# Performance Analysis Framework for High-Level Language Applications in Reconfigurable Computing

JOHN CURRERI, SETH KOEHLER, ALAN D. GEORGE, BRIAN HOLLAND, and RAFAEL GARCIA
NSF Center for High-Performance Reconfigurable Computing (CHREC),
University of Florida

High-Level Languages (HLLs) for Field-Programmable Gate Arrays (FPGAs) facilitate the use of reconfigurable computing resources for application developers by using familiar, higher-level syntax, semantics, and abstractions, typically enabling faster development times than with traditional Hardware Description Languages (HDLs). However, programming at a higher level of abstraction is typically accompanied by some loss of performance as well as reduced transparency of application behavior, making it difficult to understand and improve application performance. While runtime tools for performance analysis are often featured in development with traditional HLLs for sequential and parallel programming, HLL-based development for FPGAs have an equal or greater need yet lack these tools. This paper presents a novel and portable framework for runtime performance analysis of HLL applications for FPGAs, including an automated tool for performance analysis of designs created with Impulse C, a commercial HLL for FPGAs. As a case study, this tool is used to successfully locate performance bottlenecks in a molecular dynamics kernel in order to gain speedup.

Categories and Subject Descriptors: B.8.2 [PERFORMANCE AND RELIABILITY] Performance Analysis and Design Aids
General Terms: Performance, Measurement
Additional Key Words and Phrases: FPGA, profile, trace, high-level language, high-level synthesis tools, Impulse C, Carte C

## 1. INTRODUCTION

High-level synthesis tools translate high-level languages (e.g., Impulse C [Pellerin et al. 2005] or Carte C [Poznanovic 2005]) to hardware configurations on FPGAs. Today's high-level languages (HLLs) simplify software developers' transition to reconfigurable computing and its performance advantages without the steep learning curve associated with traditional Hardware Description Languages (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, high-level synthesis tools support debugging at the HLL source-code level on a traditional microprocessor without performing translation from HLL to HDL. Current commercial high-level synthesis 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 HLL source-code level. In addition, research

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422.

Authors' addresses: J. Curreri, S. Koehler, B. Holland, R. Garcia and A. D. George, Larsen Hall, PO Box 116200, ECE Department, University of Florida, Gainesville, FL 32611-6200, email: {curreri, koehler, holland, garcia, george}@chrec.org.

Based on "Performance Analysis with High-Level Languages for High-Performance Reconfigurable Computing", by J. Curreri, S. Koehler, B. Holland, and A. D. George which appeared in Proc. of 16th IEEE Symposium on Field-Programmable Custom Computing Machines, © 2008 IEEE.

on runtime performance analysis[1] for FPGAs is lacking with few exceptions and none of which is targeted towards HLLs.

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 HLL source-code level to a software developer.

RC applications have potential for high performance, but HLL-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 paper focuses upon performance analysis of an HLL application on a reconfigurable system by monitoring the application at runtime. We have gained the majority of our insight about performance analysis with high-level languages from our experience 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. We develop a performance analysis framework based on Impulse C, prototyping this framework into an automated tool in order to demonstrate its effectiveness on a molecular dynamics application.

The remainder of this paper is organized as follows. Section 2 discusses related work and provides background information on runtime performance analysis. Next, Section 3 covers the challenges of performance analysis for HLLs targeting FPGAs. Section 4 provides details for our performance analysis framework for Impulse C. Section 5 then presents a case study using a molecular dynamics kernel written in Impulse C. Finally, Section 6 concludes and presents ideas for future work.

## 2. BACKGROUND & RELATED WORK

Performance analysis can be divided into five steps (derived from Maloney's work on the TAU performance analysis framework for traditional processors [Shende et al. 2006]) whose end goal is to produce an optimized application. These steps are Instrument, Measure, Analyze, Present, and Optimize (see Figure 1). 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

[1] Runtime performance analysis (timing analysis on data gathered during execution) is hereafter referred to as performance analysis.

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 including PPW [Su et al. 2008], TAU [Shende et al. 2006], KOJAK [Mohr et al. 2003] and HPCToolkit [Mellor-Crummey et al. 2002].

