Communication Visualization for Bottleneck Detection of High-Level Synthesis Applications

John Curreri, Greg Stitt, Alan George
NSF Center for High-Performance Reconfigurable Computing
Department of Electrical and Computer Engineering
University of Florida
{curreri, gstitt, george}@chrec.org

ABSTRACT
High-level synthesis tools increase FPGA productivity but can decrease performance compared to register-transfer level designs. To help optimize high-level synthesis applications, we introduce a bottleneck detection tool that provides a developer with a visualization of communication bandwidth between all application processes, while identifying potential bottlenecks via color coding. We evaluated the tool using third-party applications to identify and optimize bottlenecks in just several minutes, which achieved speedups ranging from 1.25x to 2.18x compared to the original FPGA execution. Overhead was modest with less than 2% resource overhead and 3% frequency overhead.

Categories and Subject Descriptors
B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids – *Verification*

General Terms
Measurement, Performance, Design, Verification.

Keywords
High-level synthesis; visualization; performance analysis; bottleneck detection.

1. INTRODUCTION
High-level synthesis (HLS) tools [2][8] increase productivity by allowing developers to program field-programmable gate arrays (FPGAs) using higher abstractions than register-transfer-level (RTL) code. However, higher abstractions can result in performance bottlenecks that are challenging for developers to identify due to a lack of visibility into the synthesized circuit structure and runtime behavior.

Although software developers commonly use performance analysis tools to identify bottlenecks, there is a lack of such tools for FPGAs, especially for circuits generated from high-level synthesis. One of the key capabilities missing from existing FPGA performance-analysis approaches is visualization of communication bandwidth between different application processes. Without this information, a developer often must guess at the problem in order to perform optimizations, or has to add extra code to profile the circuit, which would be impractical for large applications. Alternatively, the developer must analyze the synthesized RTL code, which is not practical for many developers.

In this paper, we enable such visualization and bottleneck detection for high-level synthesis. The presented tool provides the developer with a high-level bandwidth visualization of all communication between application processes for both the CPU and FPGA and graphically identifies bottlenecks via color coding. To measure bandwidth, the tool automatically adds hardware counters and software timers to the application code, executes the application, and then uses the measurements to generate the communication visualization of the application. Optimizing the bottlenecks shown by the visualization required only several minutes of developer effort with speedups ranging from 1.25x to 2.18x compared to the original FPGA application with a modest 2% resource overhead and 3% frequency overhead.

FPGA'12, February 22–24, 2011, Monterey, California, USA.

2. RELATED RESEARCH
Visualization and analysis tools have been heavily researched [6] for identifying software bottlenecks. To our knowledge, there are few studies on performance analysis of HLS-generated circuits. Performance analysis has been developed for ASICs by Calvez et al. [1] and FPGA circuits by DeVille et al. [4], but neither study targets HLS tools. Related work exists for performance analysis [7] of HDL applications, but that work is not appropritae for HLS applications due to lack of source-code correlation and bandwidth visualizations. Previous work exists for performance analysis of HLS applications and visualization using a modified version of PPW [3]. The techniques in this paper complement this previous timing-based performance analysis by providing instrumentation for bandwidth measurement and communication visualizations.

3. COMMUNICATION VISUALIZATION
This section discusses the visualizations and corresponding bottleneck detection and instrumentation techniques.

3.1 Visualizations
High-level synthesis generates an application-specific graph of processes mapped to different devices (e.g., FPGA, CPU, memory), where edges between processes correspond to communication. The presented visualizations display this graph along with measured bandwidths for each edge.

Although the proposed techniques potentially apply to any high-level synthesis tool with constructs equivalent to parallel

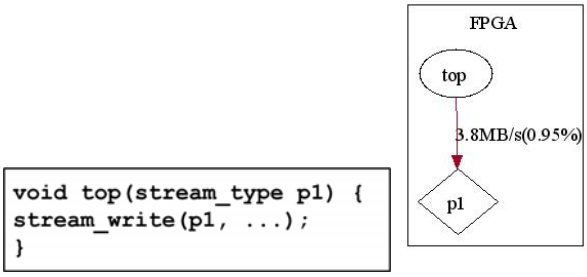

Figure 1: Streaming-communication call visualization

processes, in this paper we target Impulse-C. In Impulse-C, communication channels fall into two categories: streaming and DMA transfers. Streaming transfers use buffers between two communicating processes. Direct memory access (DMA) transfers move data to/from dedicated memory such as SRAM.

