Reconfigurable Computing: A Survey of Systems and Software

KATHERINE COMPTON

Northwestern University

AND

SCOTT HAUCK

University of Washington

Due to its potential to greatly accelerate a wide variety of applications, reconfigurable computing has become a subject of a great deal of research. Its key feature is the ability to perform computations in hardware to increase performance, while retaining much of the flexibility of a software solution. In this survey, we explore the hardware aspects of reconfigurable computing machines, from single chip architectures to multi-chip systems, including internal structures and external coupling. We also focus on the software that targets these machines, such as compilation tools that map high-level algorithms directly to the reconfigurable substrate. Finally, we consider the issues involved in run-time reconfigurable systems, which reuse the configurable hardware during program execution.

Categories and Subject Descriptors: A.1 [**Introductory and Survey**]; B.6.1 [**Logic Design**]: Design Style—*logic arrays*; B.6.3 [**Logic Design**]: Design Aids; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*gate arrays*

General Terms: Design, Performance

Additional Key Words and Phrases: Automatic design, field-programmable, FPGA, manual design, reconfigurable architectures, reconfigurable computing, reconfigurable systems

1. INTRODUCTION

There are two primary methods in conventional computing for the execution of algorithms. The first is to use hardwired technology, either an Application Specific Integrated Circuit (ASIC) or a group of individual components forming a

This research was supported in part by Motorola, Inc., DARPA, and NSF.

K. Compton was supported by an NSF fellowship.

S. Hauck was supported in part by an NSF CAREER award and a Sloan Research Fellowship.

Authors' addresses: K. Compton, Department of Electrical and Computer Engineering, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208-3118; e-mail: kati@ece.northwestern.edu; S. Hauck, Department of Electrical Engineering, The University of Washington, Box 352500, Seattle, WA 98195; e-mail: hauck@ee.washington.edu.

board-level solution, to perform the operations in hardware. ASICs are designed specifically to perform a given computation, and thus they are very fast and efficient when executing the exact computation for which they were designed. However, the circuit cannot be altered after fabrication. This forces a redesign and refabrication of the chip if any part of its circuit requires modification. This is an expensive process, especially when one considers the difficulties in replacing ASICs in a large number of deployed systems. Board-level circuits are also somewhat inflexible, frequently requiring a board redesign and replacement in the event of changes to the application.

The second method is to use software-programmed microprocessors—a far more flexible solution. Processors execute a set of instructions to perform a computation. By changing the software instructions, the functionality of the system is altered without changing the hardware. However, the downside of this flexibility is that the performance can suffer, if not in clock speed then in work rate, and is far below that of an ASIC. The processor must read each instruction from memory, decode its meaning, and only then execute it. This results in a high execution overhead for each individual operation. Additionally, the set of instructions that may be used by a program is determined at the fabrication time of the processor. Any other operations that are to be implemented must be built out of existing instructions.

Reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware. Reconfigurable devices, including field-programmable gate arrays (FPGAs), contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements, sometimes known as logic blocks, are connected using a set of routing resources that are also programmable. In this way, custom digital circuits can be mapped to the reconfigurable hardware by computing the logic functions of the circuit within the logic blocks, and using the configurable routing to connect the blocks together to form the necessary circuit.

FPGAs and reconfigurable computing have been shown to accelerate a variety of applications. Data encryption, for example, is able to leverage both parallelism and fine-grained data manipulation. An implementation of the Serpent Block Cipher in the Xilinx Virtex XCV1000 shows a throughput increase by a factor of over 18 compared to a Pentium Pro PC running at 200 MHz [Elbirt and Paar 2000]. Additionally, a reconfigurable computing implementation of sieving for factoring large numbers (useful in breaking encryption schemes) was accelerated by a factor of 28 over a 200-MHz UltraSparc workstation [Kim and Mangione-Smith 2000]. The Garp architecture shows a comparable speed-up for DES [Hauser and Wawrzynek 1997], as does an FPGA implementation of an elliptic curve cryptography application [Leung et al. 2000].

Other recent applications that have been shown to exhibit significant speed-ups using reconfigurable hardware include: automatic target recognition [Rencher and Hutchings 1997], string pattern matching [Weinhardt and Luk 1999], Golomb Ruler Derivation [Dollas et al. 1998; Sotiriades et al. 2000], transitive closure of dynamic graphs [Huelsbergen 2000], Boolean satisfiability [Zhong et al. 1998], data compression [Huang et al. 2000], and genetic algorithms for the travelling salesman problem [Graham and Nelson 1996].

In order to achieve these performance benefits, yet support a wide range of applications, reconfigurable systems are usually formed with a combination of reconfigurable logic and a general-purpose microprocessor. The processor performs the operations that cannot be done efficiently in the reconfigurable logic, such as data-dependent control and possibly memory accesses, while the computational cores are mapped to the reconfigurable hardware. This reconfigurable logic can be

composed of either commercial FPGAs or custom configurable hardware.

Compilation environments for reconfigurable hardware range from tools to assist a programmer in performing a hand mapping of a circuit to the hardware, to complete automated systems that take a circuit description in a high-level language to a configuration for a reconfigurable system. The design process involves first partitioning a program into sections to be implemented on hardware, and those which are to be implemented in software on the host processor. The computations destined for the reconfigurable hardware are synthesized into a gate level or register transfer level circuit description. This circuit is mapped onto the logic blocks within the reconfigurable hardware during the technology mapping phase. These mapped blocks are then placed into the specific physical blocks within the hardware, and the pieces of the circuit are connected using the reconfigurable routing. After compilation, the circuit is ready for configuration onto the hardware at run-time. These steps, when performed using an automatic compilation system, require very little effort on the part of the programmer to utilize the reconfigurable hardware. However, performing some or all of these operations by hand can result in a more highly optimized circuit for performance-critical applications.

Since FPGAs must pay an area penalty because of their reconfigurability, device capacity can sometimes be a concern. Systems that are configured only at power-up are able to accelerate only as much of the program as will fit within the programmable structures. Additional areas of a program might be accelerated by reusing the reconfigurable hardware during program execution. This process is known as run-time reconfiguration (RTR). While this style of computing has the benefit of allowing for the acceleration of a greater portion of an application, it also introduces the overhead of configuration, which limits the amount of acceleration possible. Because configuration can take milliseconds or longer, rapid and efficient configuration is a critical issue. Methods such as configuration compression and the partial reuse of already programmed configurations can be used to reduce this overhead.