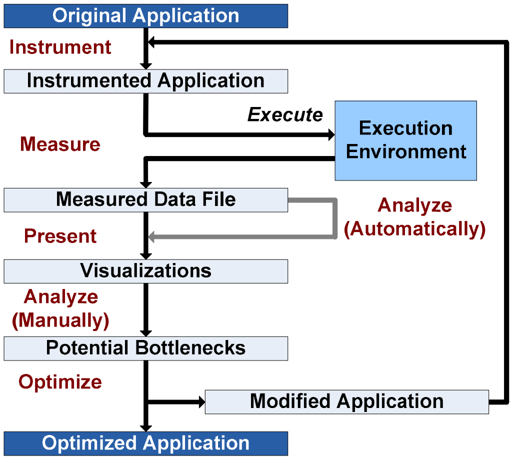


Figure 1. Performance analysis steps

To the best of our knowledge from a comprehensive literature search, little previous work exists concerning runtime performance analysis for FPGA applications. Hardware performance measurement modules have been integrated into FPGAs before; however, they were designed specifically for monitoring the execution of soft-core processors [Tong et al. 2007]. 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 [Schulz et al. 2005]. Range adaptive profiling has been prototyped on FPGAs but was not used for profiling an application executing on an FPGA [Mysore 2008]. Runtime debugging of an HLL, Sea Cucumber, has been developed by Hemmert et al. [2003]. However, the Sea Cucumber Debugger does not support performance analysis. Calvez et al. [1995] describe performance analysis for ASICs while DeVille et al. [2005] discuss performance monitoring probes for FPGA circuits; however, neither work targets HLLs.

This paper significantly extends our previous work on performance analysis for HDL applications [Koehler et al. 2008] by expanding this HDL framework to support the challenges of high-level synthesis tools. This paper also extends our previous work on performance analysis of HLL-based applications [Curreri et al. 2008] by expanding the scope of challenges faced in instrumentation, measurement, and visualization of these applications, detailing our framework for automated instrumentation and visualization, and providing results from a prototype incorporating this automation and visualization.

## 3. HLL PERFORMANCE ANALYSIS CHALLENGES

While all stages of performance analysis mentioned above are of interest for HLL-based applications, we limit our discussion to the challenges of instrumentation, measurement, analysis, and visualization for the remainder of this paper; optimization is beyond the scope of this paper. Thus, Section 3.1 covers the challenges of instrumenting an application, Section 3.2 explains the challenges associated with measuring performance data from an application, Section 3.3 discusses the challenges of analyzing performance data, and Section 3.4 examines the challenges of visualizing performance data.

## 3.1 HLL Instrumentation Challenges

Instrumentation, the first step of performance analysis, enables access to application data at runtime. For HLL-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, while the selection of what to monitor is covered in Section 3.1.2.

*3.1.1 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 HLL-based applications. For details on instrumentation levels below HDL, see Graham et al. [2001].

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, we choose to instrument at the HDL level in order to provide fine-grained performance analysis that would otherwise be impossible for HLLs that lack hardware timing functions. Even for HLLs 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 [Xilinx 2007] or Altera's SignalTap [Altera 2008].

*3.1.2 Instrumentation Selection.* Application performance can generally be considered in terms of communication and computation. Many HLLs, 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.

We chose instrumentation of state machines 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 HLL. 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.

### 3.2 HLL 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 3.2.1 presents standard techniques for measuring application data while Section 3.2.2 discusses the challenges of extracting measurement data.

*3.2.1 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 [Koehler et al. 2008]. Therefore, we incorporate their profiling and tracing measurement techniques in our framework.

*3.2.2 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 [IEEE Computer Society 2001] 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 HLLs 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, we have chosen data extraction 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 HLLs as well as between FPGA platforms using the same HLL, 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.

### 3.3 HLL 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 HLL-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.