The visualization tool initially constructs a directed graph based on the source code of the application. The tool visualizes resources such as CPUs, FPGAs, and memories using boxes. The tool visualizes processes as nodes (i.e., ovals) that are placed into the box of the corresponding resource. For each communication API call in the application code, the tool creates a corresponding edge between nodes. Streaming buffers are shown as diamonds.

Figure 1 shows an example of streaming communication in Impulse-C, along with the corresponding visualization of the process *top* writing to a stream buffer *p1*. Figure 2 shows an example of a DMA-communication call inside a function and the corresponding visualization of an FPGA processs *top* writing to an SRAM using a buffer *datamemx*. Memory used for DMA is shown as a separate box from the CPU or FPGA and the buffer is displayed as a separate box inside of the corresponding memory.

As shown in Figure 3, the tool annotates each edge with the measured bandwidths, as discussed in Section 3.3, in addition to the percentage of maximum possible bandwidth. Note that the percentages correspond only to the time that each process transfers data. For example, a process may execute for 100 cycles without data transfers, which would not affect the measured bandwidth. Therefore, a percentage of 100% does not necessarily suggest a saturated channel, but rather the efficient use of the channel. For example, if multiple processes read from a single memory at different times, both read channels could potentially achieve 100% bandwidth. To simplify bottleneck detection, the edges of the graph corresponding to data transfers are color coded with varying shades of red below 50% bandwidth and shades of green above 50% bandwidth.

3.2 Bottleneck Detection

This section describes analysis techniques to detect communication bottlenecks, which can be used manually or automated to suggest potential optimizations.

Streaming transfer bottlenecks can occur when a streaming buffer is full, in which case input bandwidth is lower than output bandwidth, as shown in Figure 3(a). Although such a bandwidth difference may be counterintuitive due to buffers normally being used to balance bandwidth, as stated in the previous section, the bandwidths only correspond to times that each process transfers data. In this situation, lower input bandwidth occurs because the producer process must block when the buffer is full, which increases the time for transfers and reduces bandwidth. To

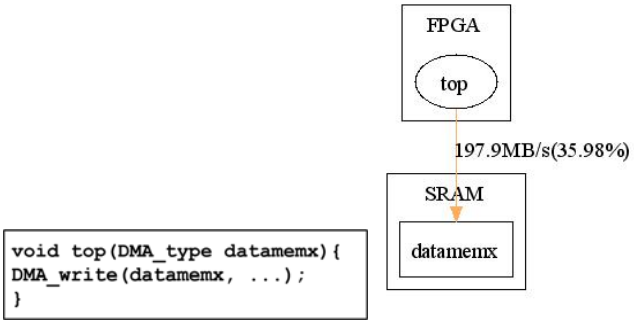

Figure 2: Streaming-communication call visualization

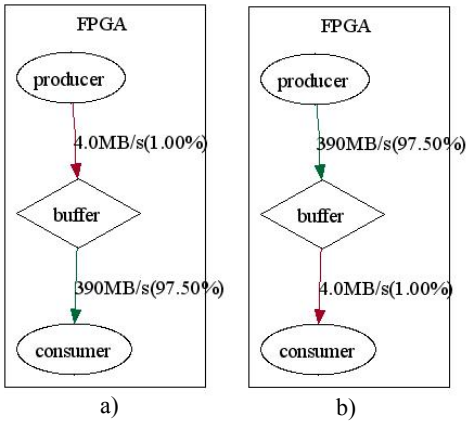

Figure 3: Example of (a) full and (b) empty streaming buffers.

optimize this bottleneck, a designer can increase the buffer size, which allows the producer process to execute for longer before the buffer fills. Alternatively, the consumer process can be optimized via pipelining, different stream widths, etc.

Empty stream buffers are another common bottleneck, which cause the consumer process to block, resulting in low output bandwidth, as shown in Figure 3(b). To optimize this bottleneck, the producer bandwidth should be increased using numerous possibilities including 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. This bottleneck can be caused by multiple problems, such as streaming bursts of data into streaming buffers that become full and empty at different times. Section 4.2 gives an additional example. To reduce this bottleneck, a combination of both optimization methods can be used.

Memory buffers have common bottlenecks due to simultaneous transfers, in which case processes making DMA calls must block, resulting in decreased bandwidth. 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.

In some situations, DMA transfers to external memory can result in bottlenecks appearing on other edges. For example, for a process writing to memory, the physical bandwidth of the memory may become saturated, in which case streaming buffers upstream will eventually become full.

3.3 Instrumentation and Measurement

To measure bandwidth of each communication channel, the tool instruments the high-level code to measure the amount of data transferred and the total transfer time, which includes idle time as a result of blocking. In software, the tool adds a wrapper around each communication call type (e.g., stream read) that measures the time and records the transfer size.