This article presents a survey of current research in hardware and software systems for reconfigurable computing, as well as techniques that specifically target run-time reconfigurability. We lead off this discussion by examining the technology required for reconfigurable computing, followed by a more in-depth examination of the various hardware structures used in reconfigurable systems. Next, we look at the software required for compilation of algorithms to configurable computers, and the trade-offs between hand-mapping and automatic compilation. Finally, we discuss run-time reconfigurable systems, which further utilize the intrinsic flexibility of configurable computing platforms by optimizing the hardware not only for different applications, but for different operations within a single application as well.

This survey does not seek to cover every technique and research project in the area of reconfigurable computing. Instead, it hopes to serve as an introduction to this rapidly evolving field, bringing interested readers quickly up to speed on developments from the last half-decade. Those interested in further background can find coverage of older techniques and systems elsewhere [Rose et al. 1993; Hauck and Agarwal 1996; Vuillemin et al. 1996; Mangione-Smith et al. 1997; Hauck 1998b].

2. TECHNOLOGY

Reconfigurable computing as a concept has been in existence for quite some time [Estrin et al. 1963]. Even general-purpose processors use some of the same basic ideas, such as reusing computational components for independent computations, and using multiplexers to control the routing between these components. However, the term *reconfigurable computing*, as it is used in current research (and within this survey), refers to systems incorporating some form of hardware programmability—customizing how the hardware is used using a number

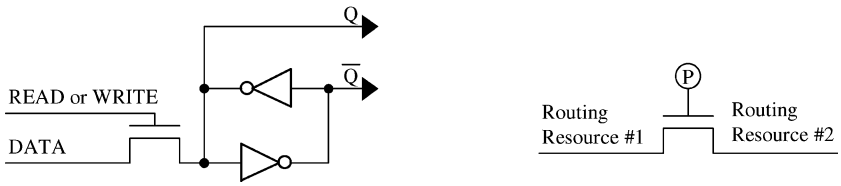

Fig. 1. A programming bit for SRAM-based FPGAs [Xilinx 1994] (left) and a programmable routing connection (right).

of physical control points. These control points can then be changed periodically in order to execute different applications using the same hardware.

The recent advances in reconfigurable computing are for the most part derived from the technologies developed for FPGAs in the mid-1980s. FPGAs were originally created to serve as a hybrid device between PALs and Mask-Programmable Gate Arrays (MPGAs). Like PALs, FPGAs are fully electrically programmable, meaning that the physical design costs are amortized over multiple application circuit implementations, and the hardware can be customized nearly instantaneously. Like MPGAs, they can implement very complex computations on a single chip, with devices currently in production containing the equivalent of over a million gates. Because of these features, FPGAs had been primarily viewed as glue-logic replacement and rapid-prototyping vehicles. However, as we show throughout this article, the flexibility, capacity, and performance of these devices has opened up completely new avenues in high-performance computation, forming the basis of reconfigurable computing.

Most current FPGAs and reconfigurable devices are SRAM-programmable (Figure 1 left), meaning that SRAM[1] bits are connected to the configuration points in the FPGA, and programming the SRAM bits configures the FPGA. Thus, these chips can be programmed and reprogrammed about as easily as a standard static RAM. In fact, one research project, the PAM project [Vuillemin et al. 1996], considers a group of one or more FPGAs to be a RAM unit that performs computation between the memory write (sending the configuration information and input data) and memory read (reading the results of the computation). This leads some to use the term *Programmable Active Memory* or *PAM*.

One example of how the SRAM configuration points can be used is to control routing within a reconfigurable device [Chow et al. 1999a]. To configure the routing on an FPGA, typically a passgate structure is employed (see Figure 1 right). Here the programming bit will turn on a routing connection when it is configured with a true value, allowing a signal to flow from one wire to another, and will disconnect these resources when the bit is set to false. With a proper interconnection of these elements, which may include millions of routing choice points within a single device, a rich routing fabric can be created.

Another example of how these configuration bits may be used is to control multiplexers, which will choose between the output of different logic resources within the array. For example, to provide optional stateholding elements a D flip-flop (DFF) may be included with a multiplexer selecting whether to forward the latched or unlatched signal value (see Figure 2 left). Thus, for systems that require stateholding the programming bits controlling the multiplexer would be configured to select the DFF output, while systems that do not need this function would choose the bypass route that sends the input directly to the output. Similar structures

[1] The term "SRAM" is technically incorrect for many FPGA architectures, given that the configuration memory may or may not support random access. In fact, the configuration memory tends to be continually read in order to perform its function. However, this is the generally accepted term in the field and correctly conveys the concept of static volatile memory using an easily understandable label.

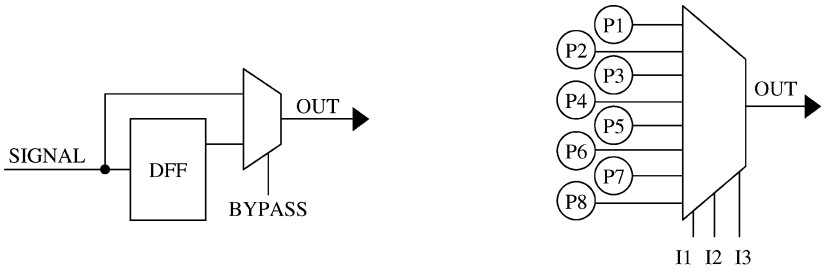

Fig. 2. D flip-flop with optional bypass (left) and a 3-input LUT (right).

can choose between other on-chip functionalities, such as fixed-logic computation elements, memories, carry chains, or other functions.

Finally, the configuration bits may be used as control signals for a computational unit or as the basis for computation itself. As a control signal, a configuration bit may determine whether an ALU performs an addition, subtraction, or other logic computations. On the other hand, with a structure such as a lookup table (LUT), the configuration bits themselves form the result of the computation (see Figure 2 right). These elements are essentially small memories provided for computing arbitrary logic functions. LUTs can compute any function of N inputs (where N is the number of control signals for the LUT's multiplexer) by programming the $2N$ programming bits with the truth table of the desired function. Thus, if all programming bits except the one corresponding to the input pattern 111 were set to zero a 3-input LUT would act as a 3-input AND gate, while programming it with all ones except in 000 would compute a NAND.