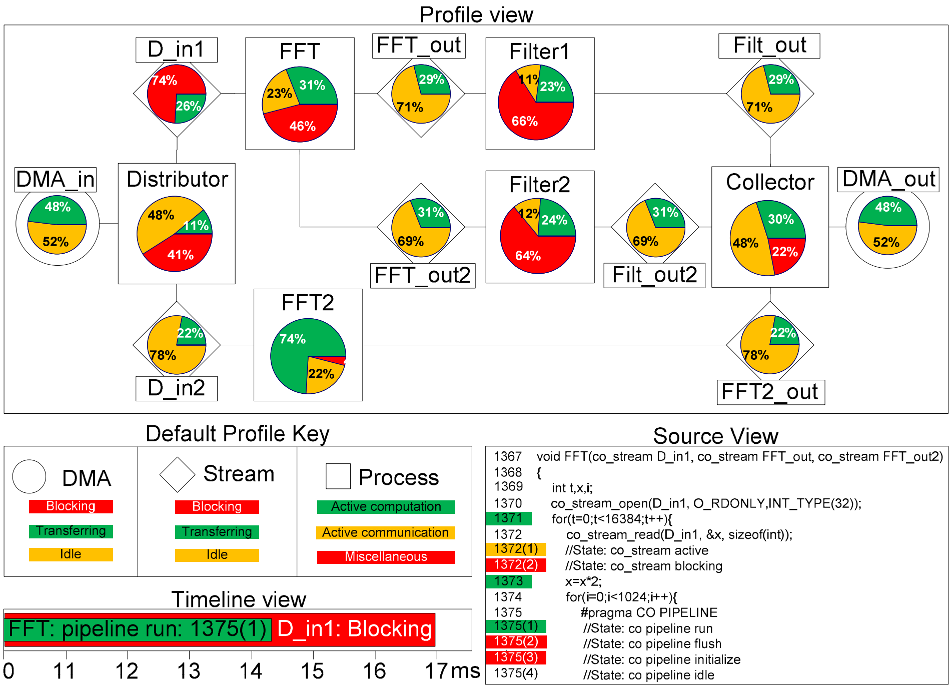
Integration of automatic analysis into our framework will be saved for future work. Further study on common reconfigurable computing bottlenecks in applications is needed in order to develop a general-purpose automatic analysis framework.

### 3.4 HLL 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 HLL-based applications, treating heterogeneous components and communication architectures as homogeneous.

Koehler et al. [2008] 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 HLL-based applications. HDL code generated by high-level synthesis tools has a predefined structure, making it more feasible to automatically generate meaningful visualizations for HLL-based applications.

Visualizations for HLL-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 2. 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 2). This type of representation allows all of the application profile data to be displayed in a single visualization while capturing the application's architecture.



```
1367    void FFT(co_stream D_in1, co_stream FFT_out, co_stream FFT_out2)
1368    {
1369       int t,x,i;
1370       co_stream_open(D_in1, O_RDONLY,INT_TYPE(32));
1371       for(t=0;t<16384;t++){
1372          co_stream_read(D_in1, &x, sizeof(int));
1372(1)       //State: co_stream active
1372(2)       //State: co_stream blocking
1373          x=x*2;
1374          for(i=0;i<1024;i++){
1375             #pragma CO PIPELINE
1375(1)           //State: co pipeline run
1375(2)           //State: co pipeline flush
1375(3)           //State: co pipeline initialize
1375(4)           //State: co pipeline idle
```

Figure 2. Example HLL 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 2). 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 2) 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 [Zaki 1999] would dedicate a row to each parallel node (in this case, each process shown in the Profile View of Figure 2 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.

## 4. HLL 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. 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. [2008], two main issues needed to be addressed: automation of instrumentation and integration with an existing software performance analysis tool. In Section 4.1, the methods needed to automate instrumentation are discussed. Section 4.2 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.

## 4.1 HLL 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 3.2.1 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 4.1.1 describes automated HLL instrumentation that inserts communication channels for measured performance data. Section 4.1.2 then presents an automated tool for HDL instrumentation in order to record and store application behavior at runtime.

*4.1.1 Automated C Source Instrumentation.* In order to communicate measured performance data from hardware back to software, our 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 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 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.

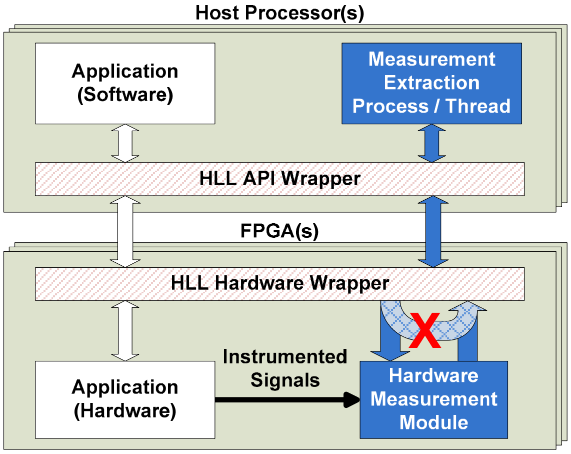


Figure 3. HLL communication loopback is added. The loopback channel’s HDL is replaced by the HMM, which monitors the application’s signals.

Figure 4 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 4; 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.

*4.1.2 Automated HDL Instrumentation.* Once the application is mapped to HDL code (step 3 in Figure 4), the HDL Instrumenter is employed, providing the application developer the choice between default and custom instrumentation (step 4 in Figure 4). 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.

