Department Editors: Kevin Rudd and Kevin Skadron

Are Field-Programmable Gate Arrays Ready for the Mainstream?

GREG STITT
University of Florida

For more than a decade, researchers have shown that field-programmable gate arrays (FPGAs) can accelerate a wide variety of software, in some cases by several orders of magnitude compared to state-of-the-art microprocessors.[1-3] Despite this performance advantage, FPGA accelerators have not yet been accepted by mainstream application designers and instead remain a niche technology used mainly in embedded systems.

By contrast, GPUs have quickly gained wide popularity. Although GPUs often provide better performance, especially for floating-point-intensive applications, FPGAs commonly have significant advantages for bit-level and integer operations.[4,5] In addition, FPGAs often have better energy efficiency,[6] in some cases by as much as 10 times.[5] A motivating example is the FPGA-based Novo-G system,[7] which provides performance for computational biology applications that's comparable to the Roadrunner and Jaguar supercomputers, while only consuming 8 kW compared to several megawatts.

With the importance of energy efficiency growing rapidly, these surprising results raise the following question: if FPGAs offer such significant advantages over alternative devices, why are they still a niche technology? This column will attempt to answer this question by discussing the major barriers that have prevented more widespread FPGA usage, in addition to surveying research and necessary innovations that could potentially eliminate these barriers.

FPGA architectures

To understand the barriers to mainstream usage, we must first understand how FPGA architectures differ from multicore and GPU architectures. The most fundamental difference is that multicores and GPUs provide general-purpose or specialized processors to execute parallel threads, whereas FPGA architectures implement digital circuits by providing numerous lookup tables implemented in small RAMs. Lookup tables implement combinational logic by storing the corresponding truth table and using the logic inputs as the address into the lookup table. Figure 1 shows a simple example of a 32-bit adder decomposed into 32 full adder circuits, which synthesis tools might map onto 32 lookup tables. Similarly, FPGAs enable sequential circuits by providing flip-flops along lookup table outputs. By providing hundreds of thousands of lookup tables and flip-flops, FPGAs can implement massively parallel circuits.

Modern FPGAs also commonly have coarser-grained resources such as configurable logic blocks (CLBs), multipliers, digital signal processors, and on-chip RAMs. Some embedded-systems-specialized FPGAs, such as Xilinx's Virtex-5Q FXT (http://www.xilinx.com/products/silicon-devices/fpga/virtex-5q/fxt.htm), even contain microprocessor cores. In addition, for FPGAs without these microprocessors, designers can use the lookup tables to implement numerous soft processors—such as Xilinx's MicroBlaze Soft Processor (http://www.xilinx.com/tools/microblaze.htm) and Altera's Nios II Processor (http://www.altera.com/devices/processor/nios2/ni2-index.html)—in some cases even more than 1,000.[8] Although such processors make an FPGA look similar to a GPU or multicore, soft processors are an order of magnitude slower than most dedicated processors, which has resulted in niche usage (for example, embedded systems and multicore emulation[8]). For mainstream computing, soft processors would likely be used as controllers for custom pipelined circuits in other parts of the FPGA.

To combine these resources into a larger circuit, FPGAs provide reconfigurable interconnect similar to Figure 2. In between each row and column of resources (such as lookup tables and CLBs), FPGAs contain numerous routing tracks, which are wires that carry signals

Published by the IEEE Computer Society

0272-1732/11/$26.00 © 2011 IEEE

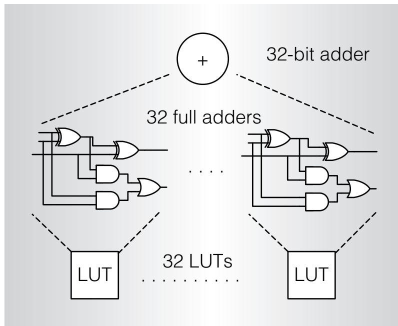


Figure 1. Field-programmable gate arrays (FPGAs) perform computation by implementing the corresponding logic (for example, a 32-bit adder) using lookup tables (LUTs). For this example, synthesis tools divide the 32-bit adder into smaller circuits (for example, 32 full adders) and then map those circuits onto lookup tables with the same or larger numbers of inputs and outputs.

across the chip. Connection boxes provide programmable connections between resource I/O and routing tracks. Similarly, switch boxes provide programmable connections between routing tracks where each row and column intersect, which allows for tools to route a signal to any destination on the device. Such an architecture is commonly called an *island-style fabric*, where computational resources are analogous to islands in a sea of interconnect.