3. HARDWARE

Reconfigurable computing systems use FPGAs or other programmable hardware to accelerate algorithm execution by mapping compute-intensive calculations to the reconfigurable substrate. These hardware resources are frequently coupled with a general-purpose microprocessor that is responsible for controlling the reconfigurable logic and executing program code that cannot be efficiently accelerated. In very closely coupled systems, the reconfigurability lies within customizable functional units on the regular datapath of the microprocessor. On the other hand, a reconfigurable computing system can be as loosely coupled as a networked standalone unit. Most reconfigurable systems are categorized somewhere between these two extremes, frequently with the reconfigurable hardware acting as a coprocessor to a host microprocessor. The programmable array itself can be comprised of one or more commercially available FPGAs, or can be a custom device designed specifically for reconfigurable computing.

The design of the actual computation blocks within the reconfigurable hardware varies from system to system. Each unit of computation, or logic block, can be as simple as a 3-input lookup table (LUT), or as complex as a 4-bit ALU. This difference in block size is commonly referred to as the *granularity* of the logic block, where the 3-bit LUT is an example of a very fine-grained computational element, and a 4-bit ALU is an example of a quite coarse-grained unit. The finer-grained blocks are useful for bit-level manipulations, while the coarse-grained blocks are better optimized for standard datapath applications. Some architectures employ different sizes or types of blocks within a single reconfigurable array in order to efficiently support different types of computation. For example, memory is frequently embedded within the reconfigurable hardware to provide temporary data storage, forming a heterogeneous structure composed of both logic blocks and memory blocks [Ebeling et al. 1996; Altera 1998; Lucent 1998; Marshall et al. 1999; Xilinx 1999].

The routing between the logic blocks within the reconfigurable hardware is also of great importance. Routing contributes significantly to the overall area of the reconfigurable hardware. Yet, when the percentage of logic blocks used in an FPGA becomes very high, automatic routing tools frequently have difficulty achieving the necessary connections between the blocks. Good routing structures are therefore essential to ensure that a design can be successfully placed and routed onto the reconfigurable hardware.

Once a circuit has been programmed onto the reconfigurable hardware, it is ready to be used by the host processor during program execution. The run-time operation of a reconfigurable system occurs in two distinct phases: configuration and execution. The programming of the reconfigurable hardware is under the control of the host processor. This host processor directs a stream of configuration data to the reconfigurable hardware, and this configuration data is used to define the actual operation of the hardware. Configurations can be loaded solely at start-up of a program, or periodically during runtime, depending on the design of the system. More concepts involved in run-time reconfiguration (the dynamic reconfiguration of devices during computation execution) are discussed in a later section.

The actual execution model of the reconfigurable hardware varies from system to system. For example, the NAPA system [Rupp et al. 1998] by default suspends the execution of the host processor during execution on the reconfigurable hardware. However, simultaneous computation can occur with the use of fork-and-join primitives, similar to multiprocessor programming. REMARC [Miyamori and Olukotun 1998] is a reconfigurable system that uses a pipelined set of execution phases within the reconfigurable hardware. These pipeline stages overlap with the pipeline stages of the host processor, allowing for simultaneous execution. In the Chimaera system [Hauck et al. 1997], the reconfigurable hardware is constantly executing based upon the input values held in a subset of the host processor's registers. A call to the Chimaera unit is in actuality only a fetch of the result value. This value is stable and valid after the correct input values have been written to the registers and have filtered through the computation.

In the next sections, we consider in greater depth the hardware issues in reconfigurable computing, including both logic and routing. To support the computation demands of reconfigurable computing, we consider the logic block architectures of these devices, including possibly the integration of heterogeneous logic resources within a device. Heterogeneity also extends between chips, where one of the most important concerns is the coupling of the reconfigurable logic with standard, general-purpose processors. However, reconfigurable devices are more than just logic devices; the routing resources are at least as important as logic resources, and thus we consider interconnect structures, including 1D-oriented devices that are beginning to appear.

3.1. Coupling

Frequently, reconfigurable hardware is coupled with a traditional microprocessor. Programmable logic tends to be inefficient at implementing certain types of operations, such as variable-length loops and branch control. In order to run an application in a reconfigurable computing system most efficiently, the areas of the program that cannot be easily mapped to the reconfigurable logic are executed on a host microprocessor. Meanwhile, the areas with a high density of computation that can benefit from implementation in hardware are mapped to the reconfigurable logic. For the systems that use a microprocessor in conjunction with reconfigurable logic, there are several ways in which these two computation structures may be coupled, as Figure 3 shows.

First, reconfigurable hardware can be used solely to provide reconfigurable functional units within a host processor [Razdan and Smith 1994; Hauck et al. 1997]. This allows for a traditional programming environment with the

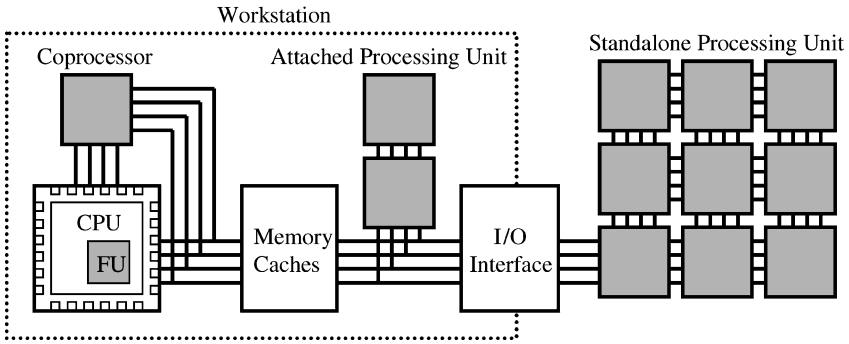

Fig. 3. Different levels of coupling in a reconfigurable system. Reconfigurable logic is shaded.

addition of custom instructions that may change over time. Here, the reconfigurable units execute as functional units on the main microprocessor datapath, with registers used to hold the input and output operands.