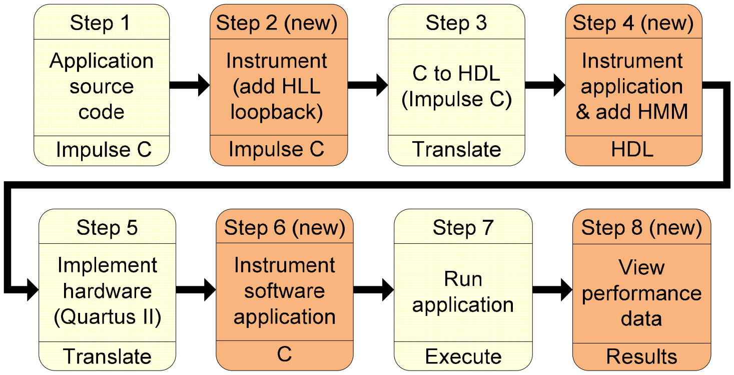


Figure 4. Design flow for Impulse C performance analysis

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). The loopback process is then replaced by the Hardware Measurement Module (HMM) shown in the lower dark box in Figure 3. The HMM contains customized hardware with profiling and tracing capabilities (see Figure 5) and was originally designed for HDL performance analysis [Koehler et al. 2008]. 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.

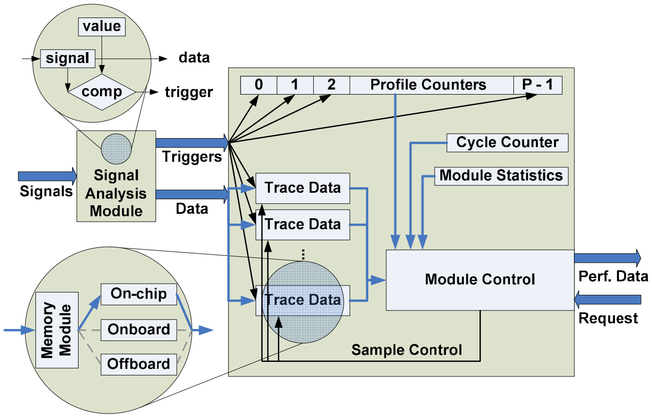


Figure 5. Hardware Measurement Module (HMM)

Once instrumentation is complete, the HDL is ready to be converted to a bitstream (step 5 in Figure 4) and programmed into the FPGA. In general, the techniques used in section 4.1 and Figure 3 should be valid for any HLL that employs communication channels and generates unencrypted and non-obfuscated HDL.

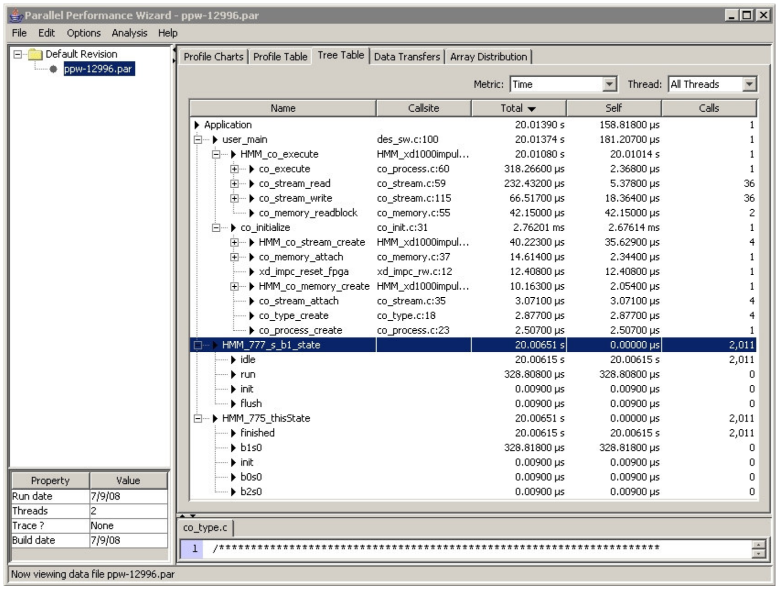
### 4.2 Performance Analysis Tool Integration

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, our Impulse C hardware performance analysis framework was integrated into an existing performance analysis tool, Parallel Performance Wizard (PPW) [Su et al. 2008], 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 4.2.1 explains how PPW+RC (i.e., PPW augmented with our framework for RC performance analysis) extracts performance data from the HMM. Section 4.2.2 then describes how measurement data for Impulse C hardware processes are visualized by PPW+RC.

