An End-to-End Tool Flow for FPGA-Accelerated Scientific Computing

Greg Stitt, Alan George, and Herman Lam
University of Florida

Melissa Smith
Clemson University

Vikas Aggarwal, Gongyu Wang, and James Coole
University of Florida

Casey Reardon
MITRE Corp.

Brian Holland
SRC Computers, LLC

Seth Koehler
Altera Corp.

Editor's note:
As part of their ongoing work with the National Science Foundation (NSF) Center for High-Performance Reconfigurable Computing (CHREC), the authors are developing a complete tool chain for FPGA-based acceleration of scientific computing, from early-stage assessment of applications down to rapid routing. This article provides an overview of this tool chain.

—George A. Constantinides (Imperial College London) and Nicola Nicolici (McMaster University)

FPGAs have been widely shown to have significant performance and power advantages compared to microprocessors.[1] Novo-G, for example, is an FPGA-based supercomputer that achieves performance comparable to top supercomputers for computational-biology applications, while only consuming 8 kilowatts of power.[2] Although GPUs often outperform FPGAs for floating-point-intensive applications, FPGAs have better computational density per watt,[2] which is becoming increasingly important as energy and cooling costs start to dominate the total cost of supercomputer ownership.

Despite these advantages, FPGA use has been limited by significantly increased application design complexity as compared to software design, which is mainly due to RTL design challenges. Numerous studies have raised abstraction levels with high-level synthesis (HLS) tools such as AutoESL's AutoPilot, Synfora's PICO the open-source ROCCC (Riverside Optimizing Compiler for Configurable Computing tool, Impulse Accelerated Technologies' Impulse-C, and Mentor Graphics' Catapult C, in addition to graphical-design environments (e.g., MathWorks' Simulink and National Instruments' LabVIEW).

Although HLS tools have improved productivity, FPGA use is still largely limited to hardware experts. Recent studies have identified other key productivity bottlenecks, which include prohibitively long design iterations, limited portability, interoperability, and design reuse, in addition to limited debug and performance analysis.[3]

In this article, we describe a study in which we extended and combined existing FPGA tools to create a tool flow that addresses these bottlenecks. Although previous studies have addressed bottlenecks individually, in many cases the tasks performed by these tools are not interoperable. Therefore, a primary contribution of our study is the focus on end-to-end productivity improvement. In addition, to avoid significant tool-flow disruptions that have previously limited tool acceptance, the tool flow complements existing FPGA tools, in many cases letting designers use their existing languages and synthesis tools. The key contributions of the tool flow include formulation techniques for rapid design-space exploration, a coordination framework for communication and synchronization between tasks in different languages and

0740-7475/11/$26.00 © 2011 IEEE Copublished by the IEEE CS and the IEEE CASS

devices, intermediate fabrics for fast placement and routing (PAR), and tools for performance analysis and bottleneck detection.

Tool flow overview

Figure 1 illustrates the end-to-end FPGA tool flow, which complements existing languages and synthesis tools with four main extensions that address key productivity bottlenecks: formulation tools, a coordination framework, intermediate fabrics, and performance analysis tools.

To reduce design iterations, the tool flow enables early design-space exploration, which we refer to as *formulation*. FPGA application designers often spend weeks creating an implementation, only to find that fundamental design decisions (e.g., parallelism, communication, or mappings) cause inefficient execution. The designer then iteratively modifies the design until achieving acceptable efficiency, which can take weeks. Formulation avoids these lengthy design iterations by combining abstract modeling and performance prediction to enable rapid design-space exploration of different parallelization strategies, architectures, and mappings of tasks onto architectural components, all before the designer has written any code. The output of formulation is a task-graph model of an application and a mapping of tasks onto devices in which behavior of each task has not been specified in code.

After formulation, the designer specifies behavior of each task according to the device mapping. For the example that Figure 1 shows, a designer may implement software-mapped tasks using C or C++, while implementing FPGA tasks using an HDL language (e.g., VHDL) or a high-level language (e.g., Impulse-C).

One limitation of current FPGA tools is a lack of interoperability between tasks defined in different languages for different devices. Designers currently handle communication between such tasks manually using mechanisms specific to a particular board, device, or language, which reduces code reuse and portability. To address this problem, the tool flow uses a *coordination framework* to transparently communicate and synchronize (i.e., coordinate) between tasks.

The tool flow also addresses increasingly long PAR times, which can add hours or days to a design iteration. Lengthy PAR also prohibits mainstream design and debug methodologies based on rapid

