INVITED PAPER

Reconfigurable Computing Architectures

This paper provides an overview of the broad body-of-knowledge developed in the field of reconfigurable computing.

By Russell Tessier, *Senior Member IEEE*, Kenneth Pocek, *Life Member IEEE*, and André DeHon, *Member IEEE*

ABSTRACT | Reconfigurable architectures can bring unique capabilities to computational tasks. They offer the performance and energy efficiency of hardware with the flexibility of software. In some domains, they are the only way to achieve the required, real-time performance without fabricating custom integrated circuits. Their functionality can be upgraded and repaired during their operational lifecycle and specialized to the particular instance of a task. We survey the field of reconfigurable computing, providing a guide to the body-of-knowledge accumulated in architecture, compute models, tools, run-time reconfiguration, and applications.

KEYWORDS | Field programmable gate arrays; reconfigurable architectures; reconfigurable computing; reconfigurable logic

I. INTRODUCTION

Field-programmable gate arrays (FPGAs) were introduced in the mid-1980s (e.g., [1]) as a larger capacity platform for glue logic than their programmable array logic (PAL) ancestors. By the early 1990s, they had grown in capacity and were being used for logic emulation (e.g., Quickturn [2]) and prototyping, and the notion of customizing a computer to a particular task using the emerging capacity of these FPGAs became attractive. These custom computers could satisfy the processing requirements for many important and enabling real-time tasks (video and signal processing, vision, control, instrumentation, networking) that were too high for microprocessors. On simulation and optimization tasks, spatial computation and specialization made it possible to achieve supercomputer-level performance at workstation-level costs. Furthermore, by reprogramming the FPGA, a specialized computer could be reconfigured to different tasks and new algorithms. In 2015, the use of FPGAs for computation and communication is firmly established. FPGA implementations of applications are now prevalent in signal processing, cryptography, arithmetic, scientific computing, and networking. Commercial acceptance is growing, as illustrated by numerous products that employ FPGAs for more than just glue logic.

Reconfigurable computing (RC)—performing computations with spatially programmable architectures, such as FPGAs—inherited a wide body-of-knowledge from many disciplines including custom hardware design, digital signal processing (DSP), general-purpose computing on sequential and multiple processors, and computer-aided design (CAD). As such, RC demanded that engineers integrate knowledge across these disciplines. It also opened up a unique design space and introduced its own challenges and opportunities. Over the past 25 years, a new community has emerged and begun to integrate and develop a body-of-knowledge for building, programming, and exploiting this new class of machines.

How do you organize programmable computations spatially? The design space of architecture and organization for the computation is larger when you can perform gate-level customization to the task, and when the machine can change its organization during the computation. When you get to choose the organization of your machine for a particular application or algorithm, or even a particular data set, how do you maximize performance, minimize area, and minimize energy?

How should you program these machines? How can they be programmed to exploit the opportunity to exquisitely optimize them to the problem? How can they be made accessible to domain and application experts?

Manuscript received August 18, 2014; revised November 13, 2014; accepted December 18, 2014. Date of current version April 14, 2015.
R. Tessier is with the ECE Department, University of Massachusetts, Amherst, MA 01003 USA (e-mail: tessier@umass.edu).
K. Pocek, retired, was with Intel USA.
A. DeHon is with the ESE Department, University of Pennsylvania, Philadelphia, PA 19104 USA.
Digital Object Identifier: 10.1109/JPROC.2014.2386883

Should they be programmed more like software or more like hardware? It became clear that neither traditional hardware nor traditional software forms of design capture can exploit the full flexibility that these machines offer. What abstractions are useful for these machines? for programmers? for platform scaling? How do you manage the resources in these machines during runtime?

How do we map and optimize to these machines? One way to ease the programmer burden is to automate the lower-level details of customizing a computation to the capabilities of these reconfigurable computers. Again, off-the-shelf CAD tools and software compilers did not address the full opportunities. Many CAD challenges, at least initially, were fairly unique to RC, including heterogeneous parallelism, hybrid space-time computations, an organizational structure that can change during an algorithm, operator customization, memory structure tuning (e.g., cache, memory sizes, and placement), and custom data encoding to the task. In some cases, optimization problems that one would reasonably have performed manually for custom designs (e.g., bitwidth selection, cache sizing) became optimization problems to solve for every application program and dataset, thereby demanding a new level of automation.

What algorithms make sense for these machines, and how does one tailor algorithms to them? The cost structure is different, often enabling or demanding different algorithms to solve the problem. Approaches that were previously inconceivable (e.g., dedicating a set of gates or a processing element to each data item) become viable and often superior solutions to problems. How do we exploit the massive level of fine-grained parallelism? These demands encouraged the RC community to identify highly parallel algorithms with regular communications before Graphics Processing Units (GPUs) and multicore chips became available with similar demands.

In this paper, we provide a guide to this accumulated body-of-knowledge. We start by reviewing the exploration of architectures for RC machines (Section II). Section III reviews approaches to program these RCs. We then review developments in tools to automate and optimize design for RCs (Section IV) and models and tools for Run-Time Reconfiguration (RTR, Section V). Section VI highlights important application domains. We conclude with some final observations in Section VII.

II. ARCHITECTURE AND TECHNOLOGY

The invention of FPGAs in the early 1980s [1] seeded the field that has become known as reconfigurable computing. FPGAs offered the lure of hardware performance. It was well known that dedicated hardware could offer orders of magnitude better performance than software solutions on a general-purpose computer and that machines custom-built for a particular purpose could be much faster than their counterparts. However, building custom hardware is expensive and time consuming. Custom VLSI was the domain of a select few; the numbers that could profitably play in that domain were already small in the 1990s and have shrunk considerably since then. Microprocessor performance scaled with Moore's Law, often delivering performance improvements faster than a custom hardware design could be built. FPGAs provided a path to the promise of hardware customization without the huge development and manufacturing costs and lead times of custom VLSI. It was possible to extract more computational throughput per unit silicon from FPGAs than processors [3], and it was possible to do so with less energy [4].

In the late 1980s it was still possible to provide hardware differentiation by assembling packaged integrated circuits in different ways at the printed-circuit board level. However, as chips grew in capacity and chip speeds increased, the cost of accessing functions off chip grew as well. There was an increasing benefit to integrating more functionality on chip. Opportunities for board-level differentiation decreased, increasing the demand for design customization and differentiation on chip. FPGAs provided a way to get differentiation without using custom VLSI fabrication.

The challenge then is how should we organize our computation and customization on the FPGA? How should the specialized FPGA be incorporated into a system? How can the computation take advantage of the FPGA capabilities? As FPGAs grow in capacity and take on larger roles in computing systems than their initial glue-logic niche, how should FPGAs evolve to support computing, communication, and integration tasks?

A. Pioneers

Just a few years after the introduction of FPGAs, multiple pioneering efforts demonstrated the potential benefits of FPGA-based RC. SPLASH arranged 16 FPGAs into a linear systolic array and outperformed contemporary supercomputers (CM-2, Cray-2) on a DNA sequence matching problem with a system whose cost was comparable to workstations [5]. Programmable active memories (PAM) arranged 16 FPGAs into a two-dimensional (2-D) grid and demonstrated high performance on a collection of applications in signal and image processing, scientific computing, cryptography, neural networks, and high bandwidth image acquisition [6]. PAM held the record for the fastest RSA encryption and decryption speeds across all platforms, including exceeding the performance of custom chips.

B. Accelerators

A key idea from the beginning was that FPGAs could serve as generic, programmable hardware accelerators for general-purpose computers. Floating-point units were well known and successful at accelerating numerical applications. Many applications might benefit from their own

custom units. The question is how should they be interfaced with the general-purpose CPU, and how should the RC be managed?

When floating-point co-processors were still separate chips, PRISM showed how an FPGA added as an external co-processor could accelerate bit-level operations that were inefficient on CPUs [7]. As chip capacities grew, research explored architectures that integrated an FPGA or reconfigurable array on the same die with the processor [8]–[10]. Key concerns included the visible architectural model for the array, how state was shared with the configurable array, how different timing requirements of the processor and array should be accommodated, and how to maximize the bandwidth and minimize the latency required to communicate with the array. The reconfigurable logic could be integrated as a programmable functional unit for a RISC processor [8], [11], share state with the conventional register file, and manage the reconfigurable array as a cache of programmable instructions [12].

It was soon apparent that the processor would become a bottleneck if it had to mediate the movement of all data to and from the reconfigurable array. GARP showed how to integrate the array as a co-processor and how the array could have direct access to the memory system [13]. The architecture for dynamically reconfigurable embedded systems (ADRES) also provided direct memory access in a co-processor model; ADRES combined a VLIW processor core with the reconfigurable array, sharing functional units between the VLIW core and the array [14]. Later work explored interfacing the reconfigurable logic with on-chip caches and virtual memory [15] and streaming operations that used scoreboard interlocks on blocks of memory [16].

Accelerators can also have an impact beyond raw application performance. Offloading tasks to a reconfigurable array can be effective at reducing the energy required for a computation [17], as is important for embedded systems. For a survey of both fixed and reconfigurable accelerators and estimates of their energy and performance efficiency, see [18]. To simplify design representation, mapping, and portability, the Queue Machine [19] showed how a processor and an array could run the same machine-level instructions.

These designs at least anticipated and perhaps helped motivate commercial processor-FPGA hybrids. Xilinx offered a PowerPC on the Virtex2-Pro and now integrates ARM cores on their Zynq devices. Altera includes ARM cores on Cyclone V and Arria V SoC FPGAs. Stretch provided an integrated processor and FPGA device [20]. Intel integrated an Atom processor and an Altera device in a multichip package for embedded computations and also integrates a Xeon with an FPGA for server applications.

Multiprocessors may also benefit from attached accelerators. The logic could be used for custom acceleration, similar to its use in single node machines, or for improving communication and synchronization [21], [22]. Cray integrated FPGAs into their XD1, SRC offered a parallel supercomputer with FPGA accelerators, and Convey Computer now ships a supercomputer with an FPGA acceleration board.

C. Fabric

Using FPGAs as computing substrates creates different needs and opportunities for optimization than using FPGAs for glue logic or general-purpose register-transfer level (RTL) implementation engines. As a result, there is a significant body of work exploring how reconfigurable arrays might evolve beyond traditional FPGAs, which we highlight in this section. For the evolution of FPGAs for traditional usage, including the impact of technology, see the companion article [23]. For a theoretical and quantitative comparison of reconfigurable architectures, see [24].

1) Organization: Fine-grained reconfigurable architectures, such as FPGAs, give us freedom to organize our computation in just about any way, but what organizational patterns are actually beneficial for use on the FPGA? This freedom fails to provide guidance in how to use the FPGA. As a result, there has been considerable exploration of computing patterns to help conceptualize the computation. These ideas manifest as conceptual guides, as tools, as overlay architectures, and as directions for specializing the reconfigurable fabric itself. Overlay architectures are implemented on top of existing FPGAs and often support tuning to specialize the generic architecture to the specific needs of a particular application.

Cellular automata is a natural model for an array of identical computing cells. On the FPGA, the cellular processing element can be customized to the application task. The CAL array from Algotronix was designed specifically with this model in mind [25]. Others showed how to build cellular computations on top of FPGAs [26] including how to deal with cellular automata that are larger than the physical FPGA available [27]–[29].

Dataflow models are useful in abstracting spatially communicating computational operators. Dataflow handshaking can simply be used to synchronize data- or implementation-dependent timing [30], but it can also be used to control operator sharing [31].

When you must sequentialize a larger graph onto limited resources, very long instruction word (VLIW) organization provides a straightforward model for operator control. As an overlay architecture, the composition of operators can be selected to match the needs of the application [32], [33]. As we will see later (Section II-C4 and C5), when we time-switch the fabric itself, the result is essentially a VLIW organization.

Highly pipelined vector processors can be efficient for data parallel tasks [34]. The vector lanes provide a model

for trading area and time [35] and can be extended with custom accelerators [36].

Some have found multicore and manycore overlays to be useful models for mapping applications onto FPGAs [37]–[39].

2) Coarse-Grained Blocks: While the gate-level configurability in FPGAs allows great customization, it also means that common computing blocks used in DSP and numerical computations are inefficient compared to custom building blocks (e.g., multipliers, floating-point units). Would it be better to provide custom elements of these common computing blocks alongside or inside the FPGA fabric? When FPGAs were small, it was reasonable to consider adding dedicated chips for floating point at the board level [40]. The virtual embedded block model was introduced to explore hard logic integration in an FPGA array and illustrated the benefits of coarse-grain hardware to support floating-point computations [41]. Embedded multipliers and memories help narrow the performance, energy, and area gap between application-specific integrated circuits (ASICs) and FPGAs [42]. The modern extensions to Toronto's FPGA CAD flow supports exploration of customized blocks [43]. Both Altera and Xilinx now embed hard logic DSP blocks that support wide-word addition and multiplication inside their fine-grained logic arrays.

3) Synchronous, Asynchronous: Processors and ASICs are typically designed around a fixed clock. This allows the integrated circuit (IC) designer to carefully optimize for performance and means the user of the processor does not have to worry about timing closure for their designs. As we consider high throughput computational fabrics, fixed-frequency, clocked arrays can offer similar advantages to spatial computing arrays like FPGAs, as shown by the high-speed, hierarchical synchronous reconfigurable array (HSRA) [44] and SFRA [45]. A different way of avoiding issues with timing closure is to drop the clock completely for asynchronous handshaking, as shown in [46]–[48].

4) Multicontext: Since FPGAs started out as gate array replacements, they contained a static configuration to control the gates and interconnect. The fact that this configuration could be changed opened up new opportunities to change the circuit during operation. However, FPGA reconfiguration was slow. Adding multiple configurations to the FPGA allows the FPGA to change rapidly among a set of behaviors [49], [50]. By time-multiplexing the expensive logic and interconnect, this effectively allowed higher logic capacity per unit silicon, often with little performance impact. Tabula now offers a commercial multicontext FPGA [51]. The companion paper [24] identifies conditions under which a multicontext FPGA can be lower energy than a single-context FPGA.

5) Coarse-Grained: Early FPGAs contained fine-grained logic largely because of their limited capacity, their gate array roots, and their use as glue logic. As their application domain moved to address signal processing and computing problems, there was interest in more efficiently supporting larger, wide-word logic. Could we have flexible, spatially configurable, field-programmable machines with wide-word, coarse-grained computing blocks? Would a small amount of time-multiplexed sharing of these units be useful?

Designed to prototype DSP algorithms, PADDI [52] was one of the first coarse-grained reconfigurable architectures. It essentially used a VLIW architecture, with 16b words and 8 contexts per processing element. Other VLIW-style architectures have been used for low-power multimedia processing, including the 16b ALU-register-file Reconfigurable Multimedia Array Coprocessor (REMARC) array [53], the 32b-word ADRES [54] with 32 local contexts and 8 entry register files, and the 8b-word, 16 entry register file dynamically reconfigurable processor (DRP) [55].

The reconfigurable datapath architecture (rDPA) used 32b-wide ALUs in a cellular arrangement with data presence flow control [56]. The reconfigurable pipelined datapath (RaPiD) arranged coarse-grain elements, 16b ALUs, multipliers and RAM blocks in a one-dimensional (1-D) datapath for systolic, pipelined computations, using a parallel, configurable bit-level control path for managing the dynamic behavior of the computation [57]. PipeRench also used the model of a directional pipeline, adding a technique to incrementally reconfigure resources and virtualize the depth in the pipeline, allowing developers to be abstracted from the number of physical pipe stages in the design and allowing designs to scale across a range of implementations [58].

