designer could potentially verify a signal processing filter using an assertion statement that performs an FFT and then checks
Table 5-7. Data Encryption Standard (DES) hang-detection overhead

<table>
<thead>
<tr>
<th></th>
<th>EP2S180</th>
<th>Original</th>
<th>Assert</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Used (out of 143520)</td>
<td>21051 (14.67%)</td>
<td>21739 (15.15%)</td>
<td>+688 (+0.48%)</td>
<td></td>
</tr>
<tr>
<td>Comb. ALUT (out of 143520)</td>
<td>12986 (9.05%)</td>
<td>13440 (9.36%)</td>
<td>+454 (+0.32%)</td>
<td></td>
</tr>
<tr>
<td>Registers (out of 143520)</td>
<td>13884 (9.67%)</td>
<td>14015 (9.77%)</td>
<td>+121 (+0.09%)</td>
<td></td>
</tr>
<tr>
<td>Block RAM (9383040 bits)</td>
<td>149184 (1.59%)</td>
<td>149184 (1.59%)</td>
<td>0 (0%)</td>
<td></td>
</tr>
<tr>
<td>Block Interconnect (out of 536440)</td>
<td>38924 (7.26%)</td>
<td>40241 (7.50%)</td>
<td>+1317 (+0.25%)</td>
<td></td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>78.8</td>
<td>77.0</td>
<td>-1.80 (-2.28%)</td>
<td></td>
</tr>
</tbody>
</table>

To see if a particular frequency is below a pre-defined value. In this case, the synthesized assertion would contain a circuit for an FFT, which could have a large overhead. Note that such overhead is not a limitation of the presented synthesis techniques, but rather a fundamental limitation of in-circuit assertions.

To minimize this overhead, we suggest certain coding practices. Whenever possible, designers should use assertion statements that compare pre-computed values. Designers should try to avoid consolidating assertions in loops with comparison values stored in arrays because the unnecessary usage of arrays and loops with assertions can increase overhead as shown in Section 5.5.5. Designers should try to avoid using many logical operators because these operators can cause the HLS tool to create a large state machine to check all combination possibilities of the assertion as shown in Figure 5-2B. By following these guidelines, the assertions will require a minimum amount of resources. Assertion parallelization optimization and resource replication optimization can increase the resource overhead to reduce the performance overhead. Accessing the same array multiple times in an assertion (e.g. assert(a[i] > a[i - 1])) can be costly either in terms
of performance or resource depending if resource replication optimization is used. Even accessing an array only once in an assertion could be costly if the application would normally be using the same array element in the same clock cycle.

5.6 Conclusions

High-level synthesis tools often rely upon software simulation for verification and debugging executing FPGA processes as threads on the CPU. However, FPGA programming bugs not exposed by software simulation become difficult to remedy once the application is executing on the target platform. Similarly, HLS tools often lack detailed timing-analysis capabilities, making it difficult for an application designer to determine which regions of an application do not meet timing constraints during FPGA execution. The assertion-based verification techniques presented in this chapter provide ANSI-C-style verification both for the FPGA and CPU while in simulation and when executing on the target platform. This approach allows assertions to be seamlessly transferred from simulation to execution on the FPGA without requiring the designer to understand HDL or cycle-accurate simulators. The ability of assertions to verify a portion of the application’s functionality and debug errors not found during software simulation was demonstrated. ANSI-C timing functions allowed assertions to check application time constraints during execution. Automated hang detection provided source information indicating where software or hardware processes failed to complete in a timely manner. Techniques were shown to enable debugging of errors not found during software simulation that incurred a small area overhead of 0.7% or less and a maximum clock frequency overhead of less than 3% for several application case studies on an EP2S180 and EP3SE260. The presented techniques were shown to be highly scalable, reducing resource overhead of 128 assertions by over 3x, requiring only 1.34% ALUT resources and improving clock frequency by over 18%. The performance overhead of optimized assertions was also demonstrated to be low, with no performance impact observed in the edge-detection case study in terms of frequency degradation or increased
cycle usage. A general analysis of performance for single comparison assertions showed that the presented optimizations resulted in a throughput increase ranging from 33% to 100%, when compared to unoptimized assertions, potentially eliminating all throughput overhead. Future work includes further exploration and automation of hang detection.
CHAPTER 6
CONCLUSIONS