Second, a reconfigurable unit may be used as a coprocessor [Wittig and Chow 1996; Hauser and Wawrzynek 1997; Miyamori and Olukotun 1998; Rupp et al. 1998; Chameleon 2000]. A coprocessor is, in general, larger than a functional unit, and is able to perform computations without the constant supervision of the host processor. Instead, the processor initializes the reconfigurable hardware and either sends the necessary data to the logic, or provides information on where this data might be found in memory. The reconfigurable unit performs the actual computations independently of the main processor, and returns the results after completion. This type of coupling allows the reconfigurable logic to operate for a large number of cycles without intervention from the host processor, and generally permits the host processor and the reconfigurable logic to execute simultaneously. This reduces the overhead incurred by the use of the reconfigurable logic, compared to a reconfigurable functional unit that must communicate with the host processor each time a reconfigurable "instruction" is used. One idea that is somewhat of a hybrid between the first and second coupling methods, is the use of programmable hardware within a configurable cache [Kim et al. 2000]. In this situation, the reconfigurable logic is embedded into the data cache. This cache can then be used as either a regular cache or as an additional computing resource depending on the target application.

Third, an attached reconfigurable processing unit [Vuillemin et al. 1996; Annapolis 1998; Laufer et al. 1999] behaves as if it is an additional processor in a multiprocessor system or an additional compute engine accessed semifrequently through external I/O. The host processor's data cache is not visible to the attached reconfigurable processing unit. There is, therefore, a higher delay in communication between the host processor and the reconfigurable hardware, such as when communicating configuration information, input data, and results. This communication is performed though specialized primitives similar to multiprocessor systems. However, this type of reconfigurable hardware does allow for a great deal of computation independence, by shifting large chunks of a computation over to the reconfigurable hardware.

Finally, the most loosely coupled form of reconfigurable hardware is that of an external stand-alone processing unit [Quickturn 1999a, 1999b]. This type of reconfigurable hardware communicates infrequently with a host processor (if present). This model is similar to that of networked workstations, where processing may occur for very long periods of time without a great deal of communication. In the case of the Quickturn systems, however, this hardware is geared

more towards emulation than reconfigurable computing.

Each of these styles has distinct benefits and drawbacks. The tighter the integration of the reconfigurable hardware, the more frequently it can be used within an application or set of applications due to a lower communication overhead. However, the hardware is unable to operate for significant portions of time without intervention from a host processor, and the amount of reconfigurable logic available is often quite limited. The more loosely coupled styles allow for greater parallelism in program execution, but suffer from higher communications overhead. In applications that require a great deal of communication, this can reduce or remove any acceleration benefits gained through this type of reconfigurable hardware.

3.2. Traditional FPGAs

Before discussing the detailed architecture design of reconfigurable devices in general, we will first describe the logic and routing of FPGAs. These concepts apply directly to reconfigurable systems using commercial FPGAs, such as PAM [Vuillemin et al. 1996] and Splash 2 [Arnold et al. 1992; Buell et al. 1996], and many also extend to architectures designed specifically for reconfigurable computing. Hardware concepts applying specifically to architectures designed for reconfigurable computing, as well as variations on the generic FPGA description provided here, are discussed following this section. More detailed surveys of FPGA architectures themselves can be found elsewhere [Brown et al. 1992a; Rose et al. 1993].

Since the introduction of FPGAs in the mid-1980s, there have been many different investigations into what computation element(s) should be built into the array [Rose et al. 1993]. One could consider FPGAs that were created with PAL-like product term arrays, or multiplexer-based functionality, or even basic fixed functions such as simple NAND and XOR gates. In fact, many such architectures have been built. However, it seems to be fairly well established that the best function block for a standard FPGA, a device whose primary role is the implementation of random digital logic, is the one found in the first devices deployed—the lookup table (Figure 2 right). As described in the previous section, an N-input LUT is basically a memory that, when programmed appropriately, can compute any function of up to N inputs. This flexibility, with relatively simple routing requirements (each input need only be routed to a single multiplexer control input) turns out to be very powerful for logic implementation. Although it is less area-efficient than fixed logic blocks, such as a standard NAND gate, the truth is that most current FPGAs use less than 10% of their chip area for logic, devoting the majority of the silicon real estate for routing resources.

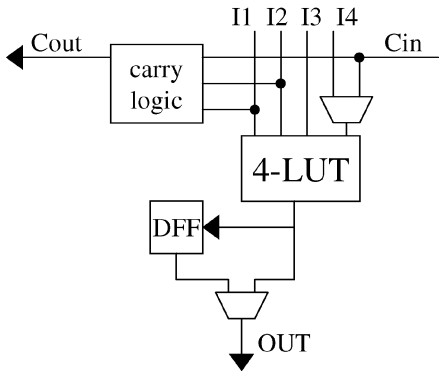

Fig. 4. A basic logic block, with a 4-input LUT, carry chain, and a D-type flip-flop with bypass.

The typical FPGA has a logic block with one or more 4-input LUT(s), optional D flip-flops (DFF), and some form of fast carry logic (Figure 4). The LUTs allow any function to be implemented, providing generic logic. The flip-flop can be used for pipelining, registers, stateholding functions for finite state machines, or any other situation where clocking is required. Note that the flip-flops will typically include programmable set/reset lines and clock signals, which may come from global signals routed on special resources, or could be routed via the standard interconnect structures from some other input or logic block. The fast carry logic

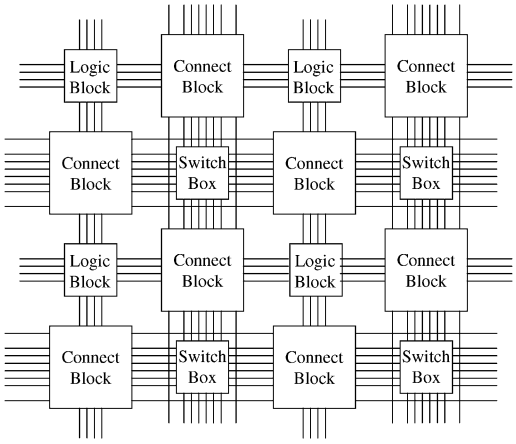

Fig. 5. A generic island-style FPGA routing architecture.

is a special resource provided in the cell to speed up carry-based computations, such as addition, parity, wide AND operations, and other functions. These resources will bypass the general routing structure, connecting instead directly between neighbors in the same column. Since there are very few routing choices in the carry chain, and thus less delay on the computation, the inclusion of these resources can significantly speed up carry-based computations.