To reduce the cost of reconfiguring the cells during a multimedia computation, MorphoSys used a single instruction, multiple data (SIMD) architecture to control a coarse-grained reconfigurable array with tiles composed of 16b ALUs, multipliers, and 4 element register files [59]. Morphosys uses a 32-context instruction memory at the periphery of the array that is as wide as a side of the 2-D array. Instructions can be shared across rows or columns in the 2-D array, and the memory is wide enough to provide separate control for each row or column.

Graphics processors evolved from fixed-function rendering pipes to general-purpose computing engines in the form of general-purpose graphics processing units (GPGPUs) around the same time that FPGAs moved into computing and Coarse-Grained Reconfigurable Arrays (CGRAs) were developing. At a high-level, GPGPUs share many characteristics with CGRAs, providing a high-density spatial array of processing units that can be programmed to perform regular computing tasks. GPGPUs were initially focused on SIMD, single-precision, floating-point computing tasks, and do not support the spatially local

communications typical of FPGAs and CGRAs. The companion paper [24] shows why locality exploitation can be a key advantage for reconfigurable architectures. For compiled CUDA, the native language of many GPGPUs, the GPGPUs provide 1–4× higher throughput than FPGAs, but at a cost of over 4–16× the energy per operation [60]. The peak, single-precision, floating-point throughput of GPGPUs exceeds processors and FPGAs, but the delivered performance on applications can be lower than FPGAs and the energy per operation can be higher [61]. FPGAs can outperform GPGPUs [62] for video processing, depending on the nature and complexity of the task.

A key differentiator in the architectures we have seen so far is whether the reconfigurable resources are controlled with a static configuration, like FPGAs, or with multicontext memories, like the VLIW-style CGRAs above. A second differentiator is how many resources can share the same instructions as the SIMD designs exploit. This motivated a set of designs that made it possible to use configuration to select between these choices. MATRIX [63] explored configuring instruction distribution to efficiently compose a wide range of architectures (systolic, VLIW, SIMD/Vector, MIMD) using a coarse-grain architecture based around an 8b ALU-multiplier cell that includes a local 256 × 8 RAM. The RAMs could be used either as embedded data memories (e.g., register files, FIFOs) or instruction stores and the programmable interconnect could carry both data streams and instruction streams, including streams that could control the interconnect. The CHESS reconfigurable arithmetic array used 4b ALUs and datapaths with local 32 × 4 memories where the ALUs could be configured statically or sequenced dynamically from local memories, but the routing could not [64].

If the domain is large enough, it makes sense to create a custom reconfigurable array specialized for a specific domain. The Totem design showed how to optimize a reconfigurable array for a specific domain and illustrated the area savings in the DSP domain [65], how to perform function allocation for a domain-customized architecture [66], how to provide spare capacity for later changes [67], how to automate the layout of domain-optimized reconfigurable arrays, and how to compile applications to a specialized instance of the architecture.

Some of the ideas from the MATRIX CGRA led to the array of SpiceEngine vector processors in Broadcom's Calisto architecture [68]. The Samsung reconfigurable processor (SRP) is a derivative of the ADRES CGRA architecture [69], [70].

6) Configuration Compression and Management: While FPGAs are in-field reconfigurable, the slow reconfiguration times limit the use of reconfiguration during the execution of a task. One key reason configurations are slow is the large number of configuration bits required to specify the FPGA. Consequently, one way to accelerate reconfiguration is to compress the transfer of configuration data. By adapting ideas from covering in two-level logic minimization, the wildcard scheme on the Xilinx XC6200 could be effectively used to reduce bitstream size and load time [71]. When the sequence of configurations is not static, a cache for frequently used configurations can be used to accelerate FPGA reconfiguration [72].

7) On-Chip Dynamic Networking: The load-time configured interconnect in FPGAs is suitable for connecting together gates or performing systolic, pipelined, and cellular computations. However, as FPGAs started hosting more diverse tasks, including computation that used interconnect less continuously, it became valuable to explore disciplines for dynamically sharing limited on-chip FPGA communication bandwidth—to provide network-on-chip (NoC) designs on FPGAs. This prompted the design of packet-switch overlay networks for FPGAs [73], [74]. Often a time-multiplexed overlay network can be more efficient than a packet-switched network [75]. Because of the different cost structure between ASICs and FPGAs, overlay NoCs on FPGAs should be designed differently from ASIC NoCs [76], [77]. Ultimately, it may make sense to integrate packet-switched NoC support directly into the FPGA fabric [78].

D. Emulation

Since FPGAs are programmable gate arrays, an obvious use for them was to emulate custom logic designs. However, since FPGA capacity is lower than contemporary and future ASICs, there is typically a need to assemble large numbers of FPGAs to support an ASIC design [2]. The lower density I/O between chips compared to on-chip interconnect meant that direct partitioning of gate-level netlists onto multiple FPGAs suffered bottlenecks at the chip I/O that left most FPGA logic capacity underutilized. To avoid this bottleneck, Virtual Wires virtualized the pins on the FPGA, exploiting the ability to time multiplex the I/O pins many times within an ASIC emulation cycle [79]. FPGA-based Veloce logic emulators sold by Mentor Graphics are based on these techniques. Today's FPGAs include hardware support for high-speed serial links to address the chip I/O bandwidth bottleneck.

As FPGAs grew in size, natural building blocks could fit onto a single FPGA, including complete processors [80]. The emulation focus turned to the system level. Boards of FPGAs were used to directly support algorithm development [81]. With an FPGA now large compared to a processor core, single FPGAs and boards of FPGAs have been used to support multicore architecture research [82], [83]. Nonetheless, directly implementing one instantiated processor core on the FPGA for one emulated processor core can demand very large multi-FPGA systems. Since these designs include many, identical processor cores, an alternative is to time-multiplex multiple virtual processor cores over each physically instantiated core [84].

Furthermore, infrequently exercised core functionality can be run on a host processor, allowing the physically instantiated core to be smaller and easier to develop.

While it is possible to simulate the entire processor on an FPGA, defining and maintaining the model for the FPGA and keeping it running at high frequency can be difficult. FPGA-accelerated simulation technologies (FAST) observes that cycle-accurate simulation on a processor is dominated by the timing modeling, not the functional modeling. Consequently, one can move the timing model logic to the FPGA and achieve significant simulation speedups without significantly changing the design and reuse flow for the functional simulator [85]. A-Port networks provide a generalization for explicitly modeling the simulated timing of a structure where the FPGA primitive clock cycles do not necessarily correspond directly to the modeled timing cycles in the emulated system [86].

E. Integrated Memory

As FPGA capacity grew, it became possible to integrate memory onto the FPGA die, and it became essential to do so to avoid memory bottlenecks. This created opportunities to exploit high, application-customized bandwidth and raised questions of how to organize, manage, and exploit the memory.

Irregular memory accesses often stall conventional processor and memory systems. Consequently, an FPGA accelerator that gathers and filters data can accelerate irregular accesses on a processor [87]. Similarly, conventional processors perform poorly on irregular graph operations. The GraphStep architecture showed how to organize active computations around embedded memories in the FPGA to accelerate graph processing [88].

When it is not possible to store the entire dataset on chip, it is often useful to stream, buffer, or cache the data in the embedded memories. Windows into spatial regions are important for cellular automata [27] and many image and signal processing and scientific computing problems [89]. CoRAM provided an abstraction for using the on-chip memories as windows on a larger, unified off-chip memory and a methodology for providing the control and communication infrastructure needed to support the abstraction [90].

The embedded memories in FPGAs are simple RAMs, often with native support for dual-port access. It is straightforward to build scratchpad memories, simple register files, FIFOs, and direct mapped caches [91] from these memories. Recent work has demonstrated efficient ways to implement multiported memories [92] and (near) associative memories [93].

F. Defect and Fault Tolerance

We can exploit the homogeneous set of uncommitted resources in FPGA-like architectures to tolerate defects. Because of this regularity, FPGAs can use design-independent defect-tolerance techniques such as row and column sparing, as is familiar from memories. Nonetheless, the tile in an FPGA is much larger than a memory cell in a RAM, so it may be more efficient to spare resources at the level of interconnect links rather than tiles [94]. However, this kind of sparing builds in two levels of reconfigurability: one to tolerate fabrication defects and one to support the design. It is more efficient to unify our freedom for design-mapping and defect-avoidance. With today's cluster-based FPGAs, it is simple to reserve a spare lookup table (LUT) in a cluster to tolerate LUT defects [95]. When a natural cluster does not already exist, it is possible to overlay a conceptual cluster and reserve one space within every small grid of compute blocks so that it is easy to precompute logic permutations that avoid any single compute block failure within the overlay grid [96]. Interconnect faults can be repaired quickly by consulting precomputed alternate paths for nets that are disrupted by defects [97]. These techniques repair the design locally, avoiding the need to perform a complete mapping of the design; the cost for this simplicity is the need to reserve a small fraction of resources for spares even though most will not be used.

If we must tolerate higher defect rates or reduce the overhead for spares, we can map around the specific defects in a device. TERAMAC pioneered the more aggressive, large-scale approach of identifying defects, both in the FPGA and in the interconnect between FPGAs, and mapping an application to avoid them [98]. This style of design- and component-specific mapping can be used to tolerate defects in Programmable Logic Arrays (PLAs), even at the high defect rates expected in molecular-scale designs [99]. The TERAMAC [100] and nanoPLA [101] designs show how reconfigurable architectures can accommodate messy, bottom-up fabrication technologies. In order to perform this device-specific mapping, it is necessary to map out the defects in the design. Reconfigurability can also be exploited to efficiently test for defects in an FPGA or molecular-scale interconnect [102]. Xilinx's EasyPath program exploits the ability to map around defects by matching partially defective FPGAs with specific bitstreams and offers them at a reduced price.

As feature size continues to scale down and we explore post-silicon technologies, variation increases, potentially degrading FPGA performance [103]. At modest variation levels, the worst-case effects from variation-induced slowdown on a few paths can be avoided using multiple precomputed configurations [104]. For larger variation, the defect-avoidance techniques reviewed above can minimize the impact of variation [105]. This will require that we characterize the delays of FPGA resources, which we can also do by exploiting FPGA reconfigurability [106], [107].

Shrinking feature sizes also make components more susceptible to failures during operation due to wear-out

effects. Partial reconfiguration can be used to interleave tests for failures during operation with quick reconfigurations to mask the new failures [108]. Resources in a design will not be used uniformly, both this and the fact that not all resources will be used opens up an opportunity to remap the design during operation to even out the wear on resources, extending the component's lifetime [109].

Soft errors can change the configuration bits in SRAM-based FPGAs, leading them to perform incorrect functions. The impact of these upsets is quantified in [110]. The expected upset rates from radiation for systems varies based on size and location; while it has traditionally been a concern primarily for space-based systems, it is becoming a real concern for airplanes and large ground-based systems [111]. The Cibola Flight Experiment provides useful statistical information for radiation upsets in satellite systems [112]. Triple modular redundancy (TMR) is a common solution for systems that cannot tolerate outage periods. Partial TMR reduces the cost of TMR protection when transient errors are acceptable but persistent state errors must be avoided [113]. To avoid single points of failure and minimize performance degradation for TMR, additional care is needed during placement [114]. To minimize the impact of configuration upsets, Xilinx provides designs to scrub an FPGA bitstream and identify and correct configuration bit upsets.

G. Energy

As noted, many scenarios have been identified where reconfigurable architectures use substantially less energy than processors and GPGPUs. Instruction and data memory consume most of the energy in processors, while spatial architectures eliminate instruction energy and reduce data memory energy using smaller, distributed memories local to the computation. The companion paper shows how the locality exploitation in spatially distributed computations fundamentally allows them to achieve lower energy than sequential computations [24].

H. Looking Forward

As technology continues to scale towards the atomic scale, energy and reliability are emerging as some of the biggest challenges facing the electronics landscape. In today's environment, a reconfigurable architecture's ability to reduce energy is often as important as its ability to reach new levels of performance. Power concerns have slowed processor clock frequency scaling, making spatial architectures that exploit parallelism increasingly attractive. Furthermore, power concerns are driving increasing interest in specialized accelerator architectures and heterogeneous computational organizations on a single chip—both areas that have seen significant development within this community. Reconfiguration remains a promising way to address the reliability challenges of highly scaled CMOS, ultra-low voltage CMOS, and post-CMOS computing substrates. These trends suggest that the body-of-knowledge developed for RC architecture is increasingly valuable to the design of all kinds of future computing systems.

III. LANGUAGES AND COMPUTE MODELS

In the early days of RC, programming for general-purpose computers and FPGAs was split into two very different domains. While procedural languages like C were generally used to target microprocessors, most FPGA application designers were still burdened with drawing schematics and writing Boolean equations. Hardware description languages (HDLs), such as Verilog and VHDL, were gaining a footing, but HDL synthesis at the time tended to produce designs that were larger and slower than hand-crafted designs. In general, achieving reasonable design performance using the limited logic and routing resources in most devices required hand tuning and an understanding of the basic FPGA architecture.

For RC machines to be accessible to the masses, new programming environments and models of computation would be needed. Since the manycore era was still a human generation away, the massive parallelism available in RC required designers to consider succinct ways of expressing and exploiting massive parallelism long before these issues were mainstream in the general-purpose computing community. However, a desire to maintain familiarity for programmers of microprocessors created a dilemma: Should familiar processor-based languages and compute models be migrated to reconfigurable platforms or should whole new models be created?

Languages and compute models for RC must serve two goals. First, they must allow designers to express applications in a succinct fashion that can be verified prior to hardware implementation. Second, and perhaps more importantly, application representations must be amenable to efficient compilation to the target hardware platform to take advantage of the fine-grained parallelism and specialization offered by reconfigurable devices. Since not every type of application requires the same type of parallelism (fine-grained, coarse-grained, memory-intensive, etc.), a range of languages and models are needed. These domain-specific languages and compute models often are not unique to RC. In fact, as hardware synthesis and parallel computing methodologies have improved, many of the techniques from these domains have been directly applied to RC, with a series of required changes.

A. C-to-Hardware

A key goal in the early days of RC was to make the programming environment for RC systems as similar as possible to microprocessor-based systems to increase

accessibility for a large population of programmers. Since C was the primary language of the day, it was natural that this procedural language was a starting point. In many respects, C is not an ideal choice for hardware development since it does not have explicit support for clock domains, fine-grained parallelism, or synchronization. An initial C-to-hardware compiler [115] converted simple chains of C operations (e.g., add, shift) into HDL code that was then compiled to the gate-level using synthesis tools. This work demonstrated the benefit of specialization as multiple sequential C operations could be combined into a single specialized hardware unit. The idea was then extended [116] to consider the use of a simple state machine to execute multiple sequential hardware operations for each extracted C block.

Since C syntax is limiting for hardware design, features were added to the language to better express parallelism and synchronization. Early compilers often relied on users to manually identify parallelism in C programs using pragma statements [117]. Expressions or whole subroutines could be selected for synthesis to target hardware. User-defined program annotation to support bitwidth specification and communication between the synthesized hardware and the rest of the circuit provided a mechanism for designers to guide the C-to-FPGA compilation process.