To implement an application on this architecture, FPGA tools typically take a register-transfer-level (RTL) circuit and perform technology mapping to convert the circuit into device resources (for example, converting an adder to lookup tables in Figure 1), followed by placement to map each technology-mapped component onto physical resources, and finally routing to program the interconnect to implement all connections in the circuit. When using processors, either soft or hard, device vendors provide tool frameworks such as Xilinx's EDK (Embedded Development Kit) and Altera's SOPC (System on a Programmable Chip) Builder that provide software compilers in addition to codesign tools for integrating the processors with custom circuits in other parts of the FPGA. For FPGAs in PCI Express accelerator boards, board vendors provide a communication API for transferring data to and from software code running on a host processor.

Design methodologies and programming models

To use an FPGA for accelerating software, designers typically start by profiling software, identifying the most computationally intensive regions, and then implementing those regions as circuits in the FPGA.

To implement a region of code on the FPGA, there are several possible programming models. The most common model is to manually convert the code into a semantically equivalent RTL circuit, which designers typically specify using hardware-description languages (HDLs) such as VHSIC Hardware Description Language (VHDL) or Verilog. Figure 3a shows an example of a high-level loop and a corresponding pipelined data path in Figure 3b that fully unrolls the inner loop, resulting in 64 multiplications and 63 additions (that is, one iteration of the outer loop) each cycle. Other parts of the circuit (such as control, buffers, and memory) are not shown.

As you would expect, designing RTL circuits is time consuming. For the example in Figure 3b, designers must specify the entire structure of the data path and must also define control for reading inputs from memories into buffers, stalling the data path when buffers are full or empty, writing outputs to memory, and so on. For a typical FPGA board, a complete RTL implementation of the several lines of C code in Figure 3a can require more than 1,000 lines of VHDL. Such complexity leads to the frequent question: if compilers can extract parallelism from high-level code for multicores and GPUs, why can't they do the same for FPGAs? For more than a decade, researchers have worked toward this goal with high-level synthesis tools.[9] Commercial high-level synthesis tools are becoming increasingly common (for example, AutoPilot, Catapult C, Impulse C),

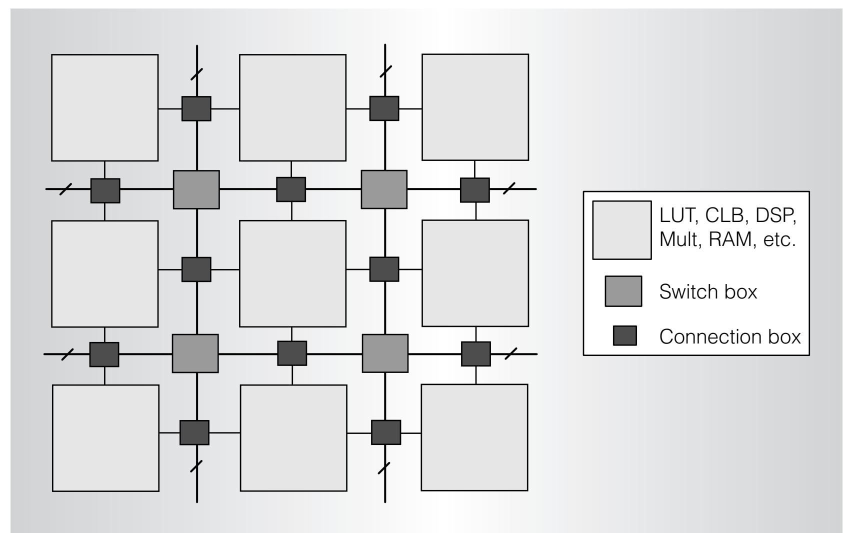


Figure 2. A typical FPGA architecture. FPGAs contain numerous routing tracks, which carry signals across the chip between computational resources (for example, configurable logic blocks [CLBs], multipliers [Mult], and digital signal processors [DSPs]). Connection boxes provide programmable connections between resource I/O and routing tracks, while switch boxes provide connections between routing tracks only.

```
void filter(float a[],
  float b[], float c[]) {
    for(int i=0;
      i < OUTPUT_SIZE; i++) {
        c[i]=0;
        for (int j=0; j < 64;
          j++) {
            c[i] += a[i+j]*b[j];
        }
    }
}
```

(a)