FPGA processes are instrumented by adding monitoring circuits to each communication call. During execution, the monitoring circuits store measured bandwidths locally in registers, which are extracted by the microprocessor after the application has finished. For Impulse-C, the invocation of a communication API call is specified by a specific state in a state machine. Counters are used to monitor cycles spent in a particular call, from which total transfer time can be determined given the clock frequency.

Some communication calls have a static transfer size while others can change dynamically. For a static transfer size, the fixed size of each transfer can be parsed from the source code and the total data can be determined by the measured number of invocations. For dynamic transfer sizes, a counter is used to sum the total bytes transferred each time the process invokes the API call.

To measure the maximum bandwidth of a particular type of communication, the tool runs benchmarks that 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.

4. EXPERIMENTAL RESULTS

Although ideally the proposed techniques would be integrated into a high-level synthesis tool, Impulse-C is proprietary, so we instead added instrumentation to the application code using Perl scripts. A Java GUI front end is used to select files and instrumentation features such as which processes and states should be monitored. The resulting visualization uses Graphviz.

We evaluated the techniques on two different platforms. The first platform was the XtremeData XD1000, which contains a dual-processor motherboard with an Altera Stratix-II EP2S180 FPGA in one of the Opteron sockets. The second platform was one node of the Novo-G supercomputer [5], which uses a GiDEL PROCStar III card with four Stratix-III EP3SE260 FPGAs. 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.

4.1 Triple DES

Triple DES is a block cipher used for encryption. The application consists of a modified version of code provided by Impulse-C. We evaluated this application on the XD1000 platform.

Figure 4 shows the resulting visualization, which solely uses streaming communication. By analyzing the visualization, we identified a bottleneck (shown by the two arrows) corresponding to transfers between the CPU and FPGA. This bottleneck was caused by the low FPGA-to-CPU 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 blocked because the buffer is empty.

To optimize the application, we exploited DMA transfers to increase transfer rates between the CPU and FPGA, which enabled a 2.18x speedup and required about half an hour to complete (not including synthesis times). Logic overhead for the instrumented code was 2%. Clock frequency overhead was 3%.

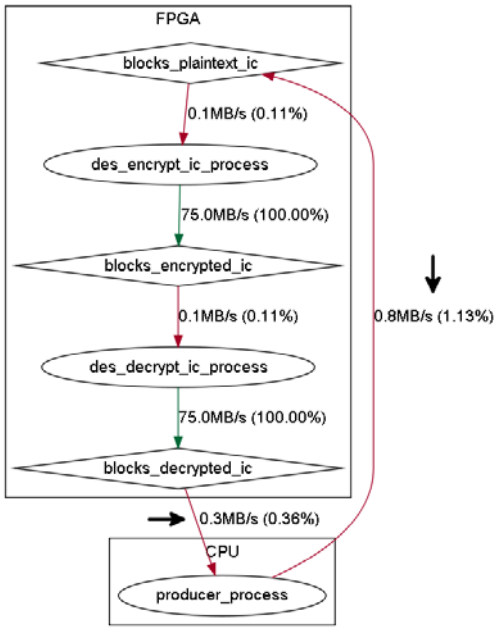

Figure 4: Visualization of streaming Triple DES.

4.2 Molecular Dynamics

Molecular Dynamics (MD) simulates interactions between atoms and molecules over discrete time intervals. For our experiments, the simulation computes forces of 16,384 molecules. Serial C MD code was obtained from Oak Ridge National Lab and optimized to run on the XD1000 FPGA using Impulse-C.

By analyzing the visualization in Figure 5, we identified a bottleneck resulting from all of the streaming buffers with the letter *p* becoming full. Also, the streaming buffers with the letter *a* have low bandwidth both on the inputs and outputs due to 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 buffer size, which only required several minutes of effort and achieved a speedup of 1.25x. The speedup compared to the serial baseline running on the 2.2 GHz Opteron improved from 6.2x to 7.8x.

4.3 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. Although the figure is omitted for brevity, the resulting visualization showed an obvious bottleneck where PCI data transfers are 8-25% of peak speeds due to full streaming buffers.