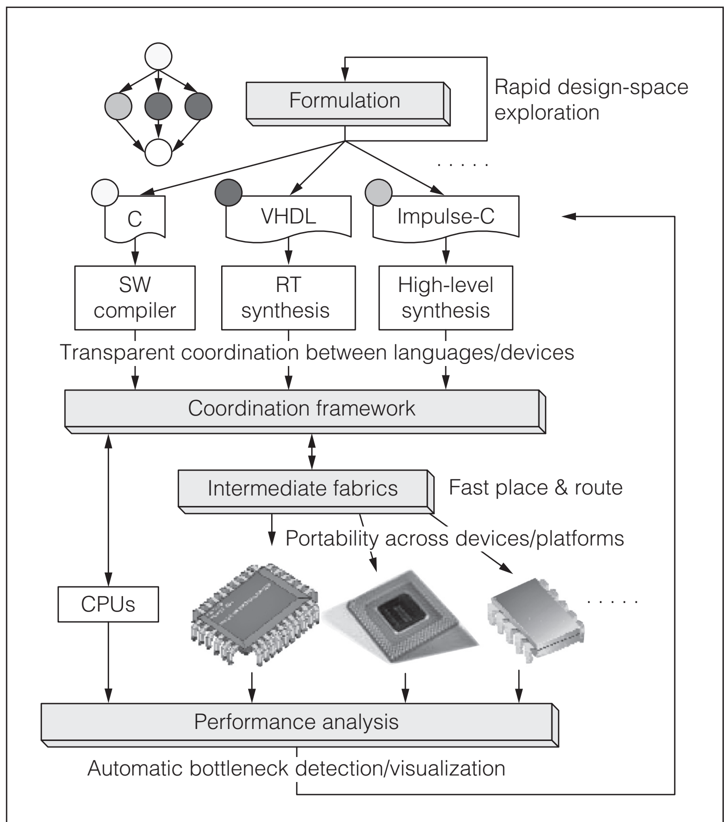

Figure 1. Overview of end-to-end FPGA tool flow.

compilation. To achieve faster PAR, the tool flow uses virtual PAR-specialized FPGAs, referred to as *intermediate fabrics*, implemented on commercial off-the-shelf (COTS) devices. In addition to achieving PAR speedup, intermediate fabrics also enable application portability across different devices and platforms, which improves design reuse.

An additional contribution of the tool flow is the ReCAP (Reconfigurable Computing Application Performance) *performance analysis* tool, which helps identify and eliminate performance bottlenecks. Although such tools are common in software design, to our knowledge ReCAP is the first performance analysis tool for FPGA applications.

Although the tool flow integrates all extensions, each step is optional. For example, designers unconcerned with PAR times could skip intermediate fabrics.

To illustrate tool use in the tool flow we created, we use a time-domain finite-impulse-response (TDFIR) benchmark from the HPEC Challenge, which defines a set of common kernels for signal and image processing,[4] implemented on a single node of Novo-G, which consists of a quad-core

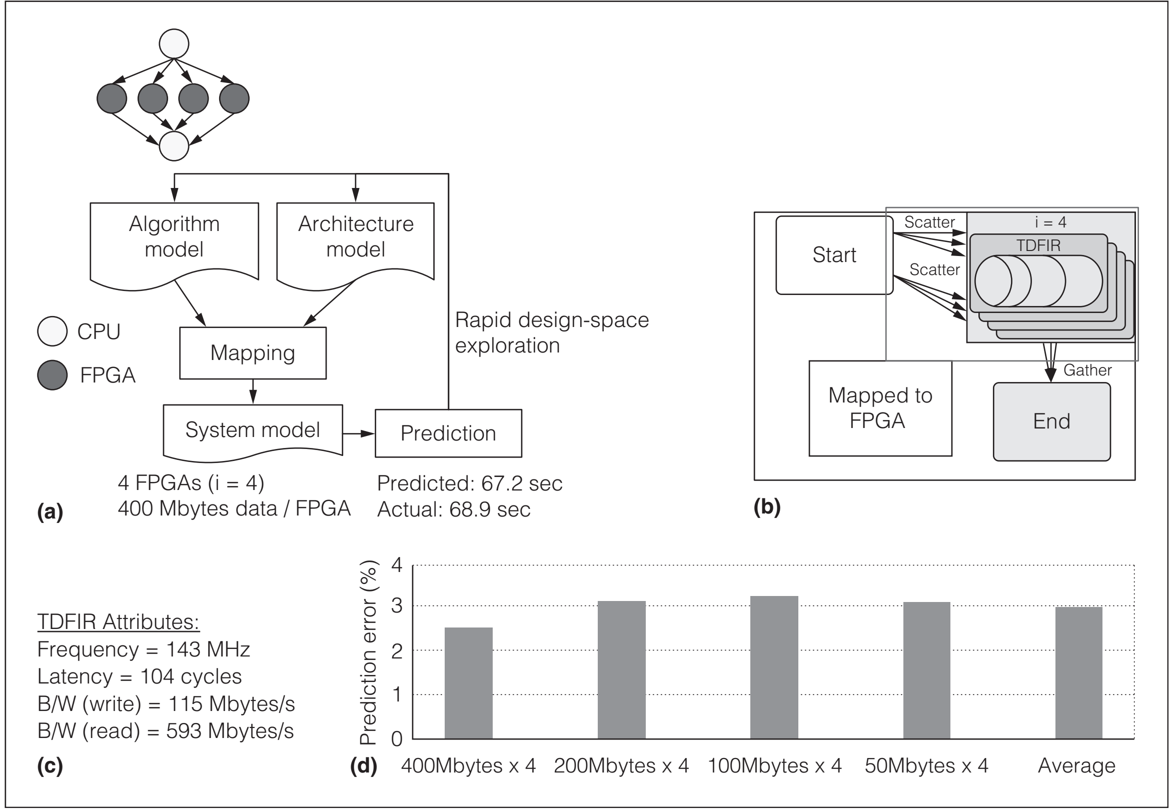

Figure 2. A formulation approach for rapid design-space exploration (a) in which designers use Reconfigurable Computing Modeling Language (RCML) models to describe parallelism, such as time-domain finite-impulse response (TDFIR) (b), which when combined with user- and tool-estimated attributes (c), enable accurate performance prediction (d). (B/W: bandwidth.)