Using HDL tools for performance analysis and verification of HLS applications can be difficult. The most obvious challenge is correlating results from HDL to HLS source code. HLS tools typically provide correlation data in intermediate representation files. Although it is tedious for developers to read through such files, intermediate representation files can be parsed to provide source-line correlation and variable names linked to a particular state of a state machine. Another challenge that is specific to HLS is data extraction of results. Use of the HLS communication calls was chosen over JTAG for ease of use and wider platform support. For HLS tools that do not have open-source support, a method of generating a communication loopback in hardware is used. The loopback stub in HDL is then overwritten and used as an interface to gather results postmortem. Although many HLS tools strive to be ANSI-C compliant, most if not all HLS tools do not support the use of ANSI-C libraries. Part of this challenge was addressed by providing support for ANSI-C assertion and timing calls for verification and performance checking. Finally, the challenges of visualizing HLS applications were addressed. Visualizations from an existing performance analysis tool were leveraged and custom communication visualizations for bottleneck detection were created to help the developer optimize their application.

The challenges stated above were addressed in three phases. In Phase 1, a framework for performance analysis of HLS application was described. Techniques were developed for instrumenting, measuring, and presenting performance data of HLS applications. This framework provides automated profiling of HLS code. The state machines generated by the HLS tool were instrumented and cycle counts were measured to determine the timing of the HLS application. HLS profiling was integrated into the Parallel Performance Wizard tool to provide comprehensive performance analysis for multiple CPUs and FPGAs working together. Performance analysis visualization from PPW were also leveraged. This framework and tool is believed to be the first HLS performance analysis tool via
an extensive literature review. The performance analysis tool was used on a Molecular Dynamics application. The overhead of streaming transfers in the FPGA were found to be high. The streaming buffer size was increased which increased the speedup of the Molecular Dynamics application from 6.2 to 7.8 versus the serial baseline.

In Phase 2, a communication bottleneck visualization was explored. Instrumentation of communication API calls was added to monitor the bandwidth of the communication architecture. Methods to analyze the bandwidth characteristic of a communication architecture to suggest potential optimizations were presented. Graphviz was used to generate a multi-level visualization of the communication architecture of the application and state machines of each process. This framework and tool to visualize communication bottlenecks of HLS applications is believed to be the first of its kind via an extensive literature review. Several case studies were used to illustrate analysis techniques to identify communication bottlenecks. A speedup of 6.29 times was achieved for a Triple DES application by switching from streaming to DMA communication. Bottlenecks were also found in a Molecular Dynamics application and a Backprojection application.

In Phase 3, an assertion-based verification framework was described for HLS tools. The framework allows ANSI-C assertions to be synthesized into FPGA circuits, reporting valuable information concerning any assertion failures such as source-line information and the condition that failed. Hang detection was also explored for HLS applications. Watchdog timers were used to determine if the state machine hangs on a single state and software timers were used to detect hangs of API communication calls. The framework and tool for in-circuit assertion-based verification of HLS applications is believed to be the first of its kind via an extensive literature review. Several case studies demonstrate the low overhead and importance of assertion-based verification for HLS. Triple DES and edge detection were used as case studies for assertion parallelization optimization. Minimal overhead in terms of additional execution cycles and required FPGA resources was shown. In-circuit timing analysis was performed for Backprojection. Timing calls were added to
measure the time required for the pipelined loop to finish generating 512 pixels. Hang
detection was used on a problematic version of Triple DES. The hardware hang detector
was able to report back the exact line number of a memory read communication call that
should have been a memory write.

The contributions of this research include frameworks and tools for in-circuit
performance analysis and assertion-based verification of HLS applications. An HLS
framework was demonstrated for performance analysis and assertion-based verification
that is widely applicable to various RC platforms and HLS tools. The source-level results
given by these tools can increase developer productivity and reduce the need of RC
programmers to understand HDL. As the abstraction level of programming languages for
FPGAs increases and HDL programming becomes a footnote in history, there will be an
even greater importance for source-level productivity tools.