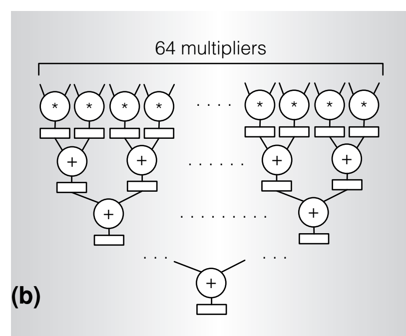


Figure 3. An example of high-level code (a) converted to a pipelined register-transfer-level data path that performs 64 multiplications and 63 additions every cycle (b). This data path fully unrolls the inner loop so that all operations can be pipelined, allowing for a single iteration of the outer loop to be executed every cycle.

and open-source tools have also begun to appear.[10,11]

Barriers to mainstream usage

FPGAs have numerous limitations, but the main barriers to mainstream usage are amenability, cost, and productivity.

Amenability

One significant barrier to mainstream FPGA usage is that not all applications are amenable to FPGA implementation. One reason is that common software constructs (such as recursion, virtual functions, and pointer-based data structures) generally don't have efficient circuit implementations. For competitive performance, FPGA applications also generally require significant parallelism owing to clock speeds that are commonly 10 to 20 times slower than CPUs and GPUs.

Such limitations aren't unique to FPGAs. GPUs have similar problems with these same higher-level software constructs and also require a significant amount of parallelism. One key difference between FPGA and GPU amenability is that high-performance FPGA applications generally require deep pipelines, such as the example in Figure 3b, where inputs and outputs continually stream through each cycle. FPGA applications must often execute hundreds or thousands of operations each cycle to outweigh slow clock frequencies. Although such parallelism can be achieved in many ways, pipelining is common because of area efficiency. For the example in Figure 3b, an alternative approach using thread-level parallelism could eliminate the pipeline registers and replicate the circuit multiple times to execute multiple iterations in separate threads. However, each thread would require an additional 64 multipliers and 63 adders, and would significantly slow down the FPGA clock by removing the registers, while also greatly increasing memory bandwidth requirements, making it difficult to reach the level of parallelism needed to account for the even-slower clock. Although this thread-based approach could save area by sharing resources or by using soft processors, these implementations would reduce each thread's throughput, in turn requiring even more parallelism. Alternatively, pipelining lets a designer generate one output every cycle with smaller area requirements and with clock frequencies that can be several times faster than non-pipelined circuits. Furthermore, pipelining and thread-level parallelism aren't mutually exclusive; in many cases, high-performance FPGA implementations will replicate pipelines to generate multiple outputs each cycle. Such replication is often referred to as ''wide'' parallelism, in contrast to ''deep'' parallelism achieved by pipelining. GPUs, by contrast, exploit thread-level parallelism by scheduling hundreds of threads simultaneously onto specialized microprocessors.

The lack of amenability for mainstream applications has also limited FPGA usage by inhibiting the emergence of a ''killer app'' supporting high sales volumes that subsidize development of new features and provide economies of scale to lower prices. Whereas GPUs became widespread owing to computer graphics' popularity in desktops, laptops, and gaming consoles, FPGAs evolved from a much smaller market of designers requiring glue logic or application-specific integrated circuit (ASIC) prototyping. Having already achieved widespread usage, GPUs are in an attractive position for use in general-purpose acceleration, whereas FPGAs are still considered niche devices. For similar reasons, the discovery of such a killer app for FPGAs would likely increase general-purpose usage, but designers would still likely resist FPGA usage because of the other barriers.

Cost

Another significant reason for mainstream FPGA resistance is high device costs. New FPGAs generally cost more than $10,000, whereas new GPUs cost several hundred to several thousand dollars. FPGA prices do fall significantly over time, and there are much cheaper, smaller FPGAs, but a designer wanting to use the newest, largest FPGA should expect a price of at least $10,000. Even worse, this price is only for the FPGA device; board vendors, IP vendors, and tools can significantly add to the total cost.

Productivity

Although device costs are a deterrent, prohibitive application design costs resulting from low productivity[12] are the most significant barrier to more widespread usage. Many FPGA designers believe the main reason FPGAs have not been more widely accepted is application design complexity that far exceeds that of other platforms. This complexity has resulted in poor productivity that has raised development costs and caused resistance to FPGAs, which in turn has prevented economy-of-scale benefits and led to significantly higher device costs. While high-end FPGAs

will likely never be as cheap as GPUs, solving the productivity problems could narrow the gap significantly.