Over the past decade, advancements in C-to-hardware synthesis for both ASICs and FPGAs have moved this design style into the mainstream. Modern systems can automatically extract parallelism from C source code [118] and consider the interface between the synthesized hardware and memory [119]. Application code can also be profiled using software execution [120] to determine target code for hardware acceleration. An important aspect of C-to-FPGA compilers is the estimation of functional unit area and performance [121], an issue made easier by the recent inclusion of hard macro blocks in FPGAs. For most contemporary systems [120], [121], all C constructs are supported, including pointers, recursion, and subroutines. Specialized systems have also been created to target synthesis for floating point [122].

The research associated with RC has spawned a number of commercial procedural language-to-hardware offerings over the years, including Xilinx's Vivado (based on AutoESL's AutoPilot), Calypto's Catapult C, and Cadence's C-to-Silicon. A demand for higher productivity in the broader design automation community has also fueled a push to C- and system-level design and synthesis. Extensive improvements have led to increasing acceptance and a slow FPGA designer migration from design specification in HDL to specification in these higher-level languages.

B. MATLAB-to-Hardware

As the use of FPGAs in commercial products increased, so did the range of languages targeted to the devices. By the end of the 1990s, the use of MATLAB for developing and testing signal processing (SP) applications had become widespread. Since many of these SP designs were ultimately targeted to FPGAs, it was only natural that techniques to compile MATLAB to FPGA-based systems would be created. The loop-based nature of many MATLAB programs makes them amenable to parallelization with minimal control overhead. An initial effort [123] focused on the development of an FPGA library of common MATLAB functions, such as matrix multiplication and filtering. The compiler identifies these modules, adjusts bit widths and control (e.g., loop counters), and makes inter-module connections. Later work [124] examined techniques for implementing MATLAB functions in FPGAs in a power-efficient manner using parameterized designs. A MATLAB mapping approach that models data flow control as a Kahn process network [125] provided relaxed inter-module control synchronization. Inter-module buffering is used to support distributed control. MATLAB support is now a staple in FPGA development tools (e.g., Altera's DSP Builder, Xilinx's System Generator, and Mathwork's HDL Coder) and is widely used for the implementation of SP blocks in FPGA systems.

C. Pipelined Computation and Communication

Since their introduction in the 1980s, LUT-based FPGAs have been recognized as desirable platforms for repetitive, datapath-oriented applications due to their inclusion of numerous flip flops. Early FPGA designers realized that a flip flop requires so little silicon area compared to a LUT and associated routing that it makes sense to include one flip flop per LUT in the FPGA. This design choice provides the opportunity for pipelining at both the fine- and coarse-grain levels. Perhaps the best example of the potential of FPGAs for fine-grain pipelining was demonstrated by von Herzen [126] who showed that an FPGA was able to deliver a raw clock speed 2 to 3× faster than a microprocessor in the same technology on heavily pipelined SP applications. This performance is only achieved through careful consideration of logic placement and interconnection. In general, common SP algorithms for image, audio, and radar processing algorithms that require little synchronization can take advantage of a systolic compute model where computation and communication take place in lockstep (see Section VI-A).

RC platforms are also amenable to coarse-grain systolic computation, especially when all compute blocks perform the same function. SIMD computing on the Splash 2 architecture [127] provided an early demonstration of this compute model. In this architecture, an instruction was globally distributed to 16 FPGAs, all performing the same operation. Data was then transferred between the devices in a systolic fashion. The repetitiveness of the computation also helped scalability; one optimized compute block could be created, optimized and replicated many times.

More sophisticated pipelined approaches followed as parallelizing compilers were developed for the general-purpose computing community and were migrated for use in RC platforms. One approach [128] automatically generated pipelined co-processors using loop unrolling that allowed for the execution of multiple loop iterations in parallel. This compiler focused on the removal of loop dependencies, a limitation in unrolling. As the systems matured, access to external memory became a bigger issue to keep the relevant pipelines full. A parallelizing compiler [129] explicitly considered memory access time in the generation of pipelines. The compiler algorithms are able to trade off communication and computation in different stages of the pipeline. More recent work [130] examined the use of dynamic data dependency analysis in allowing multiple pipelines to proceed simultaneously. A limitation in many pipelined implementations is the separation of pipeline datapath and control. VLIW-SCORE [131] overcomes this issue by distributing control across the pipeline.

Although the pipelining of data computation in RC machines is still important in today's systems, newer systems can also be bound by the time required for accesses to the memory hierarchy. In general, contemporary commercial C-to-hardware compilers support pipelining operations involving loop unrolling and tiling.

D. Support for Integrated and External Memory

As compiler technologies advanced and embedded memory blocks were added to FPGAs, more sophisticated design mapping systems were developed. In many cases, access to memory internal to the FPGA devices became the limiting factor in the high-performance implementation of compute models. Even though internal FPGA block memories are multi-ported, prioritizing memory accesses is a necessity, especially when programmable interconnect delays are taken into account. In response, a parallelizing compiler [37] that efficiently assigned unrolled loop data to on-chip block memories was developed. Data and computation were tightly clustered into small tiles to minimize time-consuming inter-tile communication using FPGA interconnect. An alternative approach [132] used a precedence graph to assign data arrays to multiple memory banks located in both internal and external FPGA memory. This type of dependency analysis is particularly important for the support of loop unrolling. Although block RAMs are attractive and necessary compilation targets, their on-chip quantities are limited. In some cases, it is desirable to consider the tradeoffs of storing data in banks of flip flops rather than block RAM [133]. This compilation approach has been shown to reduce overall execution time while limiting block RAM bandwidth. Recent work [134] focuses on mapping for systems that contain multiple reconfigurable accelerators, each of which has a memory interface including a cache. Memory accesses are optimized based on predetermined accelerator access patterns.

Advances in the use of memories by compiler technologies have encouraged FPGA companies to include specialized DRAM interfaces and large numbers of block RAMs in their devices. Convey Computer offers specialized scatter-gather external memory configurations for its multi-FPGA systems in an effort to accelerate random external memory accesses.

E. Object-Oriented and Streaming Compute Models

The similarity between instantiated hardware modules and objects in object-oriented programming languages has led to a number of attempts to represent RC as parallel collections of communicating objects. A key aspect of these models is predictable communication flow and limited communication dependencies. Although similar to pipelined implementations, streaming applications typically have coarse-grain compute objects that communicate with adjacent blocks via buffers or synchronized communication channels. Results are often sent along predetermined communication paths at predictable rates. This model is particularly useful for series of SP blocks that require minimal control flow or global synchronization. PamBlox focused on the ability to define hierarchies of C++ objects [135] and the use of embedded block memories. These blocks could then be organized into streams. The Streams-C model [136] introduced a series of communicating sequential processes that used small local memories. Streams-C tools were later commercialized into the Impulse-C compiler. Perhaps the most comprehensive stream-based, object-oriented environment to date was the commercial Ambric model [137]. In this model, a simple processor executed one or more user-defined objects that communicated with objects in other processors via self-synchronizing, dataflow channels. The commercial Bluespec SystemVerilog hardware synthesis system is also based on the manipulation of objects.

An attractive aspect of stream-based environments is the ability to abstract away details of the reconfigurable implementation platform from the user's application specification. Several projects have considered the possibility of combining stream-oriented computation with run-time reconfiguration. JHDL [138] allowed for the definition of objects whose functionality can be dynamically changed. Development tools allowed for evaluation of system performance using both simulation and in-circuit execution. The SCORE project [139] explored swapping stream-based objects on-demand at run-time. If the number of objects in an application is too large to fit in the hardware, objects can be swapped in and out as needed. As a result, the same application could be mapped to hardware platforms of many different sizes.

F. Many-Threaded Models

The emergence of manycore processors and GPGPUs has allowed for the crossover of computing models from

these domains to RC. A defining aspect of these systems is the presence of large numbers of threads of control in a single processor or across many processors. An initial effort in this space [140] examined the cost/benefit tradeoffs for multithreaded soft processor implementations on FPGAs. This work was later extended [141] to consider implementation issues associated with multiple multi-threaded soft processors on an FPGA. Since control-oriented threads are often better-suited for fixed-silicon microprocessors, a run-time system which distributes threads across heterogeneous resources (FPGA or CPU) [142] is desirable. Abstract data types can be passed between the threads, expanding flexibility. The recent popularity of OpenCL and CUDA GPGPU languages that facilitate thread creation and management has led to increased interest in the compiler systems that will map these languages to multiple FPGA compute kernels. FCUDA [60] translates CUDA threads to register-transfer level code that can be efficiently synthesized into FPGA logic. Altera has developed a similar, more comprehensive synthesis system based on OpenCL [143].

G. Looking Forward

Many of the challenges in terms of the automatic extraction of parallelism facing the manycore computing community are the same ones that have been present for RC for the past 25 years. Continued advances in both fields will help better define languages and compute models. The emergence of extreme levels of thread-level parallelism in manycores and GPGPUs will provide new insights into opportunities to leverage the massive fine-grained parallelism of RC. As hardware design continues to migrate from HDL to the behavioral level, reconfigurable resources will be more accessible to programmers with limited hardware backgrounds.

Over the past few years, massively distributed computing using thousands of processors in a "cloud" has gained increasing attention. It has been demonstrated that FPGA resources deployed in a cloud environment can be accessed with the same OpenStack software technology used to access virtual machines [144]. Although no commercial products are yet available, Microsoft has remained an active investigator in examining the use of FPGAs in the cloud. A recent paper examined the use of 1 632 servers equipped with Stratix V FPGAs to accelerate the Bing search engine. A throughput increase of 95% is achieved for a fixed latency distribution [145]. In 2011, Maxeler introduced MaxCloud, based on multiple Virtex 6 FPGAs, as an on-demand, high-availability cloud computing environment.

IV. TOOLS

The strength of FPGAs and other reconfigurable architectures is their flexibility. Unfortunately, this flexibility can make design, optimization, and debug of RC designs a challenge for designers who are not hardware experts. While most general-purpose computer systems have long been supported by advanced profilers and debug infrastructures with single-stepping and breakpoints, design and test tools for FPGA-based systems have evolved to this point from humble beginnings. Early designers used gate-level design manipulation to achieve desired performance and gate-level simulation to isolate design bugs. Today's advanced tools allow for a broad range of design options in different languages, simulation at multiple design levels, and in-circuit testing with near 100% visibility of internal logic signals. The evolution of design, test and debug tools, in concert with language and compiler development, has moved RC into the mainstream computing community. Although standard FPGA synthesis and physical design tools are often a core component of this tool environment, an additional layer of tools that customizes the mapping of designs to the reconfigurable platform and bridges the debug gap between intermediate design results and the original design specification is needed to fully support a broad range of designers.

A. Design Environments

The development of an RC application requires a number of steps including design specification, profiling, performance estimation, compilation, deployment, and testing. A number of design environments have been developed that support the initial three tasks in this set. A comprehensive design environment, such as [146], often includes graphical design tools, a floorplanner, a schematic generator, and a simulation interface. This type of environment can be used to evaluate performance tradeoffs in SP applications related to data sample rate, pipelining, and bit width [147]. As blocks are added to or removed from the system, estimates are dynamically updated. Simply measuring the performance and area of the functional units is generally not sufficient in design exploration. Interconnect speed, required data precision, and power consumption must also be estimated [148] to provide accurate results. Contemporary design environments have moved beyond simple estimates of performance, area, and power based on stored parameters for library components. A regression analysis of tradeoffs can be performed [149] that considers different combinations of functional units and interconnect. This approach can determine the optimal high-level structure of an application for a given target hardware platform without the need for time-consuming synthesis and physical design.

Today's commercial CAD packages, such as Xilinx Vivado and Altera Qsys, contain an integrated series of tools, similar to the systems described above. These tools allow for design expression, simulation, power analysis, compilation, and in-circuit testing, often using on-chip JTAG interfaces.

B. Debugging Tools

Since FPGAs generally have high logic capacity and often are difficult to analyze once they have been compiled to LUTs and flip flops, the need for effective test and debug tools are critical for application development. In many cases, debugging can be performed at a high level via simulation, but in cases where it is difficult to isolate specific bugs, analysis is needed at the hardware level after a bitstream has been loaded into the FPGA device. Most debug strategies combine a graphical waveform viewer, a simulator, and on-chip circuitry which can dynamically collect debug information from an executing design. Similar to software debuggers, a source-level debugger can be used to correlate circuit-level debug information with the original behavioral-level source code and allow for hardware breakpoints, watchpoints, and single-stepping [150]. In some cases, a per-application customized debug controller is instrumented on-chip [151] to allow for rapid debug response. Users can interact with the FPGA under test via an application programming interface (API). This approach has recently been extended [152] to include an entire debug infrastructure on chip (controller, trace buffer, and overlay network for collecting debug data).

An alternative approach to dynamically updating circuitry on-chip for debugging purposes involves direct modification of the design bitstream [153]. Small changes to features in previously compiled and implemented logic cores (e.g., constant values, pipeline registers) can be made via bitstream manipulation, providing specialization while avoiding long compilation, place, and route times. Over the years, many debug features have been incorporated into Xilinx and Altera products. The availability of Xilinx ChipScope and Altera SignalTap allows for run-time downloading and analysis of FPGA design values.

C. Hardware/Software Co-Design

Hardware/software co-design is a broad field which addresses the identification and management of application portions that are assigned to microprocessor, DSP, and custom hardware resources. Typically, the most challenging part of hardware/software co-design is the high-level area, performance, and energy estimations of the portions that are assigned to hardware. Determining which computation should be migrated to logic in such systems depends on many factors including the logic capacity of the target reconfigurable hardware, the amount of its attached memory, and the speed of the communication interface to the microprocessor. This challenge can be addressed by precompiling hardware and software components of common functions and storing the bitstreams and component statistics in a database [154]. The hArtes co-design system [155] targets microprocessors, FPGAs and digital signal processors. All hardware/software decisions are made at compile time, with designer input. A comprehensive back-end system individually targeting all three target technologies produces accurate performance estimates which can be used for additional partitioning iterations.

FPGAs provide the additional opportunity to support the dynamic migration of tasks from software to hardware during execution, if run-time conditions warrant the exchange. A self-contained approach [156] implemented the decision-making hardware/software partitioner on the FPGA and dynamically created and instantiated hardware circuits as needed. The amount of time required for synthesis and physical design limited the size of the identified tasks to circuits with a small number of LUTs. FPGAs have also been instrumented with logic that can very accurately determine the cycle counts of soft processors implemented in FPGA logic [157] at run-time. This information can be used to dynamically evaluate which tasks should be implemented in hardware. Although significant research has taken place in developing hardware/software co-design techniques over the past 25 years, most FPGA designers still manually select the computation which will be mapped to reconfigurable resources in heterogeneous systems. As designers move to more behavioral-level specification for designs, the situation may change.

D. Tools for Precision Analysis

Unlike fixed microprocessors, FPGAs offer the flexibility of implementing arithmetic hardware with bitwidths that exactly match the desired precision of the application. Over the years, many papers have examined automated techniques to determine appropriate operand word lengths and intermediate compute value bitwidths to maintain a user-specified level of accuracy. These optimizations generally take place in the context of implementation of digital signal processing applications. The design time use of precision analysis [158] considers the selection of intermediate computation bitwidths in integer-based FPGA designs to maintain a desired precision. The effect of roundoff errors in intermediate bit-limited computations is also a consideration [159]. Since FPGAs can be used to implement both fixed- and floating-point arithmetic, bitwidth tradeoffs for both types of implementations can be simultaneously considered [160] even if deviations from standard floating-point formats are required. In some cases, it makes sense to design bitwdiths to handle the precision of common-case rather than worst-case computations. If an operation requires precision that is greater than can be implemented in the hardware, it can be migrated to software [161].