Just as there has been a great deal of experimentation in FPGA logic block architectures, there has been equally as much investigation into interconnect structures. As logic blocks have basically standardized on LUT-based structures, routing resources have become primarily island-style, with logic surrounded by general routing channels.

Most FPGA architectures organize their routing structures as a relatively smooth sea of routing resources, allowing fast and efficient communication along the rows and columns of logic blocks. As shown in Figure 5, the logic blocks are embedded in a general routing structure, with input and output signals attaching to the routing fabric through connection blocks. The connection blocks provide programmable multiplexers, selecting which of the signals in the given routing channel will be connected to the logic block's terminals. These blocks also connect shorter local wires to longer-distance routing resources. Signals flow from the logic block into the connection block, and then along longer wires within the routing channels. At the switchboxes, there are connections between the horizontal and vertical routing resources to allow signals to change their routing direction. Once the signal has traversed through routing resources and intervening switchboxes, it arrives at the destination logic block through one of its local connection blocks. In this manner, relatively arbitrary interconnections can be achieved between the logic blocks in the system.

Within a given routing channel, there may be a number of different lengths of routing resources. Some local interconnections may only move between adjacent logic blocks (carry chains are a good example of this), providing high-speed local interconnect. Medium length lines may run the width of several logic blocks, providing for some longer distance interconnect. Finally, longlines that run the entire chip width or height may provide for more global signals. Also, many architectures contain special "global lines" that provide high-speed, and often low-skew, connections to all of the logic blocks in the array. These are primarily used for clocks, resets, and other truly global signals.

While the routing architecture of an FPGA is typically quite complex—the connection blocks and switchboxes surrounding a single logic block typically have thousands of programming points—they are designed to be able to support fairly arbitrary interconnection patterns. Most users ignore the exact details of these architectures and allow the automatic physical design tools to choose appropriate resources to use in order to achieve a given interconnect pattern.

3.3. Logic Block Granularity

Most reconfigurable hardware is based upon a set of computation structures that are repeated to form an array. These structures, commonly called *logic blocks* or *cells*, vary in complexity from a very

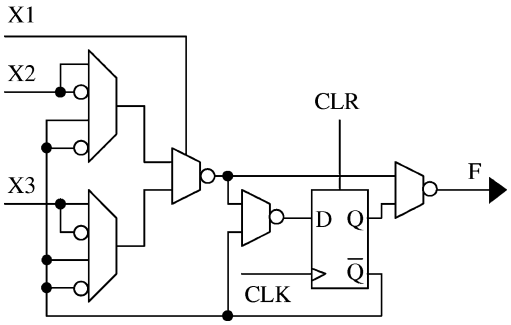

Fig. 6. The functional unit from a Xilinx 6200 cell [Xilinx 1996].

small and simple block that can calculate a function of only three inputs, to a structure that is essentially a 4-bit ALU. Some of these block types are configurable, in that the actual operation is determined by a set of loaded configuration data. Other blocks are fixed structures, and the configurability lies in the connections between them. The size and complexity of the basic computing blocks is referred to as the block's *granularity*.

An example of a very fine-grained logic block can be found in the Xilinx 6200 series of FPGAs [Xilinx 1996]. The functional unit from one of these cells, as shown in Figure 6, can implement any two-input function and some three-input functions. However, although this type of architecture is useful for very fine-grained bit manipulation, it can be too fine-grained to efficiently implement many types of circuits, such as multipliers. Similarly, finite state machines are frequently too complex to easily map to a reasonable number of very fine-grained logic blocks. However, finite state machines are also too dependent upon single bit values to be efficiently implemented in a very coarse-grained architecture. This type of circuit is more suited to an architecture that provides more connections and computational power per logic block, yet still provides sufficient capability for bit-level manipulation.

The logic cell in the Altera FLEX 10K architecture [Altera 1998] is a fine-grained structure that is somewhat coarser than the 6200. This architecture mainly consists of a single 4-input LUT with a flip-flop. Additionally, there is specialized carry-chain circuitry that helps to accelerate addition, parity, and other operations that use a carry chain. These types of logic blocks are useful for fine-grained bit-level manipulation of data, as can frequently be found in encryption and image processing applications. Also, because the cells are fine-grained, computation structures of arbitrary bit widths can be created. This can be useful for implementing datapath circuits that are based on data widths not implemented on the host processor (5 bit multiply, 18 bit addition, etc). Reconfigurable hardware can not only take advantage of small bit widths, but also large data widths. When a program uses bit widths in excess of what is normally available in a host processor, the processor must perform the computations using a number of extra steps in order to handle the full data width. A fine-grained architecture would be able to implement the full bit width in a single step, without the fetching, decoding, and execution of additional instructions, as long as enough logic cells are available.

A number of reconfigurable systems use a granularity of logic block that we categorize as medium-grained [Xilinx 1994; Hauser and Wawrzynek 1997; Haynes and Cheung 1998; Lucent 1998; Marshall et al. 1999]. For example, Garp [Hauser and Wawrzynek 1997] is designed to perform a number of different operations on up to four 2-bit inputs. Another medium-grained structure was designed specifically to be embedded inside of a general-purpose FPGA to implement multipliers of a configurable bit width [Haynes and Cheung 1998]. The logic block used in the multiplier FPGA is capable of implementing a 4×4 multiplication, or cascaded into larger structures. The CHESS architecture [Marshall et al. 1999] also operates on 4-bit values, with each of its cells acting as a 4-bit ALU. Medium-grained logic blocks may be used to implement datapath circuits of varying bit widths, similar to the fine-grained structures. However, with the ability to perform more complex operations of a greater number of inputs, this type of structure can be used efficiently to implement a wider variety of operations.