There are numerous reasons for low FPGA productivity, but the largest problem is the requirement for digital-design expertise and knowledge of low-level device details. Because FPGAs were originally intended for implementing RTL circuits, designers have traditionally created FPGA applications using HDLs as discussed earlier, which is a time-consuming process requiring skills that aren't common to mainstream designers. For example, pipelining is generally difficult to specify because of the need to create numerous interconnections of resources with specific timing requirements. Design reuse can sometimes reduce complexity, but in general, RTL designs require at least an order of magnitude more code than traditional high-level languages. GPU applications would also require additional CUDA or OpenCL code to parallelize a function and move data back and forth to the GPU, but most designers consider GPU code to be far less complex than RTL code.

In addition to requiring device expertise, FPGA applications typically require more effort than other devices to improve amenability. For example, GPUs are generally efficient for floating-point operations, which allow many designers to ignore precision issues. FPGAs are capable of handling floating-point operations, but often achieve order of magnitude improvements when using fixed-point, integer, or bit-level operations. Therefore, to achieve an FPGA's full potential, designers might have to perform numerical analysis to determine an appropriate precision, and then implement a circuit with that precision, which can significantly increase design time.

Although high-level synthesis tools help eliminate the need for creating RTL circuits, several limitations have prevented such tools from becoming widely accepted. First, many of these tools use specialized high-level languages. Although effective for FPGA experts, these specialized languages restrict common coding constructs and styles, which mainstream application designers have resisted. High-level synthesis tools also target widespread languages such as ANSI C, but such tools are highly sensitive to coding style and constructs in order to effectively extract parallelism. These tools can force designers into a particular coding style that shares many disadvantages of specialized languages, or alternatively results in inefficient circuits for code not following strict guidelines.[13]

An increasingly significant productivity bottleneck is lengthy compilation times due to the complexity of placement and routing. Although such times have always been a nuisance to FPGA designers, it's now common for FPGA compilation to take many hours or even days.[14] This lengthy process creates a major design bottleneck that prevents mainstream usage by forcing designers to adopt methodologies that conflict with common software design practices on the basis of rapid compilation. For example, mainstream designers often debug applications by rapidly making changes, recompiling, and re-executing, which clearly isn't feasible when compilation requires hours. Furthermore, lengthy compile times prevent advances in runtime optimization, such as the dynamic compilation features of OpenCL, which designers increasingly use for GPU and multicore applications. OpenCL and similar tools could potentially make FPGAs transparent, letting designers program applications in the same way for different devices. However, stalling an application for several hours to compile one FPGA kernel will likely not be accepted. For high-level synthesis tools for other parallel languages, even with precompiled FPGA kernels, designers will likely prefer other devices due to the ability to make rapid changes.

A lack of portability is an additional productivity barrier. Unlike desktop microprocessors that achieve portability with a common instruction set, FPGA applications are specific to one product, forcing designers to recompile for other products. Even worse, because FPGAs have different numbers and types of resources, designers targeting a different product must often make significant changes to an RTL circuit.

Design tools also suffer from a lack of portability. For example, high-level synthesis tools often only support a particular type of device or a specific board, which limits their usefulness. This lack of portability contrasts with software compilers, which can map the same application code onto significantly different instruction sets.

Finally, debugging an FPGA application is significantly more complex than debugging software. Designers generally debug FPGA applications by simulating the RTL code or technology-mapped circuit and analyzing hundreds of waveforms. This requires significant effort and is further complicated by lengthy compilation. Even when using high-level synthesis, designers often debug using waveform simulations because high-level synthesis tools typically provide inaccurate thread-based simulations.[15] Even worse, simulations often differ from actual on-chip behavior,[15] which forces designers to trace waveforms of on-chip behavior using tools such as SignalTap or ChipScope.

Needed improvements

To enable mainstream use of FPGAs, researchers must achieve progress on several key problems.

High-level synthesis will continue to be a critical enabling technology for mainstream usage. Unfortunately, limited success after decades of studies suggests that such tools may never expand beyond their current niche usage. However, the recent GPU trend could actually help FPGAs with this problem by providing standardized high-level languages that omit problematic constructs. Several examples for GPUs are OpenCL, CUDA, and Brook, which FPGA researchers are now investigating for high-level synthesis. If successful, mainstream designers could potentially target any accelerator device using similar high-level code.