E. Fast Application Mapping

A limiting issue for RC based on FPGAs has been the long compile times of the FPGA designs (often more than an hour per device). For small amounts of logic, a simple horizontal alignment of LUTs and a greedy routing of wiring can be performed at run-time on the FPGA [156]. An underlying theme of most fast application mapping

approaches is the use of the design hierarchy and regularity in performing FPGA synthesis, placement and routing. Design macroblocks can be aligned in a single dimension, accelerating subsequent routing [162]. Newer, routing-plentiful FPGAs can use prerouted macroblocks [163] so that only interblock routes are needed, albeit at a cost of design clock cycle performance. Several open-source FPGA tool flows [164], [165] are available which allow designers to create their own customized tools to accelerate FPGA design compilation, including bitstream manipulation.

F. Tools for Specialization

A significant benefit of FPGAs is the ability of the designer to specialize a specific FPGA circuit. For example, the multiplier hardware for a finite impulse response filter can be reduced if the filter coefficients are known and can be synthesized into the multiplier logic, reducing multiplier area and power consumption. Tool sets have been developed to assist specialization for applications such as text search [166], logic simulation [167], and automatic target recognition [168], among others. These tools use compile-time information about application data sets to optimize functional units and memory accesses in the generated hardware. Although today's powerful FPGA logic synthesis tools take advantage of data-dependent specialization when constants are specified by the user at compile time, standard interfaces for specializing computation at run time have not yet been developed.

G. Tools for Coarse-Grained Architectures

Even though coarse-grained architectures (Section II-C5) have more function-constrained compute blocks than LUT-based FPGAs, sophisticated tools are still needed for application mapping. Basic mapping operations include the scheduling of computation to coarse-grained compute blocks (typically ALUs) and placement and routing. These operations for 1D datapaths were shown to be linear time in terms of number of block computations [58]. For 2-D coarse-grained architectures, the placement of computation on blocks and the routing of interconnect is more of a challenge. The DRESC compiler [169] performed scheduling, placement, and routing for an application as part of a single simulated annealing search, a time-consuming approach for even small numbers of compute blocks. If the compute problem is constrained to computation with repetitive communication (e.g., unrolled loops accessing memory), computation, interconnect use, and memory accesses can be scheduled simultaneously following placement [170]. The SPR compiler [171] takes mapping a step further by performing each step independently. After computation is placed in blocks and inter-block connections are routed, both compute blocks and interconnect are scheduled at the cycle level. The availability of a pipelined and time-switched interconnect between neighboring blocks allows for this level of scheduling control.

H. Looking Forward

Although RC tools have improved dramatically, significant work remains in several areas. The movement of programming environments to the behavioral level provides ample opportunity for improved estimation of low-level resource use without the need for complete design synthesis. Compilation time is still a source of pain, and in some cases it may make sense to forgo full device resource usage to dramatically reduce compile times. The recent strong interest in GPGPUs makes it likely that an intermediate architectural point between these thread-parallel devices and the fine-grained parallelism, specialization, and reconfigurability of FPGAs will be developed.

A fundamental approach to energy-efficient design is shutting down resources that are not currently in use. Although contemporary FPGAs do not currently support dynamic intra-device region shutdown, future devices that do include this feature may provide a path for increased energy efficiency. The use of heterogeneity will require additional tools beyond the current state of the art. At the same time, energy efficiency is driving increased interest in heterogeneous accelerator architectures. Such heterogeneity will provide new challenges regarding hardware/software co-design and identifying which compute functions should be mapped to which resources.

V. RUN-TIME RECONFIGURATION

SRAM-configured FPGAs can change their functionality during operation. To distinguish this use of reconfiguration from the cases where the configuration remains constant during an application, the former case has been termed run-time reconfiguration (RTR). RTR fully exploits the hybrid nature of FPGAs, allowing for hardware organization changes as needed during different phases of the computation. As a result, RTR creates a distinct requirement for application mapping to coordinate these changes. The potential benefit of the approach is the specialization of the computation to the near-instantaneous needs of the application, reducing the size and energy required for the design. These benefits must be weighed against the additional space required to hold extra configuration information and the time and energy needed to transfer the information.

The broad usefulness and practicality of RTR remains an open question. For some applications, it is possible to gain the advantage of customization without the need for run-time logic modification simply by reprogramming data memories or stopping computation and statically loading one of multiple precomputed configurations. Additionally, as described in this section, the use of RTR requires additional design compilation steps and run-time execution management. Applications that are well-suited for

RTR often require clearly defined blocks of computation that can be dynamically modified during execution.

A. Run-Time Reconfiguration Design Tools

A number of tools have been created to help designers identify and extract portions of applications that are amenable to RTR. An initial effort [172] identified portions of an artificial neural network that are subject to run-time update. These changes were limited to multiplier bitwidth modifications and constant value updates. Since there may be overlap in the logic used by several different versions of the reconfigurable circuitry, configuration sharing may be possible. Software can be used to identify the overlap and minimize the amount of circuitry that must be loaded into the FPGA during reconfiguration [173]. Rather than relying on tools to extract and create versions of circuitry that can be reconfigured, it is often easier for designers to predefine the circuits with the assistance of tools. Reconfigurable circuitry can be compiled into placed and routed blocks [174] after designer-identified regions are extracted from the netlist and evaluated via simulation [175].

In general, FPGA architectures constrain the interface between static and dynamic regions (e.g., both Altera Stratix V and Xilinx Virtex devices require that this interface be made through fixed-location LUTs). Tools can be used to assist in the floorplanning of modules [176] that are then synthesized and stored in a module library [177]. A benefit of this approach is the creation of standard interfaces that make the modules physically interchangeable inside the FPGA [178]. This type of module creation is particularly beneficial for application-specific instructions implemented in reconfigurable logic to extend a microprocessor's instruction set [179]. These application-specific instructions can be swapped into the reconfigurable fabric on a per-application basis. If numerous candidate application-specific instructions exist, a selection can be made based on frequency of instruction use and overall expected application performance improvement [180].

Since RTR is often applied to stream-oriented applications with predictable computation and communication demands, tools can be used to determine when hardware swapping should take place. A comprehensive design cycle includes simulation of the computation, inter-module communication, and reconfiguration [181]. Scheduling algorithms are often used to determine the optimal sequence of reconfiguration under expected data loads. This scheduling can be expanded [182] to consider both configuration swapping and functional unit sharing to reduce the amount of required configurations. For devices with multiple, independent configuration contexts, the ordered swapping of entire tasks can be considered in an effort to reduce overall application latency [183].

FPGA companies have been slow to develop easy-to-use tools and interfaces that could aid an FPGA designer's use of RTR on a per-application basis. Recent Virtex devices do support a configuration interface (ICAP) [174] that is writable from inside the device. To some extent, the limited RTR support to date may be a result of a perceived lack of consumer demand and the slow identification of applications that could benefit from the approach. The application space that can benefit from RTR has recently grown to the point that both Altera and Xilinx are producing devices with partial RTR capabilities.

B. Operating System Support

The need to swap specialized reconfigurable circuits at run time requires software management beyond what is required for a static hardware implementation. Such management must consider the size of the reconfigurable resources and the likely performance benefit and overheads of performing run-time circuit swapping. Typically, software components must be added to the operating system of a companion microprocessor to dynamically manage available hardware resources. As interest in RTR has grown, a number of operating system components that attempt to satisfy this need have been developed. Virtual hardware managers that allowed for the scheduled run-time swapping of preplaced macro blocks in an FPGA provided a first step [184], [185].

Operating system support for dynamically placing portions of an application in FPGA logic appeared later [186]. The FPGA was viewed as a co-processor in a microprocessor-based system and its configuration was scheduled considering multiple software threads of execution. This type of dynamic scheduling is particularly difficult for embedded systems that have strict real-time constraints [187]. Task scheduling for these systems can also consider the assignment of tasks to either hardware or software based on available resources and latency requirements [188]. Operating system components for RC can also serve other functions beyond RTR. For example, Linux has been enhanced to allow for new FPGA-based hardware modules to be available to the thread scheduler [189] and to have access to the Linux file system [190]. Recently, operating systems for platforms that include one or more microprocessors and reconfigurable resources have gained in popularity (e.g., MicroC/OS), although commercial operating system support for RTR management is very limited.

C. RTR Applications

A number of applications have not only shown the benefits of using dynamic reconfiguration, but have also demonstrated novel techniques for implementing it. Many computations, like image processing, go through distinct phases. A device that implements this type of computation can be reconfigured between phases to fit a large task onto a small FPGA [191]. Another approach [192] demonstrated that constant search values could be dynamically updated to accelerate the search of the human genome. Data sorting was accelerated by dynamically updating the sorting algorithm for large data sets [193]. For a

communication system, the parameters and circuitry of an adaptive Viterbi decoder were modified on the fly to adapt to changes in channel noise [194]. These updates improved performance while maintaining a consistent data transfer quality level.

The reuse of modules for applications can be supported by developing interfaces that allow for straightforward inter-module interconnect. This approach was demonstrated for networking applications for protocol and packet filtering modules in the context of virtual networking [195]. Similarly, entire modules used for SQL database query operations were dynamically configured in milliseconds to enhance application performance [196].

After many years of slow growth in the research community, FPGA RTR is gaining traction in the commercial networking and telecommunications domains where hardware configurations must change to match protocol needs. In these applications, portions of the design must stay active while reconfiguration takes place, encouraging revitalized vendor support for partial FPGA reconfiguration.

D. Looking Forward

For RTR to gain in popularity, standard models for describing or automatically extracting opportunities for RTR at higher levels of application design must be developed. Advanced tools that can identify, synthesize, and deploy similarly sized design regions of an application for use in RTR will be needed as the RTR application space expands. Modifications in device architecture to allow easier access and interpretation of device configuration information are likely to positively influence this effort. The use of heterogeneity will also push the need for more complex operating systems to manage both resource mapping and accompanying run-time reconfiguration.

VI. APPLICATIONS

The pioneers showed that RC was good for something (Section II-A). Accumulated experience has generalized and broadened this understanding. Application-focused research shows us what can be done and what kinds of benefits reconfigurable architectures offer. This characterizes the space to help people understand what classes of applications and what application characteristics are likely to benefit from RC. They also show us *how* to solve problems on RC. Often the best formulation and algorithm for a reconfigurable architecture are very different from counterparts for a processor architecture. Many of these solutions point the way to exploiting the massive parallelism that Moore's Law scaling continues to make available to us.

A. Signal and Image Processing

Signal and image processing tasks require high throughput, regular computations. The same operations are performed repeatedly on sample inputs, usually with simple communication structure. As a result, they are amenable to high speed, pipelined implementations on FPGAs, exploiting operation parallelism and nearest-neighbor communication (Section III-C). The ability to place large numbers of, perhaps specialized, multipliers on the FPGA allows them to outperform sequential DSPs and achieve real-time performance levels that, at one time, were only possible with dedicated ASICs, particularly for the key finite-impulse response (FIR) filter kernel [197]. FPGAs can also provide high throughput on the key fast Fourier transforms (FFT) kernel [198] and matrix-matrix multiply [199]. Many basic SP operations can be implemented efficiently on FPGAs using CORDIC algorithms [200]. The ability to exploit local, spatial computations also allows reconfigurable architectures to support energy-efficient, real-time signal processing for wireless communications [201].

The high throughput that FPGAs could deliver on regular applications allowed them to achieve real-time performance on image processing applications [202]. They have been applied to image compression, including hyperspectral imaging [203], wavelet compression [204], and adaptive decoding [205] and real-time vision, including face detection [206], stereopsis [207], and background identification for object tracking [208].

B. Financial

Calculating the price of financial instruments is also a very regular, compute intensive task. There is a direct monetary advantage to producing more accurate estimates faster than competitors. There is also a premium for speed and density (both ops/m^3 and ops/W) so the calculations can be done in server rooms close to financial trading hubs. Derivative pricing uses Monte Carlo simulations to estimate risk, and these simulations are effectively accelerated with RC [209]. RC is also useful for credit risk management [210] and option pricing [211].

C. Security

Bit-level manipulation is supported efficiently on FPGAs, and bulk encryption is often amenable to streaming data through a highly pipelined datapath. Because of drastically higher performance versus microprocessor implementations, encryption and decryption have often been used to demonstrate the potential benefits of FPGA accelerators [13], [58].

1) Private Key Encryption/Decryption Acceleration: Early papers displayed the benefits of reconfigurable implementations of the Data Encryption Standard (DES) [212]. Exploiting FPGA reconfigurability, it is possible to specialize the DES circuit around a particular key schedule to reduce area and increase throughput [213]. Once AES was standardized to replace DES, 7 Gb/s encryption throughput with an FPGA was demonstrated [214]. It was

later shown [215] that dedicated DSP and memory blocks in modern FPGAs could be exploited to achieve AES encryption throughput over 55 Gb/s on a Virtex 5. Others have shown high performance on International Data Encryption Algorithm (IDEA) encryption [15], [216].

2) Public Key: FPGAs have also been heavily used for public key encryption. FPGA co-processors have been used to accelerate modular multiplication and RSA encryption and decryption [217], [218]. The implementation of Galois Field multipliers for elliptic curve cryptography has also been explored [219]. Compact microcoded [220] and pipelined and specialized [221] elliptic curve cryptography (ECC) implementations appeared subsequently.

3) Breaking Encryption: The high throughput of FPGAs on regular tasks makes them efficient at the kinds of brute force search necessary to attack encryption. FPGAs were used for brute force key search for Rivest Cipher 4 (RC4) [222], as used in the secure sockets layer (SSL), secure shell (SSH), and wired equivalent privacy (WEP). The paper showed that a single 2002-era FPGA could recover a 40b key in 50 hours and that the approach could be trivially parallelized onto multiple FPGAs. Highly efficient FPGA hardware implementations of sieve operations that factor large numbers were developed [223]. Subsequent work attacking ECC [224] showed that 160b ECC keys were probably still safe, but 112b ECC keys were potentially vulnerable.

The *Workshop on Cryptographic Hardware and Embedded Systems* collects a wealth of additional work on implementing cryptographic functions on FPGAs.

D. NP-Hard Optimization Problems

Optimization problems are a large class of applications that require sizable amounts of computation. This is particularly true when the optimization is NP-hard. For these problems, the ability to evaluate options in heavily pipelined, parallel computations makes them attractive candidates for FPGAs. Furthermore, the problem statement and state for optimization problems is often small compared to the computation required, so they avoid memory or data transfer bottlenecks.