Future research includes extending tool coverage to HLS graphical tools or other
HLS programming languages besides C. Performance prediction was not explored in
this research. Performance prediction of HLS applications could be performed by a
combination of static HDL code analysis and dynamic analysis of the source code running
on a microprocessor. This analysis combination would allow performance analysis to take
place without requiring a lengthy place and route of HDL to the FPGA and be more
accurate prediction of performance than just executing the source code running on a
microprocessor. State machines would be analyzed for cycle counts of various execution
paths. The branches taken on the microprocessor would be monitored and the cycle counts
would be used to predict the execution time on the FPGA. Source collation would be used
to match branches in the source code and state machines. More advance techniques would
be required to model communication bottlenecks caused by streaming and DMA buffers.
APPENDIX A
TARGET SYSTEMS

A.1 ProcStar-III


A.2 XD1000

The XD1000 [8] is a reconfigurable system from XtremeData Inc. The XD1000 contains a dual-processor motherboard with an Altera Stratix-II EP2S180 FPGA on a module in one of the two Opteron sockets. The HyperTransport interconnect provides a communication channel between the FPGA and host processor with Impulse C [55].

A.3 SRC-7

SRC-7 [9] is a reconfigurable system by SRC Computers, Inc. with specialized internal network and memories. The SRC-7 model 71TS comes with Series-H MAP with EP2S180 FPGAs, a Hi-Bar switch and Global Common Memory. Carte C [56] can only execute on the SRC machines.
APPENDIX B
APPLICATION CASE STUDIES

B.1 Molecular Dynamics

MD simulates interactions between atoms and molecules over discrete time intervals. MD simulations take into account standard physics, Van Der Walls forces, and other interactions to calculate the movement of molecules over time. Alam et al. [62] provides a more in-depth overview of MD simulations. The simulation used in this document keeps track of 16,384 molecules, each of which uses 36 bytes (4 bytes to store its position, velocity, and acceleration in each of the X, Y, and Z directions). Analysis is focused on the kernel of the MD application that computes distances between atoms and molecules. Serial MD code optimized for traditional microprocessors was obtained from Oak Ridge National Lab (ORNL). This application runs on the XD1000 platform.

B.2 Backprojection

Backprojection is a DSP algorithm for tomographic reconstruction of data via image transformation. The application was originally designed for the XD1000 and ported to Novo-G. This application runs on all four FPGAs of the ProcStar-III board in the Novo-G supercomputer.

B.3 Triple Data Encryption Standard (DES)

Triple-DES [60] is a block cipher used for encryption. A modified version of the triple-DES application provided by Impulse C was used in this work. The application sends encrypted text files to the FPGA to be decoded. This application runs on the XD1000 platform.

B.4 Edge Detection

The edge-detection application was provided by Impulse C. The edge-detection application reads a 16-bit grayscale bitmap file on the microprocessor, processes it with pipelined 5x5 image kernels on the FPGA, and streams the image containing edge-detection information back. This application runs on the XD1000 platform.
REFERENCES


BIOGRAPHICAL SKETCH

John Curreri is a Ph.D. Candidate in the Department of Electrical and Computer Engineering Department. He received his Bachelors of Science degree in computer engineering from the University of Alabama in Huntsville and a Master of Science degree from University of Florida. He is a member of the translation and execution productivity group at the Center for High-Performance Reconfigurable Computing (CHREC). His current focus area is in the performance analysis and verification of High-Level Synthesis. He has also been in the Advanced Space Computing (ASC) group at the High-performance Computing and Simulation (HCS) Research Laboratory. In the ASC group, his focus area while working with Honeywell in National Aeronautics and Space Administration’s (NASA’s) New Millennium Project was system services. His research interests are reconfigurable, parallel and fault-tolerant computing. He is an active member of the UF student branch of Institute of Electrical and Electronics Engineers (IEEE) and a member of American Institute of Aeronautics and Astronautics (AIAA).