Reducing FPGA compilation times is also critical to enabling mainstream usage. Even in cases where FPGAs can significantly outperform GPUs, and where device cost isn't an issue, some designers are still choosing GPUs because of the improved productivity from higher-level code and rapid compilation. Surprisingly, FPGA compilation times have largely been ignored, likely owing to the common usage of FPGAs to implement custom circuits, for which designers are used to such times. Recent work has showed promising results, with compilation speedups from 50 times[16] to 500 times.[14] The current limitation of these approaches is area and performance overhead, which will need to be reduced before widespread usage is feasible.

Portability and debugging also need significant improvements, which could be partially achieved with the discussed high-level synthesis innovations. However, one problem facing the FPGA community is the lack of commonality among FPGA boards, which has led to various board architectures. Because these different boards are specialized toward targeted domains, the variations make it harder for designers to create a portable application. For example, a GiDEL PROCStar board has four FPGAs, with three external memories per FPGA. A Nallatech H101PCIXM board has one FPGA with five external memories. The XtremeData XD1000 has a single FPGA with a single external memory. Furthermore, some boards interface with a microprocessor over PCI Express, while other systems use in-socket FPGAs in place of microprocessors. Such variety also complicates high-level synthesis tools that generally assume a logical architecture that vendors port to a specific board via a platform-support package. Because of the number of FPGA boards and the small size of tool vendors, vendors can't provide platform-support packages for all boards. Even worse, platform-support packages often are incapable of using all board features for similar reasons. For example, a platform-support package might limit memory accesses to a single memory, even when multiple memories are available. Considering that memory bandwidth is often a bottleneck for FPGA applications, these board variations cause significant performance differences between high-level synthesis and custom RTL designs.

When considering how many problems must be solved to enable mainstream FPGA usage, many application designers are pessimistic about the future of FPGAs as mainstream software accelerators. However, FPGA virtualization can potentially address most of these problems. Virtual FPGA platforms, analogous to virtual machines for high-level languages such as Java, provide a set of virtual coarse-grained resources (such as floating-point cores and fast Fourier transforms) that hide the physical platforms' fine-grained complexity while also enabling transparent sharing of those resources for different applications or different threads of the same application.

To use a virtual FPGA platform, a designer specifies his or her application independently of a specific architecture. During compilation, virtualization tools analyze the application and identify a customized set of virtual resources based on the application requirements. The virtualization tools then either select an appropriate virtual platform from a library or generate a new platform specifically for the application.

By customizing a platform to the application, virtualization significantly reduces compilation times by hiding the hundreds of thousands of fine-grained lookup tables on a physical FPGA, in addition to widely varying memory architectures. Recent work has shown that such virtualization provides compilation times that are more than 500 times faster than FPGA vendor tools.[14]

By compiling applications to a virtual FPGA platform, they become portable across any physical platform that can implement the virtual platform. Such portability reduces design complexity and lets designers move applications to newer, more powerful physical platforms with potentially no effort. Similarly, virtual FPGAs enable third-party tools to target any physical platform by providing a platform-support package for one configurable virtual platform.

Another advantage of virtualization is the ability to mimic architectures used in increasingly common programming frameworks such as OpenCL and CUDA. When combined with high-level synthesis from corresponding languages,[17,18] FPGA virtualization lets designers use the same design flow as GPUs and multicore CPUs. In fact, OpenCL's dynamic compilation features could potentially hide FPGAs from mainstream designers, letting designers simply specify parallel kernels that the OpenCL framework would simultaneously schedule onto GPUs, multicores, and virtual FPGA platforms.

The main disadvantage of FPGA virtualization is the potential for significant performance and area overhead caused by implementing a virtual FPGA on top of a physical FPGA with significantly different resources. Preliminary results show a modest performance overhead of less than 10 percent. However, area requirements of some virtual FPGAs can waste up to 30 percent of the resources on a physical FPGA. Reducing this overhead is an active area of research, and the overhead is predicted to be reduced by 90 percent with improved compilation and virtual architectures.