Boolean satisfiability (SAT) is the canonical NP-complete problem. Both because of its own importance and the fact that other NP-hard problems can be reduced to SAT, it is an important application target. It is particularly attractive because the evaluation of formulas can be cast directly as a combinational circuit that can be efficiently evaluated on an FPGA. Consequently, there is considerable parallelism and bit-level mapping efficiency to be exploited even before exploring pipelined, parallel evaluation [225]. Three orders of magnitude speedup were demonstrated on many problem instances [226]. The time to place-and-route the instance-specific design for the FPGAs presented a potential bottleneck that might undermine the performance gains. By partitioning the design into a fixed portion and a variable portion that is customized for the specific SAT expression under examination, it is possible to avoid this bottleneck [227]. In these early designs, the size of the SAT instance that an RC could attack was limited by the size of the platform. Later work [228] showed how to decompose large SAT problems into smaller ones that could be sequenced on the limited FPGA resources. The kind of pruning and state-enumeration search used in efficient SAT was extended [229] to solve the problem of explicit-state model checking.

Outside of SAT, several other NP-hard problems have been directly attacked. FPGA-accelerated genetic algorithms were used to solve Traveling Salesman Problem (TSP) instances [230], and a 2-approximate solution to the Euclidean TSP was found [231] that was six-fold faster than the best known software solution at the time for small problem instances. Set covering, specifically as it shows up in logic synthesis, was also addressed [232].

E. Pattern Matching

Pattern matching tasks, where it is necessary to identify specific patterns within a large data set, require a considerable amount of computation, but the computational tasks are generally very regular, exploiting the natural strengths of FPGAs. Matching applications that support target recognition within images, packet filtering in network data streams (Section VI-F2), and biological sequence identification in large genomic databases have all proven to be rich application classes for RC.

1) Automatic Target Recognition: ATR uses image matching to locate and identify specific target objects. The ATR task demands a large amount of regular, bit-level computation that could not be performed in real-time on the leading microprocessors of the mid-1990s. Typical ATR implementations search for any of a large set of shapes within an image. The FPGA's ability to rapidly modify its gate-level logic through fast RTR was exploited to support ATR [168]. The FPGA could be specialized to allow a search through a series of match templates. New templates could be dynamically loaded into the FPGA as needed. As a result, the approach was able to achieve real-time matching on hundreds of 16×16 pixel target images. The large compile time necessary to generate unique bitstreams for a set of image operations was avoided by using a generic architecture and specializing only the LUT ROMs to the particular operation [233].

2) Bioinformatics: Deoxyribonucleic acid (DNA) sequence matching on FPGA accelerators has been of great interest since Splash [5]. A classical, dynamic programming (DP) approach to the problem of homologous series detection (the matching of a target DNA sequence to a

reference DNA sequence in a database) outperformed conventional computers by several orders of magnitude [234]. The DP approach was extended [192] to match against the then-new Human Genome Database (3 billion base-pairs) in real-time using specialized configurations and RTR.

By the mid-2000s it was apparent that the GenBank database of known DNA sequences was doubling in size every 18 months and that conventional DP approaches were too slow. The Basic Local Alignment Search Tool (BLAST) arose as a new heuristic approach that used approximate string matching to identify homologous series. This approach became the de facto standard for sequence matching. An RC hardware implementation of both BLAST and the DP algorithm that scanned the reference genomic data in a single pass was developed [235]. Off-chip SRAMs can be used to process all stages of the BLAST algorithm on a single FPGA even when larger sequence queries are handled [236].

Recently, the focus in bioinformatics has shifted, driven by the rise of high-throughput next generation DNA sequencers that generate millions of DNA fragments a day, each of which contain up to several hundred base-pairs ("short-read" sections). These sequencers are starting to produce personal DNA genomes at a low cost (thousands of dollars per genome) that identify genetic-based disease markers and abnormalities for a given individual. These short-read sequences are aligned against a reference DNA sequence to get the best (and fastest) alignment and to identify potential mutations (additions, deletions and/or substitutions of base-pairs) in the target DNA being analyzed. A combination of software and reconfigurable hardware was used to achieve a two order of magnitude speedup over a software-only solution [237]. Subsequent work [238] tackled the problem of matching millions of short read sequences against a reference DNA sequence through the hardware-based acceleration of the popular BLAT-like Fast Accurate Search algorithm (BFAST) software algorithm. The BLAST-Like Alignment Tool (BLAT) is, itself, a variant of BLAST. The use of RNA folding was also examined [239].

The rapid advances in DNA sequencers and the resultant generation of terabytes of genomic data have set the stage for many more RCs aimed at specific bioinformatics problems.

F. Networking

Networking equipment must handle packet processing at high and growing throughputs and adapt to rapidly evolving standards. Reconfigurable hardware provides the required high throughput on an adaptable, programmable media.

1) Routers: FPGAs provide the bandwidth to serve as line cards, in-line packet processing engines, switches, and routers in modern networking systems. The Field-Programmable Port Extender interposes between a line card and a switch to provide flow-specific programmable packet processing [240] and is able to monitor 8 million TCP flows at 2.5 Gbps [241]. NetFPGA provides a standard, open platform that supports gigabit networking for experimenting with network services and network switch and router implementations [242], including an early implementation of the OpenFlow [243] architecture for software-defined networking. To make it easier to design and specify novel networking designs, a domain-specific language framework [244] was developed for composing CLICK modular router primitives into a new reconfigurable hardware design. These designs could be compiled to FPGAs achieving comparable performance to contemporary, dedicated network processors [245]. A 16×16 switch that supports 10 Gbps link speeds was built in a Virtex-6 FPGA [246]. For more on the use of reconfigurable architectures for software-defined networking, see the paper on this topic in this issue [247].

2) Network Intrusion Detection Systems: One specific use of FPGAs for in-line packet processing is packet filtering and network intrusion detection. Viruses, worms, Trojans, and zombie bots use the network to attack host machines. Network intrusion detection systems (NIDS) use regular expressions to identify malicious packets and prevent them from being delivered to host computers. A technique for compactly implementing regular expression nondeterministic finite automata (NFAs) on FPGAs was introduced in [248]. This regular expression matching was applied to the SNORT NIDS rule-set database to implement hundreds of state machines on a single FPGA [249]. The resulting acceleration engine ran at over 600 times the speed of SNORT in software. The rule encodings were compressed to pack more simultaneous rules onto an FPGA while sustaining a filtering rate of over 3 Gbps [250]. Earlier work was furthered with a dual reconfigurable processor approach [251]; one processor optimized the rules database in real-time and the other scanned each Internet packet against it. A software-based front-end rule processor was developed to support all the features found in the SNORT rules while handling 10 Gbps data rates [252].

G. Numerical and Scientific Computing

With limited gate count, the earliest FPGAs could not support interesting floating-point (FP) datapaths. However, as FPGA capacity grew due to Moore's Law scaling, that soon changed. With the advent of modern, large gate-equivalent FPGAs and FP libraries [253], FP arithmetic on an FPGA has become much easier and more practical to use. It cannot challenge the peak capacity of GPGPUs, which contain hundreds of FP-capable

processors, but FPGAs offer more raw floating-point throughput than microprocessors [254] and the ability to customize the datapath to the computation allows them to use more of the peak capacity than processors in many applications. As an example, it was shown [255] that FP FPGA arithmetic could surpass the FP performance of a conventional PC microprocessor on memory-bound, double-precision (DP) FP dense matrix operations, vector dot product, matrix-vector multiply, and matrix-matrix multiply operations. This research sparked a flurry of DP FP research on FPGAs and was extended to DP FP FFT [256].

In practice, many important matrices are sparse, meaning most of their entries are zeros. Exploiting this sparsity means there are fewer entries to store and read from memory and fewer multiplications to perform. Since the locations of the nonzeros can be irregular, this often makes the problem bound by the latency of memory performance rather than by the raw floating-point throughput. On these tasks, microprocessors are unable to deliver near their peak performance. The high-bandwidth and low latency of FPGA on-chip memories can be exploited to accelerate sparse matrix-vector multiplication [257], and the memory can be further optimized for symmetrical and banded matrices [258]. Higher accuracy and sequential accumulation performance can be achieved by avoiding normalizing shifts inside the accumulator [259]. For larger, irregular matrices that cannot fit in on-chip memory, efficient reduce circuits can be used [260] to compute sparse matrix-vector multiplication at the speed that the data can be retrieved from off-chip DRAM.

At a higher level, FPGA FP has been used for solvers. Conjugate gradient uses iterative matrix-vector multiplication to solve systems of linear equations including dense matrix problems [261] and sparse problems [262]. Mixed precisions can be exploited to efficiently solve conjugate gradient problems [263]. Other FPGA implementations solved for the eigenvalues of a matrix [264] and provided a direct solve of a system of linear equations using LU factorization [265]. The acceleration of a SPICE circuit simulator, which solves nonlinear device equations using a model evaluation for small signal linearization and a direct linear solver [266], has also been demonstrated.

H. Molecular Dynamics

The three-dimensional (3-D) modeling of physical phenomena on a discrete, particle-by-particle basis is another great challenge with an insatiable need for computation. One particular area of attention has been molecular dynamics (MD) simulations, where a large number of particles (thousands) are modeled at the atomic level using Newtonian mechanics. The forces on each particle are summed and then integrated using the classical equations of motion. Among the many applications of this technique is the modeling of biological molecules, including protein folding. It was demonstrated [267] that an FPGA clocked at 100 MHz had a 20-fold performance advantage over a contemporary GHz microprocessor. The design space of force computation pipelines and particle-particle filtering was explored in [268]. The simulation of molecular dynamics will continue to grow as more focus is placed on protein folding and the more

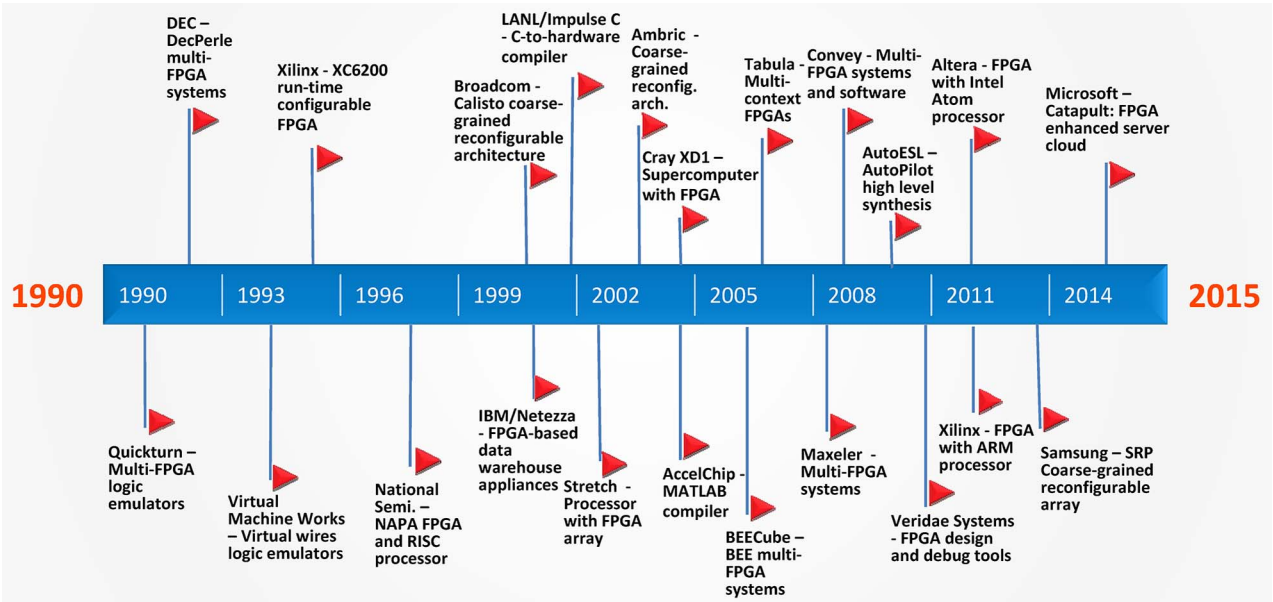

Fig. 1. *Timeline of a selection of commercial efforts with significant RC impact.*

general goal of understanding biological processes at the molecular level.

VII. CONCLUSION

RC has become a viable alternative to ASICs and fixed microprocessors in our computing systems. With increasing energy and reliability concerns, there is every reason to believe its importance will grow as technology scales. As we move to larger chips with heterogeneous fabrics and accelerators, reconfigurability will continue to merge into the mainstream computing infrastructure.

Over the past 25 years, the RC community has pioneered accelerator architectures, hardware specialization, and RTR models and support. A number of commercial RC efforts, noted in this manuscript, have appeared over this time frame (Fig. 1). The field has accumulated ample evidence of the superior performance and energy efficiency of FPGAs compared to processors over a broad range of applications. It has made significant strides in the capture of applications, C-to-gates compilation, and streaming models and architectures. With the possible exception of RTR, all of these contributions are central to today's mainstream computing trends.

There is still much to learn and a long way to go before the field begins to mature. Ease-of-use, ease-of-debug, accessibility, and a slow edit-compile-debug cycle remain challenges that must be addressed to achieve broader appeal. Hybrid architectures and RTR, while attractive, need better abstraction models and support. Demonstrations and patterns of effective use in large-scale, complex applications are still needed, as is a better understanding of the relative strengths and weaknesses of RC. Innovations in all areas of RC are reported regularly at the leading conferences (*ACM/SIGDA International Symposium on FPGAs* (FPGA), *IEEE International Symposium on Field-Programmable Custom Computing Machines* (FCCM), *International Conference on Field Programmable Technology* (ICFPT), and *International Conference on Field-Programmable Logic and Applications* (FPL)) and in the leading journal (*ACM Transactions on Reconfigurable Technology and Systems*) in the field. ■

REFERENCES

[1] W. S. Carter *et al.*, "A user programmable reconfigurable logic array," in *CICC*, May 1986, pp. 233–235.

[2] J. Varghese, M. Butts, and J. Batcheller, "An efficient logic emulation system," *IEEE Trans. VLSI Syst.*, vol. 1, no. 2, pp. 171–174, Jun. 1993.

[3] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, Apr. 2000.

[4] J. Rabaey, "Reconfigurable computing: The solution to low power programmable DSP," in *Proc. ICASSP*, Apr. 1997, pp. 275–278.

[5] M. Gokhale *et al.*, "Building and using a highly programmable logic array," *Computer*, vol. 24, no. 1, pp. 81–89, Jan. 1991.

[6] J. E. Vuillemin *et al.*, "Programmable active memories: Reconfigurable systems come of age," *IEEE Trans. VLSI Syst.*, vol. 4, no. 1, pp. 56–69, Mar. 1996.

[7] P. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, no. 3, pp. 11–18, Mar. 1993.

[8] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. Int. Symp. Microarch.*, Nov. 1994, pp. 172–180.

[9] A. DeHon, "DPGA-coupled microprocessors: Commodity ICs for the early 21st century," in *Proc. FCCM*, Apr. 1994, pp. 31–39.

[10] C. Rupp *et al.*, "The NAPA adaptive processing architecture," in *Proc. FCCM*, Apr. 1998, pp. 28–37.

[11] R. D. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *Proc. FCCM*, Apr. 1996, pp. 126–135.

[12] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera reconfigurable functional unit," *IEEE Trans. VLSI Syst.*, vol. 12, no. 2, pp. 206–217, Feb. 2004.

[13] T. Callahan, J. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000.

[14] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proc. FPL*, 2003, pp. 61–70.

[15] M. Vuletic, L. Pozzi, and P. Ienne, "Virtual memory window for a portable reconfigurable cryptography coprocessor," in *Proc. FCCM*, Apr. 2004, pp. 24–33.