Intel Xeon and four Altera Stratix III E260 FPGAs on a PCIe GiDEL PROCStar-III board. Although this example is intentionally small for demonstration, we have evaluated the tools for more complex applications.[5-7]

Formulation

Figure 2a describes our formulation approach, which lets designers explore different algorithms (i.e., parallelization and communication strategies), architectures, and mappings of algorithmic tasks to architecture components without having to write any code. Formulation consists of two main steps: RCML modeling and prediction.

RCML modeling

To perform formulation, a designer initially models a potential algorithm and architecture in the Reconfigurable Computing Modeling Language.[7] RCML differs from most modeling environments by using an abstraction level that we refer to as an estimation model. Existing modeling environments, such as Ptolemy and Simulink, typically use an abstract executable model, in which users define a functional implementation before estimating performance. Similarly, programming languages and libraries such as UPC, OpenMP, and Intel's Cilk++ let designers explore parallelism strategies, but they also require functional code. By contrast, RCML lets users quickly explore different parallelization, mapping, and work distribution alternatives by replacing a precise definition of tasks (i.e., code) with abstract attributes.

Figure 2b illustrates an example RCML algorithm model for the evaluated TDFIR application, in which initially a start task performs preprocessing and then scatters kernels and signals to four pipelined TDFIR tasks, whose results are gathered by the end task. Although existing modeling languages could also be used for formulation, the RCML algorithm model simplifies FPGA application modeling by providing communication and computation constructs common to FPGA applications. The algorithm model is

represented as a synchronous, data-independent dataflow graph, in which each node could be an arbitrary task (e.g., fast Fourier transform or convolve), or an FPGA-specialized construct (e.g., pipeline). Edges in the graph either can be direct communication between tasks or can represent more-complex communication (e.g., scatter, gather, or broadcast). To create an algorithm model, a designer requires only a basic understanding of the coarse-grained parallelism (like that given in Figure 2b) of an application. If more-detailed analysis is required, designers can create a finer-grained model.

The RCML architecture model (not shown in Figure 2) allows a designer or platform vendor to describe a system architecture by combining different devices and interconnects.

After modeling the algorithm and architecture, the designer can explore different mappings to create the RCML system model. For example, designers can map tasks expected to be computationally intensive onto FPGAs, while mapping communication between tasks onto an available interconnect (e.g., PCIe or InfiniBand). For the TDFIR example, we map all TDFIR tasks to four separate FPGAs, and the start and end tasks to a microprocessor.

Prediction tools

The RCML system model also includes designer- or tool-estimated attributes for each task and architectural component, which enables prediction tools to estimate performance. For the TDFIR example (Figure 2c), these attributes included pipeline latency and clock frequency, in addition to I/O bandwidth determined by benchmarking the PCIe interconnect bandwidth for different transfer sizes. Although any prediction tool could potentially be used, RCML currently uses the RC amenability test (RAT) for performance prediction,[5] and the core-level modeling and design (CMD) tool for estimation of FPGA frequency, area, and latency.[8]

RAT is an analytical performance prediction model that uses common characteristics of pipelined FPGA implementations to predict performance. RAT predicts communication times using transfer sizes combined with interconnect throughput and latency, both of which can be easily specified as attributes in the RCML model. RAT predicts computation time of FPGA tasks using attributes that specify the input size, frequency, and I/O bandwidth of the corresponding task. Although these attributes are clearly not sufficient for all applications, RAT is intended for pipelined (i.e., data-independent data flow) FPGA circuits, which are commonly used in scientific computing.

Although designers can manually specify attributes, the tool flow also supports automated attribute estimation. To determine attributes of FPGA tasks with the CMD tool, designers can predict frequency, area, and latency because the CMD tool lets designers model FPGA circuits as an interconnection of coarse-grained cores—e.g., floating-point operators, fast Fourier transform (FFT), or FIR—that have predetermined areas, latencies, and clock frequencies. CMD also lets designers analyze characteristics of the interconnection to make predictions for the entire circuit. The details are outside the scope of this article, but the basic approach combines the critical-path delay of individual cores with an analysis based on Rent's rule to estimate effects of routing congestion.[8] In some cases, where accuracy is a lesser concern, designers can simply estimate appropriate attribute values. For example, a designer might want to assume that 125 MHz is attainable, and then estimate that a certain number of FPGA-mapped tasks will fit on a device. Although predictions based on such assumptions will be less accurate, these predictions can still help identify fundamental bottlenecks that might require a completely different parallelization strategy.