Are FPGAs ready for the mainstream? For designers expecting similar design flows as GPUs and multicores, the answer is likely no. For designers with digital-design experience, who are willing to accept lower productivity, FPGAs can already provide significant advantages for certain domains. If the called-for innovations are all realized, FPGAs will be ready to take over a larger share of the accelerator market. In fact, FPGA virtualization has the potential to make application design for FPGAs indistinguishable from GPUs. Considering the increasing importance

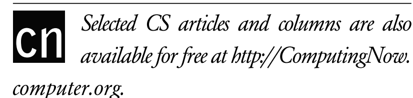
of power and energy efficiency, usage of FPGAs by mainstream designers is an exciting possibility that could significantly advance the state of the art in many domains. MICRO

References

1. S. Craven and P. Athanas, ''Examining the Viability of FPGA Supercomputing,'' *EURASIP J. Embedded Systems,* vol. 2007, no. 1, 2007, pp. 13-20.
2. A. DeHon, ''The Density Advantage of Configurable Computing,'' *Computer,* vol. 33, no. 4, 2000, pp. 41-49.
3. Z. Guo et al., ''A Quantitative Analysis of the Speedup Factors of FPGAs over Processors,'' *Proc. ACM/SIGDA 12th Int'l Symp. Field Programmable Gate Arrays* (FPGA 04), ACM Press, 2004, pp. 162-170.
4. S. Che et al., ''Accelerating Compute-Intensive Applications with GPUs and FPGAs,'' *IEEE Symp. Application-Specific Processors* (SASP 08), IEEE Press, 2008, pp.101-107.
5. J. Williams et al., ''Characterization of Fixed and Reconfigurable Multi-core Devices for Application Acceleration,'' *ACM Trans. Reconfigurable Technology and Systems,* vol. 3, no. 4, 2010, pp. 1-29.
6. G. Stitt and F. Vahid, ''Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic,'' *IEEE Design & Test of Computers,* vol. 19, no. 6, 2002, pp. 36-43.
7. A. George, H. Lam, and G. Stitt, ''Novo-G: At the Forefront of Scalable Reconfigurable Supercomputing,'' *Computing in Science & Engineering,* vol. 13, no. 1, 2011, pp. 82-86.
8. D. Burke et al., ''RAMP Blue: Implementation of a Many-Core 1008 Processor System,'' *Proc. Reconfigurable Systems Summer Institute* (RSSI 08), 2008; www.rssi2008.org/proceedings/papers/presentations/19_Burke.pdf.
9. G. Martin and G. Smith, ''High-Level Synthesis: Past, Present, and Future,'' *IEEE Design & Test of Computers,* vol. 26, no. 4, 2009, pp. 18-25.
10. A. Canis et al., ''LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems,'' *Proc. 19th ACM/SIGDA Int'l Symp. Field Programmable Gate Arrays* (FPGA 11), ACM Press, 2011, pp. 33-36.
11. J. Villarreal et al., ''Designing Modular Hardware Accelerators in C with ROCCC 2.0,'' *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 10), IEEE Press, 2010, pp. 127-134.
12. B.E. Nelson et al., ''Design Productivity for Configurable Computing,'' *Proc. Int'l Conf. Eng. Reconfigurable Systems and Algorithms* (ERSA 08), 2008, pp. 57-66.
13. S. Sirowy, G. Stitt, and F. Vahid, ''C is for Circuits: Capturing FPGA Circuits as Sequential Code for Portability,'' *Proc. 16th Int'l ACM/SIGDA Symp. Field Programmable Gate Arrays* (FPGA 08), ACM Press, 2008, pp. 117-126.
14. J. Coole and G. Stitt, ''Intermediate Fabrics: Virtual Architectures for Circuit Portability and Fast Placement and Routing,'' *Proc. IEEE/ACM/IFIP Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES/ISSS 10), IEEE Press, 2010, pp. 13-22.
15. J. Curreri, G. Stitt, and A. George, ''High-Level Synthesis of In-Circuit Assertions for Verification, Debugging, and Timing Analysis,'' *Int'l J. Reconfigurable Computing,* vol. 2011, 2011, doi:10.1155/2011/406857.
16. C. Lavin et al., ''HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping,'' *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 11), IEEE Press, 2011, pp. 117-124.
17. M. Owaida et al., ''Synthesis of Platform Architectures from OpenCL Programs,'' *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 11), IEEE Press, 2011, pp. 186-193.
18. A. Papakonstantinou et al., ''Multi-level Granularity Parallelism Synthesis on FPGAs,'' *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 11), IEEE Press, 2011, pp. 178-185.

Greg Stitt is an assistant professor in the Electrical and Computer Engineering Department at the University of Florida. His research interests include reconfigurable computing, design automation, and embedded systems. Stitt has a PhD in computer science from the University of California, Riverside.

Direct questions or comments about this column to Greg Stitt, gstitt@ece.ufl.edu.



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