*4.2.1 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.

*4.2.2 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 4.1.2), 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 6 shows the PPW+RC frontend displaying timing for software processes, hardware processes, and pipelines for a DES encryption application. 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.



Metric: Time
Thread: All Threads

| Name | Callsite | Total | Self | Calls |
|---|---|---|---|---|
| Application | | 20.01390 s | 158.81800 µs | 1 |
| user_main | des_sw.c:100 | 20.01374 s | 181.20700 µs | 1 |
| HMM_co_execute | HMM_xd1000impul... | 20.01080 s | 20.01014 s | 1 |
| co_execute | co_process.c:60 | 318.26600 µs | 2.36800 µs | 1 |
| co_stream_read | co_stream.c:59 | 232.43200 µs | 5.37800 µs | 36 |
| co_stream_write | co_stream.c:115 | 66.51700 µs | 18.36400 µs | 36 |
| co_memory_readblock | co_memory.c:55 | 42.15000 µs | 42.15000 µs | 2 |
| co_initialize | co_init.c:31 | 2.76201 ms | 2.67614 ms | 1 |
| HMM_co_stream_create | HMM_xd1000impul... | 40.22300 µs | 35.62900 µs | 4 |
| co_memory_attach | co_memory.c:37 | 14.61400 µs | 2.34400 µs | 1 |
| xd_impc_reset_fpga | xd_impc_rw.c:12 | 12.40800 µs | 12.40800 µs | 1 |
| HMM_co_memory_create | HMM_xd1000impul... | 10.16300 µs | 2.05400 µs | 1 |
| co_stream_attach | co_stream.c:35 | 3.07100 µs | 3.07100 µs | 4 |
| co_type_create | co_type.c:18 | 2.87700 µs | 2.87700 µs | 4 |
| co_process_create | co_process.c:23 | 2.50700 µs | 2.50700 µs | 1 |
| HMM_777_s_b1_state | | 20.00651 s | 0.00000 µs | 2,011 |
| idle | | 20.00615 s | 20.00615 s | 2,011 |
| run | | 328.80800 µs | 328.80800 µs | 0 |
| init | | 0.00900 µs | 0.00900 µs | 0 |
| flush | | 0.00900 µs | 0.00900 µs | 0 |
| HMM_775_thisState | | 20.00651 s | 0.00000 µs | 2,011 |
| finished | | 20.00615 s | 20.00615 s | 2,011 |
| b1s0 | | 328.81800 µs | 328.81800 µs | 0 |
| init | | 0.00900 µs | 0.00900 µs | 0 |
| b0s0 | | 0.00900 µs | 0.00900 µs | 0 |
| b2s0 | | 0.00900 µs | 0.00900 µs | 0 |

| Property | Value |
|---|---|
| Run date | 7/9/08 |
| Threads | 2 |
| Trace ? | None |
| Build date | 7/9/08 |

```
1  /***********************************************************************
```

Now viewing data file ppw-12996.par

Figure 6. PPW+RC profile tree table visualization for the DES application (Impulse C)

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, HLL 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 our 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 [Pellerin 2007]. 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.

## 5. MOLECULAR-DYNAMICS CASE STUDY

To demonstrate the benefits of HLL performance analysis and explore its associated overhead, we analyze a molecular-dynamics (MD) kernel written in Impulse C. 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. [2006] provides a more in-depth overview of MD simulations. Our 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). Our focus is on the kernel of the MD application that computes distances between atoms and molecules.

We obtained serial MD code optimized for traditional microprocessors from Oak Ridge National Lab (ORNL). We redesigned the MD code in Impulse C using an XD1000 [Altera 2007] as the target platform. The XD1000 is a reconfigurable system from Xtreme Data 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 our prototype performance analysis tool, we analyzed the performance of our MD code to determine if further speedup could be obtained.

There are three hardware processes defined in the MD hardware subroutine (Figure 7). 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.