Figure 2d compares RAT performance predictions with actual performance for the TDFIR application when using different signal sizes ranging from 400 Mbytes down to 50 Mbytes. On average, prediction error in this case was only 3%. Similar prediction errors were shown in previous application studies,[7] which used more complex models and mappings. Furthermore, we created these models and performed this analysis in minutes. If we had not been satisfied with this predicted performance, we could have revised the algorithm, architecture, or mapping to eliminate bottlenecks. For example, an FPGA implementation supporting arbitrary kernel sizes requires postprocessing, which could benefit from exploration to determine how to partition the postprocessing across the FPGAs and/or microprocessors. Assuming a designer is already familiar with an application or algorithm, we would expect similar times (i.e., minutes) for other scenarios. It is difficult to accurately estimate productivity improvement. However, by assuming that formulation will typically

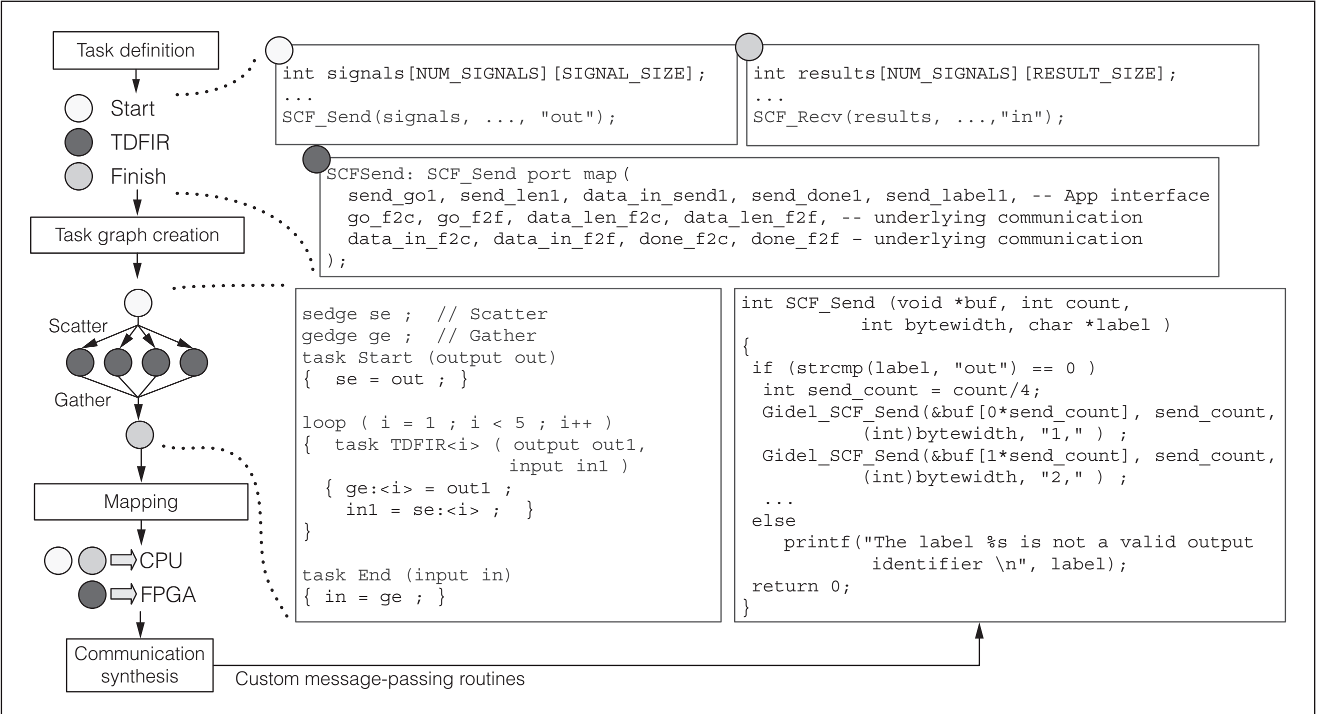

Figure 3. System coordination framework (SCF) overview. SCF allows designers to transparently combine tasks from different languages (e.g., C and VHDL) and devices (e.g., CPU and FPGA) into an application (e.g., TDFIR) by automatically generating synchronization and communication routines based on intertask communication and platform interconnect.

save at least one design iteration, it is reasonable to say that the presented techniques can reduce total design time by days or weeks.

Coordination framework

Figure 3 illustrates an overview of the *system coordination framework* (SCF),[9] which enables application tasks defined in multiple languages for different devices to transparently coordinate. SCF is conceptually similar to approaches that ease specification of FPGA multiprocessor systems such as Xilinx EDK (Embedded Development Kit; http://www.xilinx.com/support/documentation/dt_edk_edk13-1.htm), Altera SOPC Builder (System-On-A-Programmable-Chip; http://www.altera.com/support/software/system/sopc/sof-sopc_builder.html), and RAMP (Research Accelerator for Multiple Processors; http://bwrc.eecs.berkeley.edu/Research/RAMP).[10] SCF complements these approaches by providing a common interface that is vendor, device, and language neutral.

To use SCF, the designer initially defines individual tasks (shown as "Task definition") using any language, compiler, or synthesis tool. On the basis of the RCML model for the TDFIR example, we define tasks `Start` and `End` in C++ (which we compile with g++), and `TDFIR` in VHDL (which we synthesize using Altera Quartus II). SCF also supports other languages, such as Impulse-C, but we did not evaluate those languages for TDFIR.