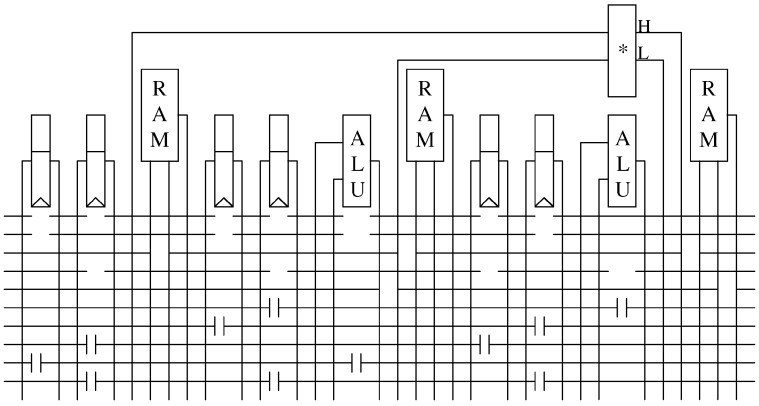

Fig. 7. One cell in the RaPiD-I reconfigurable architecture [Ebeling et al. 1996]. The registers, RAM, ALUs, and multiplier all operate on 16-bit values. The multiplier outputs a 32-bit result, split into the high 16 bits and the low 16 bits. All routing lines shown are 16-bit wide busses. The short parallel lines on the busses represent configurable bus connectors.

Very coarse-grained architectures are primarily intended for the implementation of word-width datapath circuits. Because the logic blocks used are optimized for large computations, they will perform these operations much more quickly (and consume less chip area) than a set of smaller cells connected to form the same type of structure. However, because their composition is static, they are unable to leverage optimizations in the size of operands. For example, the RaPiD architecture [Ebeling et al. 1996], shown in Figure 7, as well as the Chameleon architecture [Chameleon 2000], are examples of this very coarse-grained type of design. Each of these architectures is composed of word-sized adders, multipliers, and registers. If only three 1-bit values are required, then the use of these architectures suffers an unnecessary area and speed overhead, as all of the bits in the full word size are computed. However, these coarse-grained architectures can be much more efficient than fine-grained architectures for implementing functions closer to their basic word size.

An alternate form of a coarse-grained system is one in which the logic blocks are actually very small processors, potentially each with its own instruction memory and/or data values. The REMARC architecture [Miyamori and Olukotun 1998] is composed of an 8×8 array of 16-bit processors. Each of these processors uses its own instruction memory in conjunction with a global program counter. This style of architecture closely resembles a single-chip multiprocessor, although with much simpler component processors because the system is intended to be coupled with a host processor. The RAW project [Moritz et al. 1998] is a further example of a reconfigurable architecture based on a multiprocessor design.

The granularity of the FPGA also has a potential effect on the reconfiguration time of the device. This is an important issue for run-time reconfiguration, which is discussed in further depth in a later section. A fine-grained array has many configuration points to perform very small computations, and thus requires more data bits during configuration.

3.4. Heterogeneous Arrays

In order to provide greater performance or flexibility in computation, some reconfigurable systems provide a heterogeneous structure, where the capabilities of the logic cells are not the same throughout the system. One use of heterogeneity in reconfigurable systems is to provide multiplier function blocks embedded within the reconfigurable hardware [Haynes and

Cheung 1998; Chameleon 2000; Xilinx 2001]. Because multiplication is one of the more difficult computations to implement efficiently in a traditional FPGA structure, the custom multiplication hardware embedded within a reconfigurable array allows a system to perform even that function well.

Another use of heterogeneous structures is to provide embedded memory blocks scattered throughout the reconfigurable hardware. This allows storage of frequently used data and variables, and allows for quick access to these values due to the proximity of the memory to the logic blocks that access it. Memory structures embedded into the reconfigurable fabric come in two forms. The first is simply the use of available LUTs as RAM structures, as can be done in the Xilinx 4000 series [Xilinx 1994] and Virtex [Xilinx 1999] FPGAs. Although making these very small blocks into a larger RAM structure introduces overhead to the memory system, it does provide local, variable width memory structures.

Some architectures include dedicated memory blocks within their array, such as the Xilinx Virtex series [Xilinx 1999, 2001] and Altera [Altera 1998] FPGAs, as well as the CS2000 RCP (reconfigurable communications processor) device from Chameleon Systems, Inc. [Chameleon 2000]. These memory blocks have greater performance in large sizes than similar-sized structures built from many small LUTs. While these structures are somewhat less flexible than the LUT-based memories, they can also provide some customization. For example, the Altera FLEX 10K FPGA [Altera 1998] provides embedded memories that have a limited total number of wires, but allow a trade-off between the number of address lines and the data bit width.

When embedded memories are not used for data storage by a particular configuration, the area that they occupy does not necessarily have to be wasted. By using the address lines of the memory as function inputs and the values stored in the memory as function outputs, logical expressions of a large number of inputs can be emulated [Altera 1998; Cong and Xu 1998; Wilton 1998; Heile and Leaver 1999]. In fact, because there may be more than one value output from the memory on a read operation, the memory structure may be able to perform multiple different computations (one for each bit of data output), provided that all necessary inputs appear on the address lines. In this manner, the embedded RAM behaves the same as a very large LUT. Therefore, embedded memory allows a programmer or a synthesis tool to perform a trade-off between logic and memory usage in order to achieve higher area efficiency.

Furthermore, a few of the commercial FPGA companies have announced plans to include entire microprocessors as embedded structures within their FPGAs. Altera has demonstrated a preliminary ARM9-based Excalibur device, which combines reconfigurable hardware with an embedded ARM9 processor core [Altera 2001]. Meanwhile, Xilinx is working with IBM to include a PowerPC processor core within the Virtex-II FPGA [Xilinx 2000]. By contrast, Adaptive Silicon's focus is to provide reconfigurable logic cores to customers for embedding in their own system-on-a-chip (SoC) devices [Adaptive 2001].