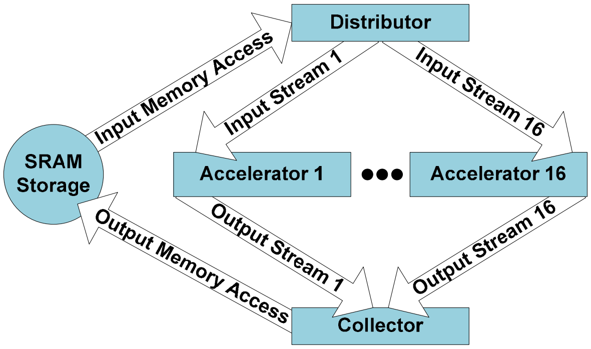
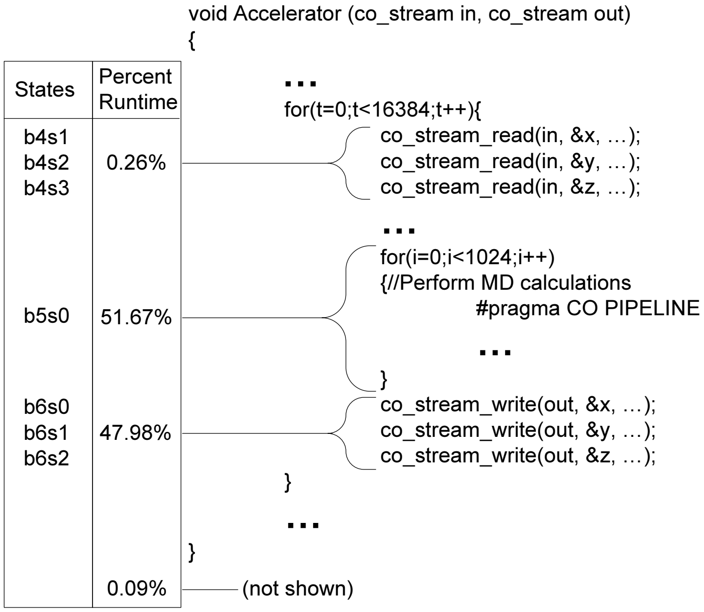


Figure 7. MD hardware subroutine

We instrumented and analyzed the MD kernel, with a focus on understanding the behavior of the state machine inside of each Accelerator process (Figure 8). 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 7) 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 7) 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 8) 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, we only present data from one of the Accelerator processes.



| States | Percent Runtime |
|---|---|
| b4s1<br>b4s2<br>b4s3 | 0.26% |
| b5s0 | 51.67% |
| b6s0<br>b6s1<br>b6s2 | 47.98% |
| | 0.09% |

```
void Accelerator (co_stream in, co_stream out)
{
    ...
    for(t=0;t<16384;t++){
        co_stream_read(in, &x, ...);
        co_stream_read(in, &y, ...);
        co_stream_read(in, &z, ...);
        ...
        for(i=0;i<1024;i++)
        {//Perform MD calculations
                #pragma CO PIPELINE
                ...
        }
        co_stream_write(out, &x, ...);
        co_stream_write(out, &y, ...);
        co_stream_write(out, &z, ...);
    }
    ...
}
(not shown)
```

Figure 8. Accelerator process source with profiling percentages

Our 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 8 and 9). 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 10 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 10) while the pipeline's runtime (bottom bar in Figure 10) 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.

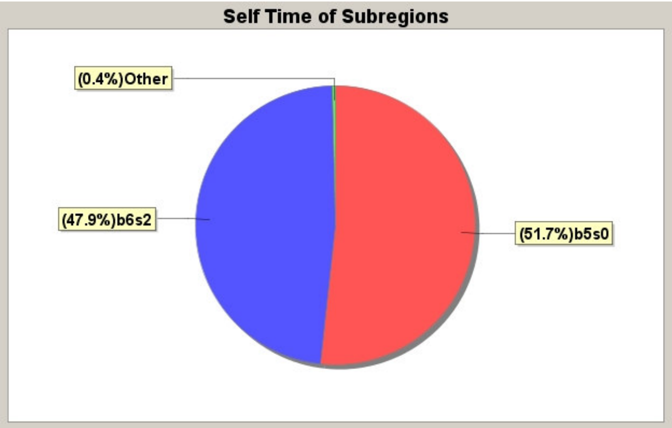


Figure 9. PPW+RC pie chart visualization of the states in Accelerator process. State b5s0 correspond to the pipeline and state b6s2 corresponds to the output stream..