To integrate tasks with SCF, the designer specifies task inputs and outputs in each language using message-passing primitives (e.g., `SCF_Send` and `SCF_Receive`) that hide low-level communication details. In fact, the task-definition code does not need to specify the source of a receive or the destination of a send. As Figure 3 shows, the C++ code simply sends to an output called "out" and receives from an input call "in." The VHDL code uses send and receive entities with a simple communication protocol to interface with the TDFIR circuit. Without SCF, defining a task I/O depends on the source and destination device, often requiring the use of different combinations of APIs for different mappings. Because the task I/O in SCF is independent of other tasks, devices, and mappings, SCF enables task portability across multiple devices and code reuse. Furthermore, SCF can potentially convert between data formats used

by different devices, which further improves productivity.

After defining tasks, the designer creates the complete application by using a specialized language to connect the tasks into a task graph (shown as "Task graph creation" in Figure 3) that defines all communication. As Figure 3 shows, this language defines constructs for connecting task I/O interfaces to specialized edges that define communication patterns (e.g., `sedge` for scatter, and `gedge` for gather). SCF currently supports scatter, gather, broadcast, and direct communication. Next, the designer performs mapping, which assigns tasks to devices in the system architecture as specified in the RCML system model (not shown). SCF then uses the mapping to perform communication synthesis, which implements all edges of the task graph using the system's specific communication capabilities. For the TDFIR example, SCF implements all CPU to FPGA communication using API calls for the GiDEL board. If this example required FPGA-to-FPGA communication, SCF would implement such communication using physical wires on the GiDEL board.

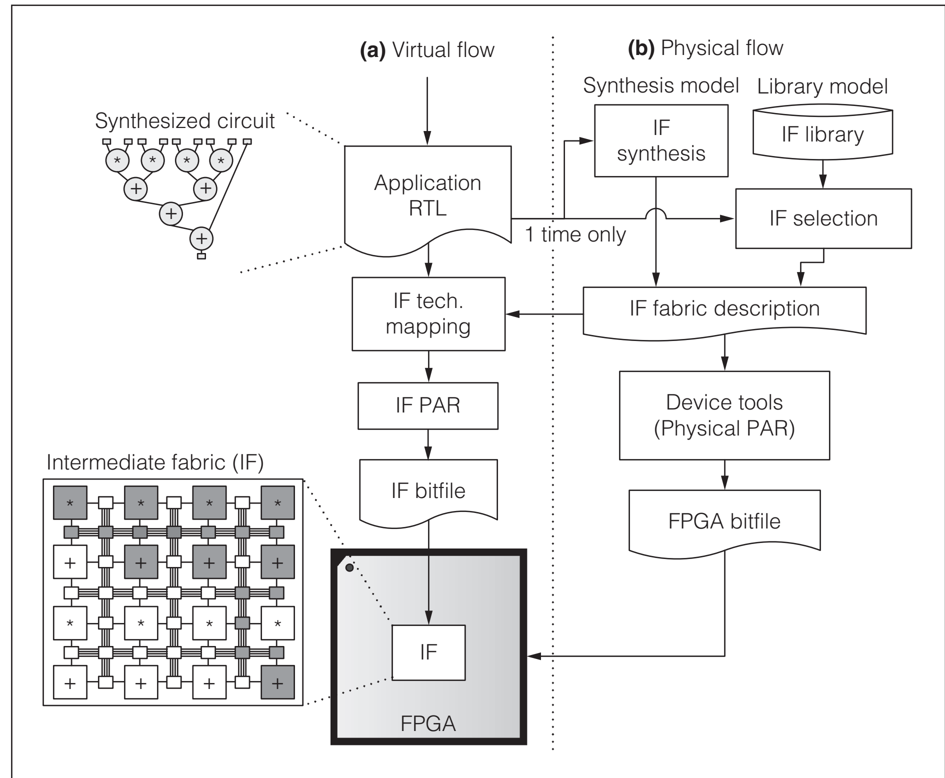

Figure 4. Intermediate fabrics implement application-specialized virtual RC architectures on FPGAs to achieve fast placement and routing (PAR).

For TDFIR on Novo-G, SCF area and performance overhead in our tool flow implementation was less than 1%. For other image-processing and scientific-computing applications that we tested, average area overhead compared to device-specific APIs was 2%, and performance overhead was 1%.[9]

Intermediate fabrics

Intermediate fabrics (IFs),[11] as Figure 4 shows, are virtual reconfigurable devices that act as an intermediate translation layer between netlists and physical FPGAs. IFs are conceptually similar to virtual overlay networks[12] but also provide fast PAR. From the application designer's point of view, the virtual IF tool flow (see Figure 4a) is identical to other reconfigurable devices. However, unlike a physical device, whose architecture must support many applications, IFs can be specialized for particular domains or even individual applications. Such specialization hides the complexity of fine-grained COTS devices, thus enabling fast PAR. In addition, IFs enable portability across any physical device that can implement the fabric. Other advantages include partial reconfiguration on devices lacking architectural support and abstraction of multiple devices (e.g., one IF spanning multiple FPGAs).

IF architectures can potentially implement any fabric, but the tools currently support island-style fabrics with application-specialized computational units (CUs) spread across reconfigurable interconnects. Because IFs are specialized for different domains, CUs can range from bit-level operations to coarser-grained operations such as fixed- or floating-point arithmetic logic units, fast Fourier transforms, filters, and so on. Other specialization options include varying fabric size, connection box or switch box flexibility and topology, percentages of long tracks, and tracks per channel, among others.[11] The intent of these specializations is to sacrifice a small amount of general routability to reduce area overhead for the target application.