This bottleneck could potentially be reduced by increasing the buffer size. However, we were unable to perform this optimization due to a lack of full Impulse-C support on Novo-G that limits streaming buffer sizes. Similarly, we could increase

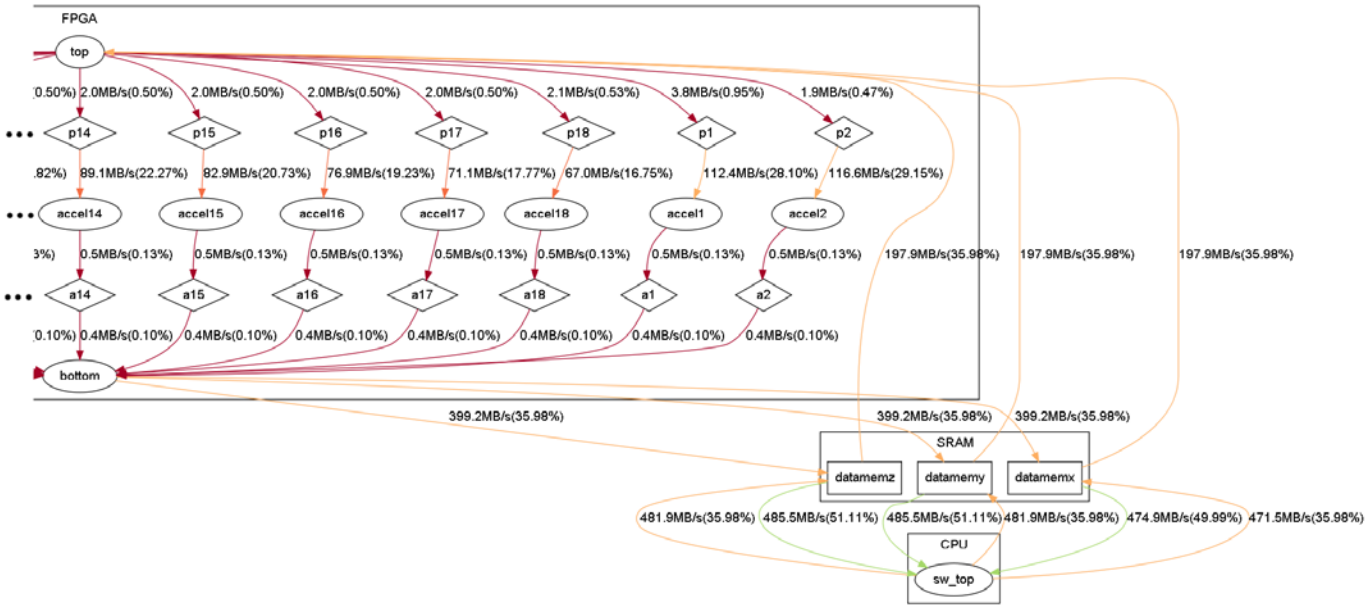

Figure 5: Bandwidth visualization for Molecular Dynamics (half of FPGA cropped to enlarge image).

bandwidths using DMA transfers, but the current Impulse-C support also excludes DMA transfers.

5. CONCLUSIONS

In this paper, we introduced a communication visualization tool for high-level synthesis that allows a developer to quickly locate communication bottlenecks. The application's processes and communication calls are visualized as a directed graph with edges between the CPU, FPGA and buffers. By analyzing the graph for bandwidth distribution and ratios, developers can identify bottlenecks and often quickly apply optimizations. Case studies showed the detection and optimization of bottlenecks, which resulted in FPGA application speedups ranging from 1.25x to 2.18x. In addition, the tool provides source-code correlation, which hides the high-level-synthesis-generated code 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.

6. 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 Altera, Impulse Accelerated Technologies, Nallatech, and Xilinx.

7. REFERENCES

[1] Calvez, J.P. and Pasquier, O. Performance monitoring and assessment of embedded HW/SW systems. In *Proc. International Conference on Computer Design (ICCD)*, 52-57, 1995.

[2] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T.. Legup: high-level synthesis for FPGA-based processor/accelerator systems. In *FPGA'11: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays,* 33–36, 2011.

[3] Curreri, J., Koehler, S., George, A., Holland, B., and Garcia, R. Performance analysis framework for high-level language applications in reconfigurable computing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3, 1, (Jan. 2010), 1-23.

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

[5] George, A., Lam, H., and Stitt, G.. Novo-g: at the forefront of scalable reconfigurable supercomputing. *Computing in Science Engineering*, 13, 1 (Jan-Feb 2011), 82-86.

[6] Heath, M. and Etheridge, J. Visualizing the performance of parallel programs. *Software, IEEE*, 8, 5, (Sep. 1991), 29-39.

[7] Koehler, S., Curreri, J., and George, A.D. Performance analysis challenges and framework for high-performance reconfigurable computing. *Parallel Computing*, 34, 4-5, (2008), 217-230.

[8] Villarreal, J., Park, A., Najjar, W., and Halstead, R.. Designing modular hardware accelerators in c with roccc 2.0. In *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 127-134, 2010.