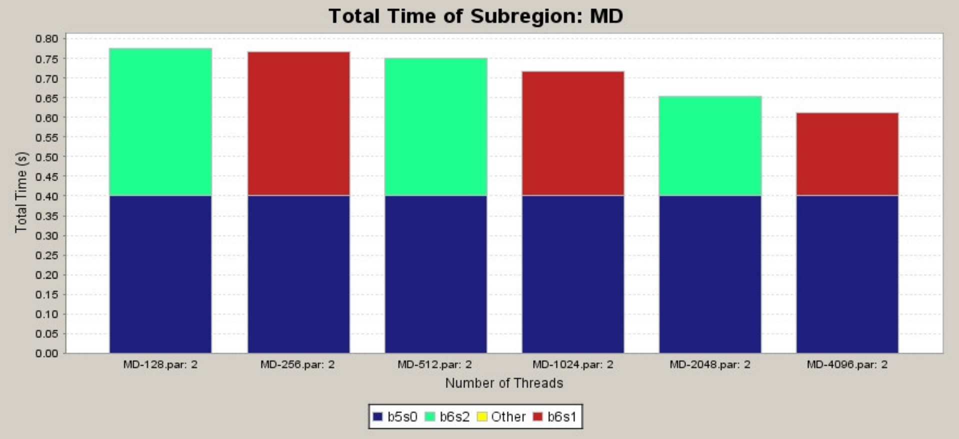


Figure 10. PPW+RC visualization depicting MD kernel performance improvement due to increased stream buffer size (128 to 4096 elements). State b5s0 corresponds to the pipeline in Accelerator while states b6s1 and b6s2 correspond to the output stream.

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 8) is performed once per molecule (16384 times). Only one of the three output transfer states in the loop generates idle cycles (Figure 10); 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 11 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.

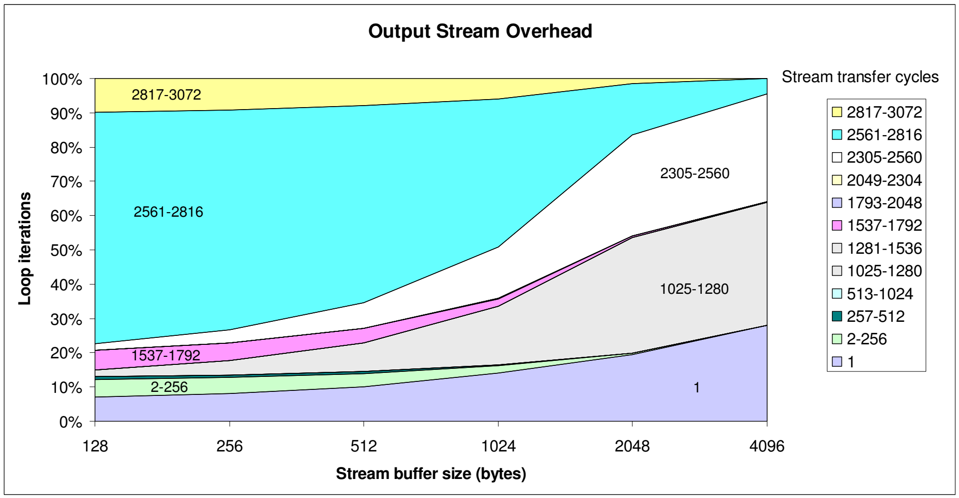


Figure 11. Output stream overhead for the Accelerator process.

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 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.

Table 1. Performance analysis overhead

| EP2S180 | Original | Instrumented | Overhead |
|---|---|---|---|
| Logic used<br>(143520) | 126252<br>(87.97%) | 131851<br>(91.87%) | +5599<br>(+3.90%) |
| Comb. ALUT<br>(143520) | 100344<br>(69.92%) | 104262<br>(72.65%) | +3918<br>(+2.73%) |
| Registers<br>(143520) | 104882<br>(73.08%) | 110188<br>(76.78%) | +5306<br>(+3.70%) |
| Block memory<br>(9383040 bits) | 3437568<br>(36.64%) | 3557376<br>(37.91%) | +119808<br>(+1.27%) |
| Block Interconnect<br>(536440) | 288877<br>(53.85%) | 300987<br>(56.11%) | +12110<br>(+2.56%) |
| Frequency (MHz) | 80.57 | 78.44 | -2.13<br>(-2.64%) |

## 6. CONCLUSIONS

High-level languages have the potential to make reconfigurable computing more productive and easier to use for application developers. While this higher-abstraction 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 HLL-based FPGA applications have been identified in this paper. 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. We discussed instrumenting control hardware employed to maintain program order, pipelines, and communication channels. 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. We also commented on the use of measured performance data for automatic bottleneck detection at the HLL source-code level to increase developer productivity. An HLL-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), our 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. We also observed low overhead (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, we noted that 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, our future plans are to link 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 HLL-based applications and expanding our tool to support performance analysis for Carte C.