The tools implement IFs on a physical device using two possible use models, as Figure 4b shows. The synthesis model creates a custom IF for the target application (e.g., TDFIR), which requires a single physical PAR, but will often have the least overhead due to the highest specialization. Alternatively, the library model uses IF selection to search for an appropriate pre-implemented fabric. The tools currently have partial support for both models, although the library model is less mature, owing to open challenges involving trade-offs between library size and support for different domains. The synthesis and selection steps currently require manual assistance, where the designer either specifies the resources needed or selects an existing fabric. After selecting an IF, the tools output corresponding RTL VHDL, which can be synthesized to an FPGA.

The main limitations of IFs are area and clock frequency overhead incurred by the RTL virtual fabric. Fortunately, the ability to specialize IFs for particular applications and domains can greatly reduce this overhead. Although we were unable to evaluate the TDFIR application due to a lack of support for IFs on Novo-G (which will be completed soon), previous results showed that IFs used for floating-point circuits achieved a PAR speedup of 1,112×,[11] with an area overhead of 14% and a clock overhead of 19% on a Xilinx Virtex 4 LX200. IFs used for 16-bit, fixed-point circuits achieved a PAR speedup of 275×, with an area overhead of 9% and a clock overhead of 18%. In both cases, retained routability was more than 90% after specialization.

Performance analysis

Figure 5a illustrates the ReCAP performance analysis tool,[6] which instruments application code to measure time spent in different regions. The software profiling of ReCAP is similar to performance-counter-based profilers such as Intel's VTune, but ReCAP extends these capabilities to identify FPGA and communication bottlenecks. For RTL code, ReCAP adds counters to all clocked process blocks to track frequencies of different states and paths. Next, the designer executes the instrumented application to collect performance measurements. To deal with limited resources inside the FPGA, ReCAP periodically stalls execution, transfers measurements to the microprocessor, and then continues. ReCAP will often avoid stalls by doing postmortem monitoring when counters and memories are guaranteed not to overflow. Upon completion, ReCAP collects measurements and creates visualizations to identify bottlenecks, which the designer then optimizes.

ReCAP also performs optional bottleneck detection, where designers add annotations via pragmas to specify the purpose of each region. For example, in a state machine, states could be used for initializing the circuit, waiting for data, performing work, and so on. ReCAP combines the pragmas with the collected measurements to automatically detect bottlenecks, and then reports optimization suggestions according to the corresponding ideal speedup.

ReCAP assists with debug by using monitors and block RAM to create in-circuit waveforms, like Altera's SignalTap and Xilinx's ChipScope tools do, but without requiring a JTAG connection. The fact that a JTAG connection is not needed eases the tool's use for designers targeting PCIe accelerator boards. To use these techniques, the designer selects the signals to monitor and optionally specifies when monitoring should occur using a VHDL condition statement consisting of any of the internal signals. ReCAP also generates testbenches based on in-circuit behavior that identify differences between simulation and actual execution, while also supporting in-circuit assertions.

A potential limitation of ReCAP is overhead introduced by the measurements, which could potentially perturb application behavior, leading to misidentified bottlenecks or exposure of application bugs that were previously hidden. However, for existing case studies, ReCAP overhead was negligible, requiring only a few counters and minimal memory. In situations where a designer would want to monitor numerous resources, overhead could potentially be higher, but in our experience such monitoring has not been necessary to identify bottlenecks.

Figure 5b illustrates TDFIR when using between one and four FPGAs for a data set of 128 filters, with signal sizes of 32,768 samples and kernel sizes of 4,096 samples. As shown, the original FPGA implementation had poor scalability. To detect this bottleneck, we used ReCAP and discovered that the FPGAs were idle most of the time, despite performing significant computation. ReCAP identified several potential causes of the bottleneck, which included inefficient communication and "late sender" bottlenecks between the FPGAs and CPU. We optimized the inefficient communication by using lower-latency API calls and by adjusting transfer sizes to maximize bandwidth. We optimized the "late sender" bottlenecks