3.5. Routing Resources

Interconnect resources are provided in a reconfigurable architecture to connect together the device's programmable logic elements. These resources are usually configurable, where the path of a signal is determined at compile or run-time rather than fabrication time. This flexible interconnect between logic blocks or computational elements allows for a wide variety of circuit structures, each with their own interconnect requirements, to be mapped to the reconfigurable hardware. For example, the routing for FPGAs is generally island-style, with logic surrounded by routing channels, which contain several wires, potentially of varying lengths. Within this type of routing architecture, however, there are still variations. Some of these differences include the ratio of wires to logic in the system, how long each of the

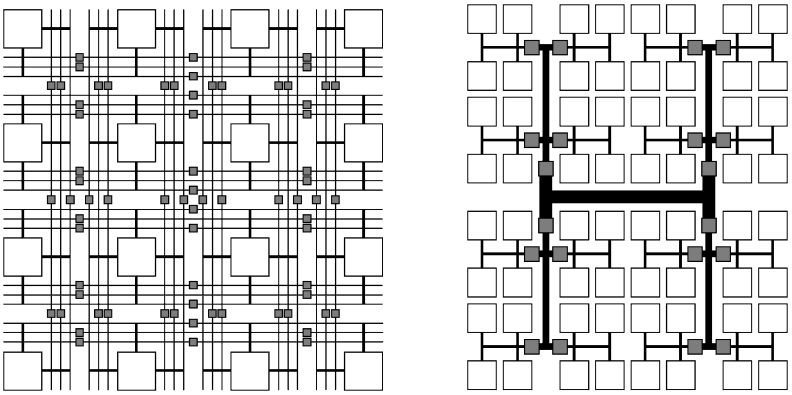

Fig. 8. Segmented (left) and hierarchical (right) routing structures. The white boxes are logic blocks, while the dark boxes are connection switches.

wires should be, and whether they should be connected in a segmented or hierarchical manner.

A step in the design of efficient routing structures for FPGAs and reconfigurable systems therefore involves examining the logic vs. routing area trade-off within reconfigurable architectures. One group has argued that the interconnect should constitute a much higher proportion of area in order to allow for successful routing under high-logic utilization conditions [Takahara et al. 1998]. However, for FPGAs, high-LUT utilization may not necessarily be the most desirable situation, but rather efficient routing usage may be of more importance [DeHon 1999]. This is because the routing resources occupy a much larger part of the area of an FPGA than the logic resources, and therefore the most area efficient designs will be those that optimize their use of the routing resources rather than the logic resources. The amount of required routing does not grow linearly with the amount of logic present; therefore, larger devices require even greater amounts of routing per logic block than small ones [Trimberger et al. 1997b].

There are two primary methods to provide both local and global routing resources, as shown in Figure 8. The first is the use of segmented routing [Betz and Rose 1999; Chow et al. 1999a]. In segmented routing, short wires accommodate local communications traffic. These short wires can be connected together using switchboxes to emulate longer wires. Frequently, segmented routing structures also contain longer wires to allow signals to travel efficiently over long distances without passing through a great number of switches. Hierarchical routing [Aggarwal and Lewis 1994; Lai and Wang 1997; Tsu et al. 1999] is the second method to provide both local and global communication. Routing within a group (or cluster) of logic blocks is at the local level, only connecting within that cluster. At the boundaries of these clusters, however, longer wires connect the different clusters together. This is potentially repeated at a number of levels. The idea behind the use of hierarchical structures is that, provided a good placement has been made onto the hardware, most communication should be local and only a limited amount of communication will traverse long distances. Therefore, the wiring is designed to fit this model, with a greater number of local routing wires in a cluster than distance routing wires between clusters.

Because routing can occupy a large part of the area of a reconfigurable device, the type of routing used must be carefully considered. If the wires available are much longer than what is required to route a signal, the excess wire length is wasted. On the other hand, if the wires available are much shorter than necessary, the signal

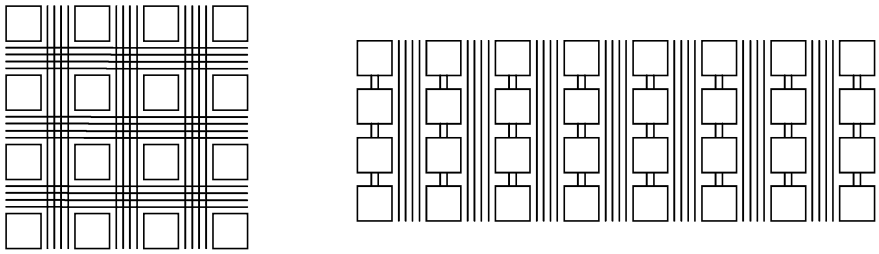

Fig. 9. A traditional two-dimensional island-style routing structure (left) and a one-dimensional routing structure (right). The white boxes represent logic elements.

must pass through switchboxes that connect the short wires together into a longer wire, or through levels of the routing hierarchy. This induces additional delay and slows the overall operation of the circuit. Furthermore, the switchbox circuitry occupies area that might be better used for additional logic or wires.

There are a few alternatives to the island-style of routing resources. Systems such as RaPiD [Ebeling et al. 1996] use segmented bus-based routing, where signals are full word-sized in width. This is most common in the one-dimensional type of architecture, as discussed in the next section.

3.6. One-Dimensional Structures

Most current FPGAs are of the two-dimensional variety, as shown in Figure 9. This allows for a great deal of flexibility, as any signal can be routed on a nearly arbitrary path. However, providing this level of routing flexibility requires a great deal of routing area. It also complicates the placement and routing software, as the software must consider a very large number of possibilities.

One solution is to use a more one-dimensional style of architecture, also depicted in Figure 9. Here, placement is restricted along one axis. With a more limited set of choices, the placement can be performed much more quickly. Routing is also simplified, because it is generally along a single dimension as well, with the other dimension generally only used for calculations requiring a shift operation. One drawback of the one-dimensional routing is that if there are not enough routing resources in a particular area of a mapped circuit, routing that circuit becomes actually more difficult than on a two-dimensional array that provides more alternatives. A number of different reconfigurable systems have been designed in this manner. Both Garp [Hauser and Wawrzynek 1997] and Chimaera [Hauck et al. 1997] are structures that provide cells that compute a small number of bit positions, and a row of these cells together computes the full data word. A row can only be used by a single configuration, making these designs one dimensional. In this manner, each configuration occupies some number of complete rows. Although multiple narrow-width computations can fit within a single row, these structures are optimized for word-based computations that occupy the entire row. The NAPA architecture [Rupp et al. 1998] is similar, with a full column of cells acting as the atomic unit for a configuration, as is PipeRench [Cadambi et al. 1998; Goldstein et al. 2000].