## ACKNOWLEDGMENTS

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by Impulse Accelerated Technologies, Altera, XtremeData, SRC Computers, and Aldec.

## REFERENCES

ALAM, S. R., VETTER, J. S., AGARWAL, P. K., AND GEIST, A. 2006. Performance characterization of molecular dynamics techniques for biomolecular simulations. *Proc. 11th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM Press, New York, NY, 59-68.

ALTERA. 2007. Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform.

ALTERA. 2008. Design debugging using the SignalTap II embedded logic analyzer. http://www.altera.com/literature/hb/qts/qts_qii53009.pdf.

Calvez, J. P., and Pasquier, O. 1995. Performance Monitoring and Assessment of Embedded HW/SW Systems, *International Conference on Computer Design (ICCD): VLSI in Computers and Processors*, Austin, TX, 2-4.

CURRERI, J., KOEHLER, S., HOLLAND, B., AND GEORGE, A. D. 2008. Performance Analysis with High-Level Languages for High-Performance Reconfigurable Computing. *Proc. of 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Palo Alto, CA, 14-15.

DEVILLE, R., TROXEL, I., AND GEORGE, A. D. 2005. Performance monitoring for run-time management of reconfigurable devices, *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA).* Las Vegas, NV, 175-181.

GRAHAM, P., NELSON, B., AND HUTCHINGS, B. 2001. Instrumenting bitstreams for debugging FPGA circuits. *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE Computer Society, Rohnert Park, CA, 41-50.

HEMMERT, K. S., TRIPP, J. L., HUTCHINGS, B. L., AND JACKSON, P. A. 2003. Source Level Debugger for the Sea Cucumber Synthesizing Compiler. *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, 228-237.

IEEE COMPUTER SOCIETY. 2001. IEEE Standard Test Access Port and Boundary-Scan Architecture. IEEE Std 1149.1-2001 (R2008).

KOEHLER, S., CURRERI, J., AND GEORGE, A. D., 2008. Performance Analysis Challenges and Framework for High-Performance Reconfigurable Computing, *Parallel Computing 34, 217-230.*

MELLOR-CRUMMEY, J., FOWLER R. J., MARIN, G., AND TALLENT., N. 2002. HPCVIEW: A tool for top-down analysis of node performance. *The Journal of Supercomputing* 23, 81–104.

MOHR, B., AND WOLF, F. 2003. KOJAK – a tool set for automatic performance analysis of parallel applications. *European Conference on Parallel Computing (EuroPar)*, pages 1301–1304, Klagenfurt, Austria, LNCS 2790, 26–29.

MYSORE, S., AGRAWAL, B., NEUBER, R., SHERWOOD, T., SHRIVASTAVA, N. AND SURI, S. 2008. Formulating and Implementing Profiling over Adaptive Ranges, *ACM Transactions on Architecture and Code Optimization (TACO)*, 5, 1.

PELLERIN, D. 2007. Email transaction on adding comments to HDL for HLL source correlation. Dec 2007.

PELLERIN, D., AND THIBAULT, S. 2005. *Practical FPGA Programming in C.* Prentice Hall PTR.

POZNANOVIC, *D.* 2005. Application Development on the SRC Computers, Inc. Systems, *Proc. 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, 78a.

SU, H., BILLINGSLEY, M., AND GEORGE, A. D. 2008. Parallel Performance Wizard: A Performance Analysis Tool for Partitioned Global-Address-Space Programming. *9th IEEE International Workshop on Parallel & Distributed Scientific and Engineering Computing (PDSEC) of IPDPS*, Miami, FL, 14-15.

SCHULZ, M., WHITE, B. S., MCKEE, S. A., LEE, H. S., AND JEITNER, J. 2005. Owl: next generation system monitoring. *Proc. 2nd Conference on Computing frontiers (CF)*, ACM Press, New York, NY, 116-124.

SHENDE, S., AND MALONY, A. D. 2006. The Tau Parallel Performance System. *International Journal of High-Performance Computing Applications*, SAGE Publications 20, 287–311.

TONG, J. G., AND KHALID, M. A. S. 2007. A Comparison of Profiling Tools for FPGA-Based Embedded Systems. *Proc. Canadian Conference on Electrical and Computer Engineering (CCECE)*, Vancouver, British Columbia, Canada, 1687-1690.

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

Zaki, O., Lusk, E., Gropp, W., and Swider, D. 1999. Toward Scalable Performance Visualization with Jumpshot. *Int. J. High Perform. Comput. Appl.* 13, 3, 277-288.