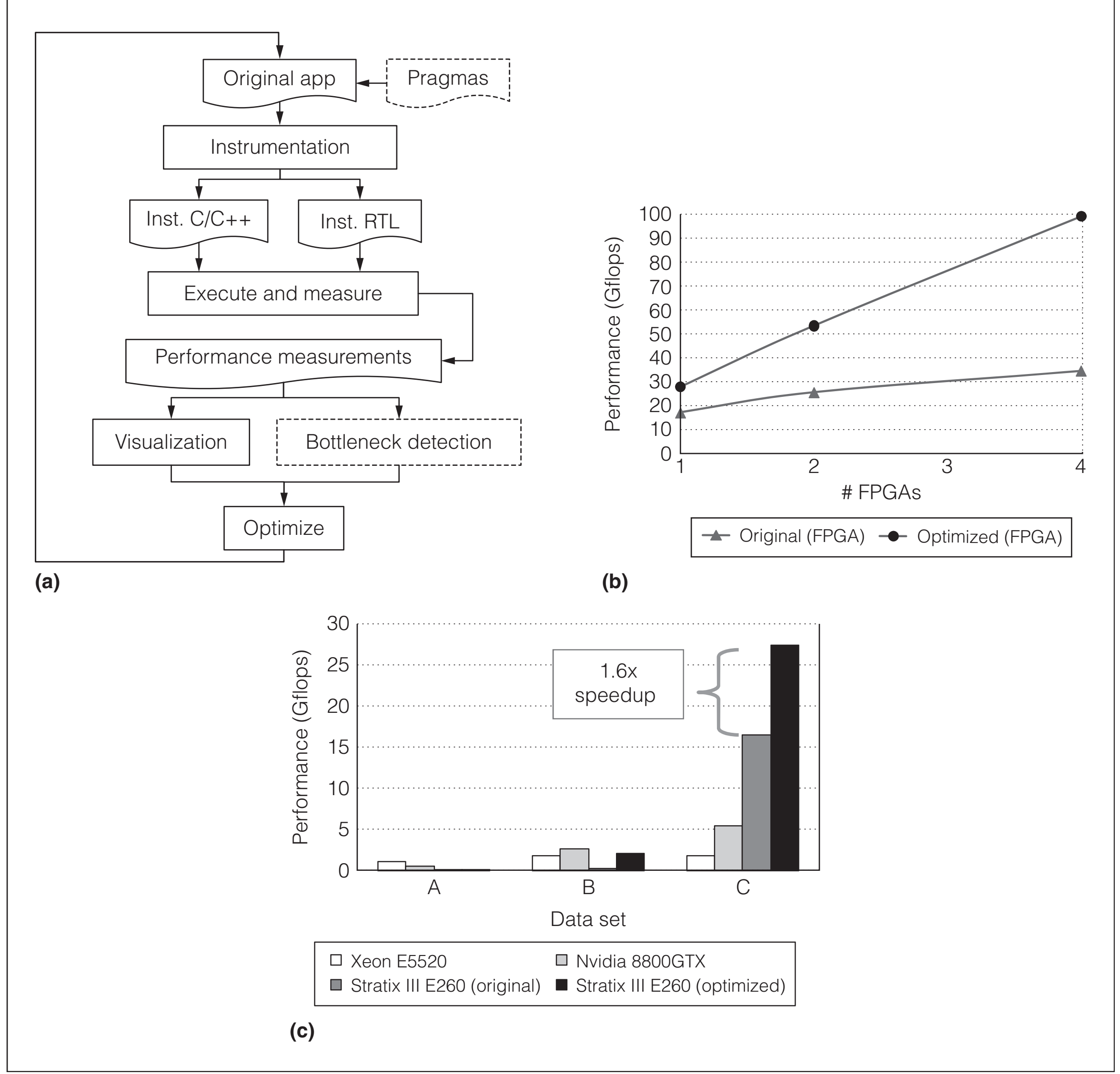

Figure 5. ReCAP (Reconfigurable Computing Application Performance) performance analysis overview (a). Results for TDFIR showing poor scalability for original FPGA version (b). Optimizing ReCAP-identified bottlenecks provided speedup of 2.9× for four FPGAs and a single-FPGA speedup of 1.6×, making the FPGA 5× faster than an Nvidia 8880GTX GPU (c). (Figure to appear in S. Koehler, G. Stitt, and A. George, "Platform-Aware Bottleneck Detection for Reconfigurable Computing Applications," *ACM Transactions on Reconfigurable Technology and Systems*, September 2011; used with permission.)

by overlapping computation and communication via buffering.

These optimizations led to a 1.6× speedup for a single FPGA and a 2.9× speedup for four FPGAs, which corresponded to a speedup of 54× over the software baseline. Compared to a previous GPU study (see Figure 5c), the ReCAP-optimized, single-FPGA implementation was 5× faster for this data set (shown as data set C). However, faster GPUs and larger FPGAs are now available. For the smaller data set A (20 filters, signal size of 1,024, kernel size of 12), the Xeon was faster than both the GPU and FPGA. Data set B uses an input size in between A and C with 64 filters, a signal size of 4,096, and a kernel size of 128. For set B, the GPU and optimized FPGA implementation were both slightly faster than the Xeon. The overhead introduced by ReCAP was a 5.1% increase in software execution time, a 1.8% increase in LUTs, a 1% increase in registers, and a less than 1% decrease in frequency.

As future work, we plan to evaluate the complete tool flow on emerging scientific-computing applications. We plan to extend the formulation studies to include Turing-complete RCML models that enable nonfunctional simulations for automatic bottleneck detection. For intermediate fabrics, future work will focus on reducing area overhead, and using intermediate fabrics to enable runtime compilation of OpenCL applications for FPGAs. ■

Acknowledgments

This work was supported in part by the I/UCRC (Industry/University Cooperative Research Centers) Program of the National Science Foundation under grant EEC-0642422, and DARPA c/o AFRL under contract FA8650-07-1-7742. We gratefully acknowledge the vendor equipment and/or tools provided by Aldec, Altera, GiDEL, Nallatech, and Xilinx.