[16] J. A. Jacob and P. Chow, "Memory interfacing and instruction specification for reconfigurable processors," in *Proc. FPGA*, Feb. 1999, pp. 145–154.

[17] G. Stitt, B. Grattan, J. Villarreal, and F. Vahid, "Using on-chip configurable logic to reduce embedded system software energy," in *Proc. FCCM*, 2002, pp. 143–151.

[18] J. Williams *et al.*, "Characterization of fixed and reconfigurable multi-core devices for application acceleration," *ACM Trans. Reconfig. Tech. Syst.*, vol. 3, no. 4, pp. 19:1–19:29, Nov. 2010.

[19] H. Schmit, B. Levine, and B. Ylvisaker, "Queue machines: Hardware compilation in hardware," in *Proc. FCCM*, 2002, pp. 152–160.

[20] J. M. Arnold, "S5: The architecture and development flow of a software configurable processor," in *Proc. ICFPT*, Dec. 2005, pp. 125–128.

[21] F. Raimbault, D. Lavenier, S. Rubini, and B. Pottier, "Fine grain parallelism on a MIMD machine using FPGAs," in *Proc. FCCM*, Apr. 1993, pp. 2–8.

[22] P. Dhaussy, J.-M. Filloque, B. Pottier, and S. Rubini, "Global control synthesis for an MIMD/FPGA machine," in *Proc. FCCM*, Apr. 1994, pp. 72–81.

[23] S. Trimberger, "Three ages of FPGAs," *Proc. IEEE*, vol. 103, no. 3, pp. 318–331, Mar. 2015.

[24] A. DeHon, "Fundamental underpinnings of reconfigurable computing architectures," *Proc. IEEE*, vol. 103, no. 3, pp. 355–378, Mar. 2015.

[25] T. Kean and J. Gray, "Configurable hardware: Two case studies of micro-grain computation," *J. VLSI Signal Proc. Syst. for Signals, Image Video Tech.*, vol. 2, no. 1, pp. 9–16, 1990.

[26] G. Milne, P. Cockshott, G. McCaskill, and P. Barrie, "Realising massively concurrent systems on the space machine," in *Proc. FCCM*, Apr. 1993, pp. 26–32.

[27] N. Margolus, "An FPGA architecture for DRAM-based systolic computations," in *Proc. FCCM*, 1997, pp. 2–11.

[28] J. Durbano and F. Ortiz, "FPGA-based acceleration of the 3D finite-difference time-domain method," in *Proc. FCCM*, Apr. 2004, pp. 156–163.

[29] T. Kobori, T. Maruyama, and T. Hoshino, "A cellular automata system with FPGA," in *Proc. FCCM*, Mar. 2001, pp. 120–129.

[30] R. Hartenstein, R. Kress, and H. Reinig, "A reconfigurable data-driven ALU for Xputers," in *Proc. FCCM*, Apr. 1994, pp. 139–146.

[31] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs," in *Proc. ICFPT*, Dec. 2009, pp. 190–198.

[32] C. Iseli and E. Sanchez, "Spyder: A reconfigurable VLIW processor using FPGAs," in *Proc. FCCM*, Apr. 1993, pp. 17–24.

[33] S. Wong, T. van As, and G. Brown, "p-VEX: A reconfigurable and extensible softcore VLIW processor," in *Proc. ICFPT*, Dec. 2008, pp. 369–372.

[34] M. Weinhardt and W. Luk, "Pipeline vectorization for reconfigurable systems," in *Proc. FCCM*, 1999, pp. 52–62.

[35] J. Yu, C. Eagleston, C. H.-Y. Chou, M. Perreault, and G. Lemieux, "Vector processing as a soft processor accelerator,"

ACM Trans. Reconfig. Tech. Syst., vol. 2, no. 2, pp. 12:1–12:34, Jun. 2009.

[36] A. Severance, J. Edwards, H. Omidian, and G. Lemieux, "Soft vector processors with streaming pipelines," in *Proc. FPGA*, 2014, pp. 117–126.

[37] J. Babb *et al.*, "Parallelizing applications into silicon," in *Proc. FCCM*, 1999, pp. 70–80.

[38] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers, "CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures," in *Proc. FPL*, Sep. 2008, pp. 173–178.

[39] I. Lebedev *et al.*, "Exploring many-core design templates for FPGAs and ASICs," *Int. J. Reconf. Comp.*, 2012, Art. ID. 439141.

[40] P. J. Bakkes, J. J. duPlessis, and B. L. Hutchings, "Mixing fixed and reconfigurable logic for array processing," in *Proc. FCCM*, 2006, pp. 118–125.

[41] C. H. Ho, P. H. W. Leong, W. Luk, S. J. E. Wilton, and S. López-Buedo, "Virtual embedded blocks: A methodology for evaluating embedded elements in FPGAs," in *Proc. FCCM*, 2006, pp. 35–44.

[42] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[43] J. Luu *et al.*, "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling," *ACM Trans. Reconfig. Tech. Syst.*, vol. 4, no. 4, pp. 32:1–32:23, Dec. 2011.

[44] W. Tsu *et al.*, "HSRA: High-speed, hierarchical synchronous reconfigurable array," in *Proc. FPGA*, Feb. 1999, pp. 125–134.

[45] N. Weaver, J. Hauser, and J. Wawrzynek, "The SFRA: A corner-turn FPGA architecture," in *Proc. FPGA*, Mar. 2004, pp. 3–12.

[46] S. Hauck, S. Burns, G. Borriello, and C. Ebeling, "An FPGA for implementing asynchronous circuits," *IEEE Des. Test. Comput.*, vol. 11, no. 3, pp. 60–69, Jul. 1994.

[47] J. Teifel and R. Manohar, "Highly pipelined asynchronous FPGAs," in *Proc. FPGA*, 2004, pp. 133–142.

[48] C. G. Wong, A. J. Martin, and P. Thomas, "An architecture for asynchronous FPGAs," in *Proc. ICFPT*, Dec. 2003, pp. 170–177.

[49] A. DeHon, "DPGA utilization and application," in *Proc. FPGA*, Feb. 1996, pp. 115–121.

[50] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *Proc. FCCM*, Apr. 1997, pp. 22–28.

[51] T. R. Halfhill, "Tabula's time machine," *Microprocessor Rep.*, Mar. 29, 2010.

[52] D. C. Chen and J. M. Rabaey, "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths," *IEEE J. Solid-State Circ.*, vol. 27, no. 12, pp. 1895–1904, Dec. 1992.

[53] T. Miyamori and K. Olukotun, "A quantitative analysis of reconfigurable coprocessors for multimedia applications," in *Proc. FCCM*, Apr. 1998, pp. 2–11.

[54] F.-J. Veredas, M. Scheppler, W. Moffat, and B. Mei, "Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes," in *Proc. FPL*, 2005, pp. 106–111.

[55] M. Suzuki *et al.*, "Stream applications on the dynamically reconfigurable processor," in *Proc. ICFPT*, Dec. 2004, pp. 137–144.

[56] R. W. Hartenstein, R. Kress, and H. Reinig, "A new FPGA architecture for word-oriented datapaths," in *Proc. FPL*, Sep. 1994, pp. 144–155.

[57] C. Ebeling, D. Cronquist, and P. Franklin, "RaPiD—Reconfigurable pipelined datapath," in *Proc. FPL*, Sep. 1996, pp. 126–135.

[58] S. C. Goldstein *et al.*, "PipeRench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, Apr. 2000.

[59] H. Singh *et al.*, "Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, pp. 465–481, 2000.

[60] A. Papakonstantinou *et al.*, "Efficient compilation of CUDA kernels for high-performance computing on FPGAs," *ACM Trans. Embed. Comp. Syst.*, vol. 13, no. 2, pp. 35–42, Sep. 2013.

[61] N. Kapre and A. DeHon, "Performance comparison of single-precision SPICE model-evaluation on FPGA, GPU, Cell, and multi-core processors," in *Proc. FPL*, 2009, pp. 65–72.

[62] B. Cope, P. Cheung, W. Luk, and S. Witt, "Have GPUs made FPGAs redundant in the field of video processing?" in *Proc. ICFPT*, Dec. 2005, pp. 111–118.

[63] E. Mirsky and A. DeHon, "MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proc. FCCM*, Apr. 1996, pp. 157–166.

[64] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *Proc. FPGA*, 1999, pp. 135–143.

[65] K. Compton and S. Hauck, "Totem: Custom reconfigurable array generation," in *Proc. FCCM*, 2001, pp. 111–119.

[66] K. Eguro and S. Hauck, "Resource allocation for coarse grain FPGA development," *IEEE Trans. Comput.-Aided Design*, vol. 24, no. 10, pp. 1572–1581, Oct. 2005.

[67] K. Compton and S. Hauck, "Automatic design of reconfigurable domain-specific flexible cores," *IEEE Trans. VLSI Syst.*, vol. 16, no. 5, pp. 493–503, May 2008.

[68] J. Nickolls *et al.*, "Calisto: A low-power single-chip multiprocessor communications platform," *IEEE Micro*, vol. 23, no. 2, pp. 29–43, Mar. 2003.

[69] S. Lee, J. Song, M. Kim, D. Kim, and S. Lee, "H.264/AVC UHD decoder implementation on multi-cluster platform using hybrid parallelization method," in *Proc. Int. Conf. Image Process.*, Sep. 2011, pp. 381–384.

[70] C. Kim *et al.*, "ULP-SRP: Ultra low-power Samsung reconfigurable processor for biomedical applications," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 3, pp. 22:1–22:15, Sep. 2014.

[71] S. Hauck, Z. Li, and E. J. Schwabe, "Configuration compression for the Xilinx XC6200 FPGA," *IEEE Trans. Comput.-Aided Design*, vol. 18, no. 8, pp. 107–1113, Aug. 1999.

[72] K. Compton, Z. Li, J. Cooley, and S. K. S. Hauck, "Configuration relocation and defragmentation for run-time reconfigurable computing," *IEEE Trans. VLSI Syst.*, vol. 10, no. 3, pp. 209–220, May 2002.

[73] G. Brebner and D. Levi, "Networking on chip with platform FPGAs," in *Proc. ICFPT*, Dec. 2003, pp. 13–20.

[74] T. Marescaux *et al.*, "Run-time support for heterogeneous multitasking on reconfigurable SoCs," *INTEGRATION, The VLSI J.*, vol. 38, no. 1, pp. 107–130, 2004.

[75] N. Kapre N. Mehta *et al.*, "Packet-switched vs. time-multiplexed FPGA overlay networks," in *Proc. FCCM*, 2006, pp. 205–213.

[76] M. K. Papamichael and J. C. Hoe, "CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs," in *Proc. FPGA*, 2012, pp. 37–46.

[77] Y. Huan and A. DeHon, "FPGA optimized packet-switched NoC using split and merge primitives," in *Proc. ICFPT*, Dec. 2012, pp. 47–52.

[78] M. S. Abdelfattah and V. Betz, "Design tradeoffs for hard and soft FPGA-based networks-on-chip," in *Proc. ICFPT*, Dec. 2012, pp. 95–103.

[79] J. Babb *et al.*, "Logic emulation with virtual wires," *IEEE Trans. Comput.-Aided Design*, vol. 16, no. 6, pp. 609–626, Jun. 1997.

[80] S.-L. L. Lu, P. Yiannacouras, T. Suh, R. Kassa, and M. Konow, "A desktop computer with a reconfigurable Pentium," *ACM Trans. Reconfig. Tech. Syst.*, vol. 1, no. 1, Mar. 2008.

[81] C. Chang, J. Wawrzynek, and R. Brodersen, "BEE2: A high-end reconfigurable computing system," *IEEE Des. Test. Comput.*, vol. 22, no. 2, pp. 114–125, 2004.

[82] J. Wawrzynek *et al.*, "RAMP: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, 2007.

[83] S. Wee *et al.*, "A practical FPGA based framework for novel CMP research," in *Proc. FPGA*, 2007, pp. 116–125.

[84] E. S. Chung *et al.*, "ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs," *ACM Trans. Reconfig. Tech. Syst.*, vol. 2, no. 2, pp. 15:1–15:32, Jun. 2009.

[85] D. Chiou *et al.*, "FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators," in *Proc. Proc. Int. Symp. Microarch.*, Dec. 2007, pp. 249–261.

[86] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-Port networks: Preserving the timed behavior of synchronous systems for modeling on FPGAs," *ACM Trans. Reconfig. Technol. Syst.*, vol. 2, no. 3, pp. 16:1–16:26, Sep. 2009.

[87] P. Diniz and J. Park, "Data search and reorganization using FPGAs: Application to spatial pointer-based data structures," in *Proc. FCCM*, 2003, pp. 207–217.

[88] M. deLorimier, N. Kapre, N. Mehta, and A. DeHon, "Spatial hardware implementation for sparse graph algorithms in GraphStep," *ACM Trans. Autonom. Adapt. Syst*, vol. 6, no. 3, pp. 17:1–17:20, Sep. 2011.

[89] H. Yu and M. Leeser, "Automatic sliding window operation optimization for FPGA-based," in *Proc. FCCM*, Apr. 2006, pp. 76–88.

[90] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for FPGA-based computing," in *Proc. FPGA*, 2011, pp. 97–106.

[91] P. Yiannacouras and J. Rose, "A parameterized automatic cache generator for FPGAs," in *Proc. ICFPT*, 2003, pp. 324–327.

[92] C. E. LaForest and G. Steffan, "Efficient multi-ported memories for FPGAs," in *Proc. FPGA*, 2010, pp. 41–50.

[93] U. Dhawan and A. DeHon, "Area-efficient near-associative memories on FPGAs," in *Proc. FPGA*, 2013, pp. 191–200.

[94] A. J. Yu and G. G. Lemieux, "FPGA defect tolerance: Impact of granularity," in *Proc. ICFPT*, 2005, pp. 189–196.

[95] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Low overhead fault-tolerant FPGA systems," *IEEE Trans. VLSI Syst.*, vol. 6, no. 2, pp. 212–221, Jun. 1998.

[96] V. Lakamraju and R. Tessier, "Tolerating operational faults in cluster-based FPGAs," in *Proc. FPGA*, 2000, pp. 187–194.

[97] R. Rubin and A. DeHon, "Choose-your-own-adventure routing: Lightweight load-time defect avoidance," *ACM Trans. Reconfig. Technol. Syst.*, vol. 4, no. 4, Dec. 2011.

[98] W. B. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider, "Defect tolerance on the TERAMAC custom computer," in *Proc. FCCM*, Apr. 1997, pp. 116–123.

[99] H. Naeimi and A. DeHon, "A greedy algorithm for tolerating defective crosspoints in NanoPLA design," in *Proc. ICFPT*, Dec. 2004, pp. 49–56.

[100] J. R. Heath, P. J. Kuekes, G. S. Snider, and R. S. Williams, "A defect-tolerant computer architecture: Opportunities for nanotechnology," *Science*, vol. 280, no. 5370, pp. 1716–1721, Jun. 12, 1998.

[101] A. DeHon, "Nanowire-based programmable architectures," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 1, no. 2, pp. 109–162, Jul. 2005.

[102] M. B. Tahoori, "Application-dependent testing of FPGAs," *IEEE Trans. VLSI Syst.*, vol. 14, no. 9, pp. 1024–1033, Sep. 2006.

[103] P. Sedcole and P. Y. K. Cheung, "Within-die delay variability in 90 nm FPGAs and beyond," in *Proc. ICFPT*, 2006, pp. 97–104.