References

1. A. DeHon, "The Density Advantage of Configurable Computing," *Computer,* vol. 33, no. 4, 2000, pp. 41-49.
2. A. George, H. Lam, and G. Stitt, "Novo-G: At the Forefront of Scalable Reconfigurable Supercomputing," *IEEE Computing in Science and Engineering,* vol. 13, no. 1, 2011, pp. 82-86.
3. B. Nelson et al., "Design Productivity for Configurable Computing," *Proc. Int'l Conf. Eng. of Reconfigurable Systems and Algorithms* (ERSA 08), CSREA Press, 2008, pp. 57-66.
4. R. Haney et al., "The HPEC Challenge Benchmark Suite," High-Performance Embedded Computing Workshop, 2005; http://www.ll.mit.edu/HPECchallenge/docs/hpecChallengePresentation.Haney.2005.pdf.
5. B. Holland, K. Nagarajan, and A. George, "RAT: RC Amenability Test for Rapid Performance Prediction," *ACM Trans. Reconfigurable Technology and Systems,* vol. 1, no. 4, 2009, pp. 1-31.
6. S. Koehler, G. Stitt, and A. George, "Platform-Aware Bottleneck Detection for Reconfigurable Computing Applications," *ACM Trans. Reconfigurable Technology and Systems,* to be published in Sept. 2011.
7. C. Reardon et al., "RCML: An Environment for Estimation Modeling of Reconfigurable Computing Systems," *ACM Trans. Embedded Computing Systems,* to be published.
8. G. Wang et al., "A Framework for Core-Level Modeling and Design of Reconfigurable Computing Algorithms," *Proc. 3rd Int'l Workshop High-Performance Reconfigurable Computing Technology and Applications* (HPRCTA 09), ACM Press, 2009, pp. 29-38.
9. V. Aggarwal et al., "SCF: A Device- and Language-Independent Task Coordination Framework for Reconfigurable, Heterogeneous Systems," *Proc. 3rd Int'l Workshop High-Performance Reconfigurable Computing Technology and Applications* (HPRCTA 09), ACM Press, 2009, pp. 19-28.
10. G. Gibeling et al., *The RAMP Architecture, Language and Compiler,* tech. report, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, 2007; http://ramp.eecs.berkeley.edu.
11. J. Coole and G. Stitt, "Intermediate Fabrics: Virtual Architectures for Circuit Portability and Fast Placement and Routing," *Proc. 5th IEEE/ACM Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES/ISSS 10), ACM Press, 2010, pp. 13-22.
12. N. Kapre et al., "Packet-Switched vs. Time-Multiplexed FPGA Overlay Networks," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 06), IEEE CS Press, 2006, pp. 205-216.

Greg Stitt is an assistant professor of electrical and computer engineering at the University of Florida, where he is also a faculty member of the National Science Foundation (NSF) Center for High-Performance Reconfigurable Computing (CHREC). His research interests include design automation for reconfigurable computing, high-performance computing, and embedded systems. He has a PhD in computer science from the University of California, Riverside, and is a member of IEEE and the ACM.

Alan George is a professor of electrical and computer engineering at the University of Florida, where he also directs the NSF CHREC. His research interests include high-performance architectures, networks, systems, services, and applications for reconfigurable, parallel, distributed, and fault-tolerant computing. He has a PhD in computer science from Florida State University, and is a member of IEEE and the ACM.

Herman Lam is an associate professor of electrical and computer engineering at the University of Florida, where he is also a senior research faculty member at the NSF CHREC. His research interests include productivity methods and tools for RC application development, particularly as applied to large-scale applications for reconfigurable supercomputing. He has a

PhD in electrical engineering from the University of Florida, and is a member of IEEE and the ACM.

Melissa Smith is an assistant professor of electrical and computer engineering at Clemson University. Her research interests include reconfigurable and high-performance computing. She has a PhD in electrical engineering from the University of Tennessee, Knoxville, and is a member of ACM and senior member of IEEE.

Vikas Aggarwal is pursuing a PhD in the Department of Electrical and Computer Engineering at the University of Florida, where he is also affiliated with the NSF CHREC. His research interests include tools and applications for reconfigurable computing and high-performance computing. He has an MS in electrical and computer engineering from the University of Florida.

Gongyu Wang is pursuing a PhD in the Department of Electrical and Computer Engineering at the University of Florida, where he is also affiliated with the NSF CHREC. His research interests include formulation and design-space exploration for reconfigurable computing. He has an MS in electrical and computer engineering from the University of Florida.

James Coole is pursuing a PhD in the Department of Electrical and Computer Engineering at the University of Florida, where he is also affiliated with the NSF CHREC. His research interests include architectures and tools for reconfigurable computing. He has a BS in computer engineering from the University of Florida.

Casey Reardon is a senior networking and distributed systems engineer at MITRE Corp. His research interests include hardware-software partitioning and design-space exploration for reconfigurable computing. He has a PhD in electrical and computer engineering from the University of Florida.

Brian Holland is a senior application engineer at SRC Computers. His research interests include modeling and amenability analysis for reconfigurable computing. He has a PhD in electrical and computer engineering from the University of Florida.

Seth Koehler is a senior software engineer at Altera Corp. His research interests include performance analysis and automatic bottleneck detection for reconfigurable computing. He has a PhD in electrical and computer engineering from the University of Florida.

■ Direct questions and comments about this article to Greg Stitt, Dept. of Electrical and Computer Engineering, University of Florida, PO Box 116200, Gainesville, FL 32611-6200; gstitt@ece.ufl.edu.

cn **Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.**