[104] Y. Matsumoto *et al.*, "Suppression of intrinsic delay variation in FPGAs using multiple configurations," *ACM Trans. Reconfig. Technol. Syst.*, vol. 1, no. 1, Mar. 2008.

[105] N. Mehta, R. Rubin, and A. DeHon, "Limit study of energy & delay benefits of component-specific routing," in *Proc. FPGA*, 2012, pp. 97–106.

[106] J. S. Wong, P. Sedcole, and P. Y. K. Cheung, "Self-measurement of combinatorial circuit delays in FPGAs," *ACM Trans. Reconfig. Technol. Syst.*, vol. 2, no. 2, pp. 1–22, Jun. 2009.

[107] B. Gojman and A. DeHon, "GROK-INT: Generating real on-chip knowledge for interconnect delays using timing extraction," in *Proc. FCCM*, 2014, pp. 88–95.

[108] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici, "Dynamic fault tolerance in FPGAs via partial reconfiguration," in *Proc. FCCM*, 2000, pp. 165–174.

[109] S. Srinivasan *et al.*, "Toward increasing FPGA lifetime," *IEEE Trans. Dep. Secure Comput.*, vol. 5, no. 2, pp. 115–127, 2008.

[110] M. J. Wirthlin, E. Johnson, N. Rollins, M. Caffrey, and P. Graham, "The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets," in *Proc. FCCM*, 2003, pp. 133–142.

[111] H. Quinn and P. Graham, "Terrestrial-based radiation upsets: A cautionary tale," in *Proc. FCCM*, 2005, pp. 193–202.

[112] M. Caffrey *et al.*, "On-orbit flight results from the reconfigurable Cibola flight experiment satellite (CFESat)," in *Proc. FCCM*, 2009, pp. 3–10.

[113] B. Pratt *et al.*, "Fine-grain SEU mitigation for FPGAs using partial TMR," *IEEE Trans. Nucl. Sci.*, vol. 55, no. 4, pp. 2274–2280, Aug. 2008.

[114] L. Sterpone, "A new timing driven placement algorithm for dependable circuits on SRAM-based FPGAs," *ACM Trans. Reconfig. Technol. Syst.*, vol. 4, no. 1, pp. 7:1–7:21, Dec. 2010.

[115] M. Wazlowski *et al.*, "PRISM II: Compiler and architecture," in *Proc. FCCM*, Apr. 1993, pp. 9–16.

[116] D. Wo and K. Forward, "Compiling to the gate level for a reconfigurable co-processor," in *Proc. FCCM*, Apr. 1994, pp. 147–154.

[117] M. Gokhale and J. Stone, "NAPA C: Compiling for a hybrid RISC/FPGA architecture," in *Proc. FCCM*, Apr. 1998, pp. 126–135.

[118] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Proc. FCCM*, May 2010, pp. 127–134.

[119] Y. Yankova *et al.*, "DWARV: Delftworkbench automated reconfigurable VHDL generator," in *Proc. FPL*, Aug. 2007, pp. 126–135.

[120] A. Canis *et al.*, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embed. Comp. Sys.*, vol. 13, no. 2, Sep. 2013.

[121] J. Cong *et al.*, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[122] J. Tripp, M. Gokhale, and K. Peterson, "Trident: From high-level language to hardware circuitry," *Computer*, vol. 40, no. 3, pp. 28–37, Mar. 2007.

[123] M. Haldar, A. Nayak, A. N. Choudhary, and P. Banerjee, "A system for synthesizing optimized FPGA hardware from MATLAB," in *Proc. ICCAD*, Nov. 2001, pp. 314–319.

[124] J. Ou and V. Prasanna, "PyGen: a MATLAB/Simulink based tool for synthesizing parameterized and energy efficient designs using FPGAs," in *Proc. FCCM*, Apr. 2004, pp. 47–56.

[125] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, "Laura: Leiden architecture research and exploration tool," in *Proc. FPL*, Sep. 2003, pp. 911–920.

[126] B. V. Herzen, "Signal processing at 250 MHz using high-performance FPGA's," in *Proc. FPGA*, Feb. 1997, pp. 62–68.

[127] M. Gokhale and B. Schott, "Data-parallel C on a reconfigurable logic array," *J. Supercomp.*, vol. 9, no. 3, pp. 291–313, 1995.

[128] M. Weinhardt and W. Luk, "Pipeline vectorization for reconfigurable systems," in *Proc. FCCM*, Apr. 1999, pp. 52–62.

[129] H. E. Ziegler, M. W. Hall, and P. C. Diniz, "Compiler-generated communication for pipelined FPGA applications," in *Proc. DAC*, Jun. 2003, pp. 610–615.

[130] R. Rodrigues, J. M. P. Cardoso, and P. C. Diniz, "A data-driven approach for pipelining sequences of data-dependent loops," in *Proc. FCCM*, Apr. 2007, pp. 219–228.

[131] N. Kapre and A. DeHon, "VLIW-SCORE: Beyond C for sequential control of SPICE FPGA acceleration," in *Proc. ICFPT*, Dec. 2011, pp. 1–9.

[132] M. Gokhale and J. Stone, "Automatic allocation of arrays to memories in FPGA processors with multiple memory banks," in *Proc. FCCM*, Apr. 1999, pp. 63–69.

[133] N. Baradaran and P. Diniz, "A compiler approach to managing storage and memory bandwidth in configurable architectures," *ACM Trans. Des. Auto. Elec. Syst.*, vol. 13, no. 4, Sep. 2008.

[134] S. Cheng, M. Lin, H. J. Liu, S. Scott, and J. Wawrzynek, "Exploiting memory-level parallelism in reconfigurable accelerators," in *Proc. FCCM*, May 2012, pp. 157–160.

[135] O. Mencer, M. Morf, and M. J. Flynn, "PAM-Blox: High performance FPGA design for adaptive computing," in *Proc. FCCM*, Apr. 1998, pp. 167–174.

[136] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *Proc. FCCM*, Apr. 2000, pp. 49–56.

[137] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *Proc. FCCM*, Apr. 2007, pp. 55–64.

[138] P. Bellows and B. Hutchings, "JHDL—An HDL for reconfigurable systems," in *Proc. FCCM*, Apr. 1998, pp. 175–184.

[139] A. DeHon *et al.*, "Stream computations organized for reconfigurable execution," *J. Microproc. Microsyst.*, vol. 30, no. 6, pp. 334–354, Sep. 2006.

[140] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for SoPC area reduction," in *Proc. FCCM*, Apr. 2006, pp. 131–142.

[141] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Scaling soft processor systems," in *Proc. FCCM*, Apr. 2008, pp. 195–205.

[142] D. L. Andrews, R. Sass, E. K. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Kom, "Achieving programming model abstractions for reconfigurable computing," *IEEE Trans. VLSI Syst.*, vol. 16, no. 1, pp. 34–44, Jan. 2008.

[143] T. S. Czajkowski *et al.*, "From OpenCL to high-performance hardware on FPGAs," in *Proc. FPL*, Sep. 2012, pp. 531–534.

[144] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," in *Proc. FCCM*, May 2014.

[145] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. ISCA*, Jun. 2014, pp. 13–24.

[146] B. Hutchings *et al.*, "A CAD suite for high-performance FPGA design," in *Proc. FCCM*, Apr. 1999, pp. 12–24.

[147] J. Hwang, B. Milne, N. Shirazi, and J. Stroomer, "System level tools for DSP in FPGAs," in *Proc. FPL*, Aug. 2001, pp. 534–543.

[148] B. Holland, K. Nagarajan, and A. George, "RAT: RC amenability test for rapid performance prediction," *ACM Trans. Reconfig. Technol. Syst.*, vol. 1, no. 4, Jan. 2009.

[149] O. Ulusel, K. Nepal, R. I. Bahar, and S. Reda, "Fast design exploration for performance, power and accuracy tradeoffs in FPGA-based accelerators," *ACM Trans. Reconfig. Technol. Syst.*, vol. 7, no. 1, Feb. 2014.

[150] K. S. Hemmert, J. L. Tripp, B. L. Hutchings, and P. A. Jackson, "Source level debugger for the sea cucumber synthesizing compiler," in *Proc. FCCM*, Apr. 2003, pp. 228–237.

[151] A. L. Slade, B. E. Nelson, and B. L. Hutchings, "Reconfigurable computing application frameworks," in *Proc. FCCM*, Apr. 2003, pp. 251–260.

[152] E. Hung and S. J. E. Wilton, "Accelerating FPGA debug: Increasing visibility using a runtime reconfigurable observation and

triggering," *ACM Trans. Des. Auto. Elec. Syst.*, vol. 19, no. 2, Mar. 2014.

[153] B. Hutchings, B. Nelson, and M. J. Wirthlin, "Designing and debugging custom computing applications," *IEEE Des. Test. Comput.*, vol. 17, no. 1, pp. 20–28, Jan. 2000.

[154] K. S. Chatha and R. Vemuri, "Hardware-software codesign for dynamically reconfigurable architectures," in *Proc. FPL*, Aug. 1999, pp. 175–184.

[155] K. Bertels *et al.*, "HArtes: Hardware-software codesign for heterogeneous multicore platforms," *IEEE Micro*, vol. 30, no. 5, pp. 88–97, Sep. 2010.

[156] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: A first approach," in *Proc. DAC*, Jun. 2003, pp. 250–255.

[157] L. Shannon and P. Chow, "Using reconfigurability to achieve real-time profiling for hardware/software codesign," in *Proc. FPGA*, Feb. 2010, pp. 190–199.

[158] M. L. Chang and S. Hauck, "Précis: A user-centric wordlength optimization tool," *IEEE Des. Test. Comput.*, vol. 22, no. 4, pp. 349–361, Jul. 2005.

[159] G. Constantinides, "Perturbation analysis for word-length optimization," in *Proc. FCCM*, Apr. 2003, pp. 81–90.

[160] A. A. Gaffar, O. Mencer, W. Luk, and P. Y. Cheung, "Unifying bit-width optimisation for fixed-point and floating-point designs," in *Proc. FCCM*, Apr. 2004, pp. 79–88.

[161] A. Klimovic and J. H. Anderson, "Bitwidth-optimized hardware accelerators with software fallback," in *Proc. ICFPT*, Dec. 2013, pp. 136–143.

[162] T. Callahan, P. Chong, A. DeHon, and J. Wawrzynek, "Fast module mapping and placement for datapaths in FPGAs," in *Proc. FPGA*, Feb. 1998, pp. 123–132.

[163] C. Lavin *et al.*, "HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping," in *Proc. FCCM*, May 2011, pp. 117–124.

[164] N. Steiner *et al.*, "Torc: Towards an open-source tool flow," in *Proc. FPGA*, Feb. 2011, pp. 41–44.

[165] C. Lavin *et al.*, "RapidSmith: Do-it-yourself CAD tools for Xilinx FPGAs," in *Proc. FPL*, Sep. 2011, pp. 349–355.

[166] P. Foulk, "Data folding in SRAM-configurable FPGAs," in *Proc. FCCM*, Apr. 1993, pp. 163–171.

[167] D. M. Lewis, M. H. van Ierssel, and D. H. Wong, "A field programmable accelerator for compiled-code applications," in *Proc. FCCM*, Apr. 1993, pp. 60–67.

[168] J. Villasenor, B. Schoner, K.-N. Chia, and C. Zapata, "Configurable computer solutions for automatic target recognition," in *Proc. FCCM*, Apr. 1996, pp. 70–79.

[169] B. Mei, S. Vemalde, D. Verkest, H. D. Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *Proc. ICFPT*, Dec. 2002, pp. 166–173.

[170] J. Eun Lee, K. Choi, and N. Dutt, "Compilation approach for coarse-grained reconfigurable architectures," *IEEE Des. Test. Comput.*, vol. 20, no. 1, pp. 26–33, Jan. 2003.

[171] S. Friedman *et al.*, "SPR: An architecture-adaptive CGRA mapping tool," in *Proc. FPGA*, Feb. 2009, pp. 191–200.

[172] J. Hadley and B. Hutchings, "Design methodologies for partially reconfigured systems," in *Proc. FCCM*, Apr. 1995, pp. 78–84.

[173] W. Luk, N. Shirazi, and P. Y. K. Cheung, "Compilation tools for run-time reconfigurable designs," in *Proc. FCCM*, Apr. 1997, pp. 56–65.

[174] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited Paper: Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *Proc. FPL*, Aug. 2006, pp. 1–6.

[175] P. Lysaght and J. Stockwood, "A simulation tool for dynamically reconfigurable field programmable gate arrays," *IEEE Trans. VLSI Syst.*, vol. 4, no. 3, pp. 381–390, Sep. 1996.

[176] C. Beckhoff, D. Koch, and J. Torresen, "GOAHEAD: A partial reconfiguration framework," in *Proc. FCCM*, May 2012, pp. 37–44.

[177] K. Vipin and S. A. Fahmy, "Automated partial reconfiguration design for adaptive systems with CoPR for Zynq," in *Proc. FCCM*, May 2014.

[178] C. Neely, G. Brebner, and W. Shang, "ReShape: Towards a high-level approach to design and operation of modular reconfigurable systems," *ACM Trans. Reconfig. Technol. Syst.*, vol. 6, no. 1, May 2013.

[179] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," in *Proc. FCCM*, Apr. 1995.

[180] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proc. FPGA*, Feb. 2004, pp. 183–189.

[181] P. Hsiung, C. Lin, and C. Liao, "Perfecto: A System C-based design-space exploration framework for dynamically reconfigurable architectures," *ACM Trans. Reconfig. Technol. Syst.*, vol. 1, no. 3, Sep. 2008.

[182] J. Cardoso, "On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures," *IEEE Trans. Comput.*, vol. 52, no. 10, pp. 1362–1375, Oct. 2003.

[183] R. Maestre *et al.*, "A framework for reconfigurable computing: Task scheduling and context management," *IEEE Trans. VLSI Syst.*, vol. 9, no. 6, pp. 858–873, Dec. 2001.

[184] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Wit, "A dynamic reconfiguration run-time system," in *Proc. FCCM*, Apr. 1997, pp. 66–75.

[185] G. Brebner, "A virtual hardware operating system for the Xilinx XC6200," in *Proc. FPL*, Sep. 1996, pp. 327–336.

[186] W. Fu and K. Compton, "An execution environment for reconfigurable computing," in *Proc. FCCM*, Apr. 2005, pp. 149–158.

[187] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1393–1407, Nov. 2004.

[188] S. A. Fahmy, J. Lotze, J. Noguera, L. Doyle, and R. Esser, "Generic software framework for adaptive applications on FPGAs," in *Proc. FCCM*, Apr. 2009, pp. 55–62.

[189] A. Ismail and L. Shannon, "FUSE: Front-end user framework for O/S abstraction of hardware accelerators," in *Proc. FCCM*, Apr. 2011, pp. 170–177.

[190] H. K.-H. So and R. W. Broderson, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *ACM Trans. Embed. Comp. Syst.*, vol. 7, no. 2, Feb. 2008.

[191] J. Villasenor, C. Jones, and B. Schoner, "Video communications using rapidly reconfigurable hardware," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 5, pp. 565–567, Dec. 1995.

[192] E. Lemoine and D. Merceron, "Run time reconfiguration of FPGA for scanning genomic databases," in *Proc. FCCM*, Apr. 1995, pp. 90–98.

[193] D. Koch and J. Torresen, "FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *Proc. FPGA*, Feb. 2011, pp. 45–54.

[194] S. Swaminathan, R. Tessier, D. Goeckel, and W. Burleson, "A dynamically reconfigurable adaptive Viterbi decoder," in *Proc. FPGA*, Feb. 2002, pp. 227–236.

[195] D. Unnikrishnan *et al.*, "Reconfigurable data planes for scalable network virtualization," *IEEE Trans. Comput.*, vol. 62, no. 12, pp. 2476–2488, Dec. 2013.

[196] C. Dennl, D. Ziener, and J. Teich, "On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library," in *Proc. FCCM*, Apr. 2012, pp. 45–52.

[197] R. J. Petersen and B. L. Hutchings, "An assessment of the suitability of FPGA-based systems for use in digital signal processing," in *Proc. FPL*, 1995, pp. 293–302.

[198] P. A. Jackson, C. P. Chan, J. E. Scalera, C. M. Rader, and M. M. Vai, "A systolic FFT architecture for real time FPGA systems," in *Proc. HPEC*, 2004.

[199] J.-W. Jang, S. Choi, and V. Prasanna, "Area and time efficient implementations of matrix multiplication on FPGAs," in *Proc. ICFPT*, Dec. 2002, pp. 93–100.

[200] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proc. FPGA*, 1998, pp. 191–200.

[201] J. M. Rabaey, "Silicon platforms for the next generation wireless systems what role does reconfigurable hardware play?" in *Proc. FPL*, 2000, pp. 277–285.

[202] N. Sedcole, P. Cheung, G. Constantinides, and W. Luk, "A reconfigurable platform for real-time embedded video image processing," in *Proc. FPL*, 2003, pp. 606–615.

[203] E. El-Araby, T. El-Ghazawi, J. Le-Moigne, and K. Gaj, "Wavelet spectral dimension reduction of hyperspectral imagery on a reconfigurable computer," in *Proc. ICFPT*, Dec. 2004, pp. 399–402.

[204] T. Fry and S. Hauck, "SPIHT image compression on FPGAs," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 9, pp. 1138–1147, Sep. 2005.

[205] H. Eeckhaut, M. Christiaens, D. Stroobandt, and V. Nollet, "Optimizing the critical loop in the H.264/AVC CABAC decoder," in *Proc. ICFPT*, Dec. 2006, pp. 113–118.

[206] R. McCready, "Real-time face detection on a configurable hardware system," in *Proc. FPL*, 2000, pp. 157–162.

[207] P. Dunn and P. Corke, "Real-time stereopsis using FPGAs," in *Proc. FPL*, 1997, pp. 400–409.

[208] M. Genovese and E. Napoli, "ASIC and FPGA implementation of the gaussian mixture model algorithm for real-time segmentation of high definition video," *IEEE Trans. VLSI Syst.*, vol. 22, no. 3, pp. 537–547, Mar. 2014.

[209] G. Zhang *et al.*, "Reconfigurable acceleration for monte carlo based financial simulation," in *Proc. ICFPT*, Dec. 2005, pp. 215–222.

[210] D. Thomas and W. Luk, "Credit risk modelling using hardware accelerated Monte-Carlo simulation," in *Proc. FCCM*, Apr. 2008, pp. 229–238.

[211] G. Morris and M. Aubury, "Design space exploration of the european option benchmark using hyperstreams," in *Proc. FPL*, Aug. 2007, pp. 5–10.

[212] P.-W. Leong, M. P. Leong, O. Cheung, T. Tung, C. Kwok, M. Wong, and K. Lee, "Pilchard—A reconfigurable computing platform with memory slot interface," in *Proc. FCCM*, Mar. 2001, pp. 170–179.

[213] J. Leonard and W. H. Mangione-Smith, "A case study of partially evaluated hardware circuits: Key-specific DES," in *Proc. FPL*, 1997, pp. 151–160.

[214] M. McLoone and J. V. McCanny, "Single-chip FPGA implementation of the advanced encryption standard algorithm," in *Proc. FPL*, 2001, pp. 152–161.

[215] S. Drimer, T. Güneysu, and C. Paar, "DSPs, BRAMs, and a pinch of logic: Extended recipes for AES on FPGAs," *ACM Trans. Reconfig. Technol. Syst.*, vol. 3, no. 1, pp. 3:1–3:27, Jan. 2010.

[216] M.-P. Leong, O. Y. H. Cheung, K. H. Tsoi, and P. H. W. Leong, "A bit-serial implementation of the international data encryption algorithm IDEA," in *Proc. FCCM*, 2000, pp. 122–131.

[217] M. Shand and J. Vuillemin, "Fast implementations of RSA cryptography," in *Proc. IEEE Symp. Comp. Arith.*, Jun. 1993, pp. 252–259.

[218] S. H. Tang, K. S. Tsui, and P.-W. Leong, "Modular exponentiation using parallel multipliers," in *Proc. ICFPT*, Dec. 2003, pp. 52–59.

[219] G. Orlando and C. Paar, "A super-serial Galois fields multiplier for FPGAs and its application to public-key algorithms," in *Proc. FCCM*, 1999, pp. 232–239.

[220] K. H. Leung, K. W. Ma, W. K. Wong, and P.-W. Leong, "FPGA implementation of a microcoded elliptic curve cryptographic processor," in *Proc. FCCM*, 2000, pp. 68–76.

[221] K. Jarvinen and J. Skytta, "High-speed elliptic curve cryptography accelerator for Koblitz curves," in *Proc. FCCM*, Apr. 2008, pp. 109–118.

[222] K. Tsoi, K. Lee, and P. H. W. Leong, "A massively parallel RC4 key search engine," in *Proc. FCCM*, 2002, pp. 13–21.

[223] M. Simka *et al.*, "Hardware factorization based on elliptic curve method," in *Proc. FCCM*, Apr. 2005, pp. 107–116.

[224] T. Güneysu, C. Paar, and J. Pelzl, "Special-purpose hardware for solving the elliptic curve discrete logarithm problem," *ACM Trans. Reconfig. Technol. Syst.*, vol. 1, no. 2, pp. 8:1–8:21, Jun. 2008.

[225] M. Abramovici and D. Saab, "Satisfiability on reconfigurable hardware," in *Proc. FPL*, 1997, pp. 448–456.

[226] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean satisfiability with configurable hardware," in *Proc. FCCM*, Apr. 1998, pp. 186–195.

[227] A. Rashid, J. Leonard, and W. Mangione-Smith, "Dynamic circuit generation for solving specific problem instances of Boolean satisfiability," in *Proc. FCCM*, Apr. 1998, pp. 196–204.

[228] J. De Sousa, J. M. Da Silva, and M. Abramovici, "A configurable hardware/software approach to SAT solving," in *Proc. FCCM*, Mar. 2001, pp. 239–248.

[229] M. Fuess, M. Leeser, and T. Leonard, "An FPGA implementation of explicit-state model checking," in *Proc. FCCM*, Apr. 2008, pp. 119–126.

[230] P. Graham and B. Nelson, "A hardware genetic algorithm for the traveling salesman problem on Splash 2," in *Proc. FPL*, 1995, pp. 352–361.

[231] I. Mavroidis, I. Papaefstathiou, and D. Pnevmatikatos, "A fast FPGA-Based 2-opt solver for small-scale Euclidean traveling salesman problem," in *Proc. FCCM*, Apr. 2007, pp. 13–22.

[232] C. Plessl and M. Platzner, "Custom computing machines for the set covering problem," in *Proc. FCCM*, 2002, pp. 163–172.

[233] S. Hemmert, B. Hutchings, and A. Malvi, "An application-specific compiler for high-speed binary image morphology," in *Proc. FCCM*, Mar. 2001, pp. 199–208.

[234] D. T. Hoang, "Searching genetic databases on Splash 2," in *Proc. FCCM*, Apr. 1993, pp. 185–191.

[235] M. Herbordt, J. Model, Y. Gu, B. Sukhwani, and T. VanCourt, "Single pass, BLAST-like, approximate string matching on FPGAs," in *Proc. FCCM*, Apr. 2006, pp. 217–226.

[236] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain, "Mercury BLASTP: Accelerating protein sequence alignment," *ACM Trans. Reconfig. Technol. Syst.*, vol. 1, no. 2, pp. 9:1–9:44, Jun. 2008.

[237] N. Alachiotis, S. Berger, and A. Stamatakis, "Accelerating phylogeny-aware short DNA read alignment with FPGAs," in *Proc. FCCM*, May 2011, pp. 226–233.

[238] C. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. Ruzzo, "Hardware acceleration of short read mapping," in *Proc. FCCM*, Apr. 2012, pp. 161–168.

[239] A. Jacob, J. Buhler, and R. Chamberlain, "Rapid RNA folding: Analysis and acceleration of the Zuker recurrence," in *Proc. FCCM*, May 2010, pp. 87–94.

[240] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable network packet processing on the field programmable port extender (FPX)," in *Proc. FPGA*, 2001, pp. 87–93.

[241] D. Schuehler and J. Lockwood, "A modular system for FPGA-based TCP flow processing in high-speed networks," in *Proc. FPL*, 2004, pp. 301–310.

[242] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, "NetFPGA: An open platform for teaching how to build Gigabit-rate network switches and routers," *IEEE Trans. Educ.*, vol. 51, no. 3, pp. 364–369, 2008.

[243] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," in *Proc. ACM/IEEE Symp. ANCS*, 2008.

[244] C. Kulkarni, G. Brebner, and G. Schelle, "Mapping a domain specific language to a platform FPGA," in *DAC*, 2004, pp. 924–927.

[245] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer, "An FPGA-based soft multiprocessor system for IPv4 packet forwarding," in *Proc. FPL*, 2005, pp. 487–492.

[246] Z. Dai and J. Zhu, "Saturating the transceiver bandwidth: Switch fabric design on FPGAs," in *Proc. FCCM*, 2012, pp. 67–75.

[247] N. Zilberman, P. Watts, C. Rotsos, and A. W. Moore, "Reconfigurable network systems and software defined networking," *Proc. IEEE*, to be published.

[248] R. Sidhu and V. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. FCCM*, Mar. 2001, pp. 227–238.

[249] B. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Proc. FCCM*, 2002, pp. 111–120.

[250] Y. Cho and W. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," in *Proc. FCCM*, Apr. 2004, pp. 125–134.

[251] Y. Cho and W. Mangione-Smith, "Fast reconfiguring deep packet filter for 1+ Gigabit network," in *Proc. FCCM*, Apr. 2005, pp. 215–224.

[252] M. Attig and J. Lockwood, "A framework for rule processing in reconfigurable network systems," in *Proc. FCCM*, Apr. 2005, pp. 225–234.

[253] X. Wang and M. Leeser, "VFloat: A variable precision fixed- and floating-point library for reconfigurable hardware," *ACM Trans. Reconfig. Technol. Syst.*, vol. 3, no. 3, pp. 16:1–16:34, Sep. 2010.

[254] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," in *Proc. FPGA*, Feb. 2004, pp. 171–180.

[255] K. Underwood and K. Hemmert, "Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance," in *Proc. FCCM*, Apr. 2004, pp. 219–228.

[256] K. Hemmert and K. Underwood, "An analysis of the double-precision floating-point FFT on FPGAs," in *Proc. FCCM*, Apr. 2005, pp. 171–180.

[257] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Proc. FPGA*, Feb. 2005, pp. 75–85.

[258] D. Boland and G. A. Constantinides, "Optimizing memory bandwidth use and performance for matrix-vector multiplication in iterative methods," *ACM Trans. Reconfig. Technol. Syst.*, vol. 4, no. 3, pp. 22:1–22:14, Aug. 2011.

[259] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *Proc. ICFPT*, Dec. 2008, pp. 33–40.

[260] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proc. FPGA*, 2005, pp. 63–74.

[261] A. Roldao and G. A. Constantinides, "A high throughput FPGA-based floating point conjugate gradient implementation for dense matrices," *ACM Trans. Reconfig. Technol. Syst.*, vol. 3, no. 1, pp. 1:1–1:19, Jan. 2010.

[262] D. Dubois, A. Dubois, T. Boorman, C. Connor, and S. Poole, "Sparse matrix-vector multiplication on a reconfigurable supercomputer with application," *ACM Trans. Reconfig. Technol. Syst.*, vol. 3, no. 1, pp. 2:1–2:31, Jan. 2010.

[263] R. Strzodka and D. Goddeke, "Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components," in *Proc. FCCM*, Apr. 2006, pp. 259–270.

[264] M. Huang and O. Kilic, "Reaping the processing potential of FPGA on double-precision floating-point operations: An eigenvalue solver case study," in *Proc. FCCM*, May 2010, pp. 95–102.

[265] W. Zhang, V. Betz, and J. Rose, "Portable and scalable FPGA-based acceleration of a direct linear system solver," *ACM Trans. Reconfig. Technol. Syst.*, vol. 5, no. 1, pp. 6:1–6:26, Mar. 2012.

[266] N. Kapre and A. DeHon, "SPICE^2: Spatial processors interconnected for concurrent execution for accelerating the SPICE circuit simulator using an FPGA," *IEEE Trans. Comput.-Aided Design*, vol. 31, no. 1, pp. 9–22, Jan. 2012.

[267] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow, "Reconfigurable molecular dynamics simulator," in *Proc. FCCM*, Apr. 2004, pp. 197–206.

[268] M. Chiu and M. C. Herbordt, "Molecular dynamics simulations on high-performance reconfigurable computing systems," *ACM Trans. Reconfig. Technol. Syst.*, vol. 3, no. 4, pp. 23:1–23:37, Nov. 2010.

ABOUT THE AUTHORS

Russell Tessier (Senior Member, IEEE) received the B.S. degree in computer and systems engineering from Rensselaer Polytechnic Institute, Troy, NY, USA, in 1989, and the S.M. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, MA, in 1992 and 1999, respectively.

He is a Professor of Electrical and Computer Engineering at the University of Massachusetts, Amherst, MA, USA. His research interests include embedded systems, FPGAs, and system verification.

Kenneth Pocek (Life Member, IEEE) received the B.S. degree in computer engineering from Case Western Reserve University, Cleveland, OH, USA, in 1969 and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 1971 and 1979, respectively.

He worked at Hewlett-Packard as a hardware architect and designer on their first computers (1969 to 1981), at General Electric Semiconductor in ASIC's (1982 to 1986), and finally at Intel Corporation in IC technology, research, and software development (1987 to 2004). He is retired and actively engaged in application development for Android and IOS devices. Dr. Pocek co-founded the IEEE FCCM conference with Duncan Buell in 1993 and chaired the conference until 2009.

André DeHon (Member, IEEE) received S.B., S.M., and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 1990, 1993, and 1996, respectively.

From 1996 to 1999, André co-ran the BRASS group in the Computer Science Department at the University of California, Berkeley, CA, USA. From 1999 to 2006, he was an Assistant Professor of Computer Science at the California Institute of Technology. Since 2006, he has been in the Electrical and Systems Engineering Department at the University of Pennsylvania where he is now a Full Professor. He is broadly interested in how we physically implement computations from substrates, including VLSI and molecular electronics, up through architecture, CAD, and programming models. He places special emphasis on spatial programmable architectures (e.g., FPGAs) and interconnect design and optimization.