In some systems, the computation blocks in a one-dimensional structure operate on word-width values instead of single bits. Therefore, busses are routed instead of individual values. This also decreases the time required for routing, as the bits of a bus can be considered together rather than as separate routes. As shown previously in Figure 7, RaPiD [Ebeling et al. 1996] is basically a one-dimensional design that only includes word-width processing elements. The different computation units are organized in a single dimension along the horizontal

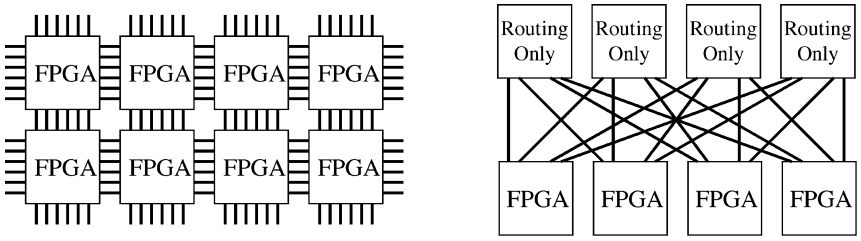

Fig. 10. Mesh (left) and partial crossbar (right) interconnect topologies for multi-FPGA systems.

axis. The general flow of information follows this layout, with the major routing busses also laid out in a horizontal manner. Additionally, all routing is of word-sized values, and therefore all routing is of busses, not individual wires. A few vertical resources are included in the architecture to allow signals to transfer between busses, or to travel from a bus to a computation node. However, the majority of the routing in this architecture is one-dimensional.

3.7. Multi-FPGA Systems

Reconfigurable systems that are composed of multiple FPGA chips interconnected on a single processing board have additional hardware concerns over single-chip systems. In particular, there is a need for an efficient connection scheme between the chips, as well as to external memory and the system bus. This is to provide for circuits that are too large to fit within a single FPGA, but may be partitioned over the multiple FPGAs available. A number of different interconnection schemes have been explored [Butts and Batcheller 1991; Hauck et al. 1998a; Hauck 1998; Khalid 1999] including meshes and crossbars, as shown in Figure 10. A mesh connects the nearest-neighbors in the array of FPGA chips. This allows for efficient communication between the neighbors, but may require that some signals pass through an FPGA simply to create a connection between non-neighbors. Although this can be done, and is quite possible, it uses valuable I/O resources on the FPGA that forms the routing bridge. One system that uses a mesh topology with additional board-level column and row busses is the P1 system developed within the PAM project [Vuillemin et al. 1996]. This architecture uses a central array of 16 commercial FPGAs with connections to nearest-neighbors. However, four 16-bit row busses and four 16-bit column busses run the length of the array and facilitate communication between non-neighbor FPGAs.

A crossbar attempts to remove this problem by using special routing-only chips to connect each FPGA potentially to any other FPGA. The inter-chip delays are more uniform, given that a signal travels the exact same "distance" to get from one FPGA to another, regardless of where those FPGAs are located. However, a crossbar interconnect does not scale easily with an increase in the number of FPGAs. The crossbar pattern of the chips is fixed at fabrication of the multi-FPGA board. Variants on these two basic topologies attempt to remove some of the problems encountered in mesh and crossbar topologies [Arnold et al. 1992; Varghese et al. 1993; Buell et al. 1996; Vuillemin et al. 1996; Lewis et al. 1997; Khalid and Rose 1998]. One of these variants can be found in the Splash 2 system [Arnold et al. 1992; Buell et al. 1996]. The predecessor, Splash 1, used a linear systolic communication method. This type of connection was found to work quite well for a variety of applications. However, this highly constrained communication model made some types of computations difficult or even impossible. Therefore, Splash 2 was designed to include not only the linear connections of Splash 1 that were found to be useful for many applications, but also a crossbar network to allow any FPGA

to communicate with any other FPGA on the same board. For multi-FPGA systems, because of the need for efficient communication between the FPGAs, determining the inter-chip routing topology is a very important step in the design process. More details on multi-FPGA system architectures can be found elsewhere [Hauck 1998b; Khalid 1999].

3.8. Hardware Summary

The design of reconfigurable hardware varies wildly from system to system. The reconfigurable logic may be used as a configurable functional unit, or may be a multi-FPGA stand-alone unit. Within the reconfigurable logic itself, the complexity of the core computational units, or logic blocks, vary from very simple to extremely complex, some implementing a 4-bit ALU or even a 16 $\times$ 16 multiplication. These blocks are not required to be uniform throughout the array, as the use of different types of blocks can add high-performance functionality in the case of specialized computation circuitry, or expanded storage in the case of embedded memory blocks. Routing resources also offer a variety of choices, primarily in amount, length, and organization of the wires. Systems have been developed that fit into many different points within this design space, and no true "best" system has yet been agreed upon.

4. SOFTWARE

Although reconfigurable hardware has been shown to have significant performance benefits for some applications, it may be ignored by application programmers unless they are able to easily incorporate its use into their systems. This requires a software design environment that aids in the creation of configurations for the reconfigurable hardware. This software can range from a software assist in manual circuit creation to a complete automated circuit design system. Manual circuit description is a powerful method for the creation of high-quality circuit designs. However, it requires a great deal of background knowledge of the particular

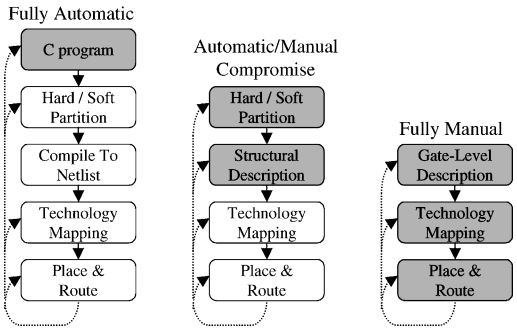

Fig. 11. Three possible design flows for algorithm implementation on a reconfigurable system. Grey stages indicate manual effort on the part of the designer, while white stages are done automatically. The dotted lines represent paths to improve the resulting circuit. It should be noted that the middle design cycle is only one of the possible compromises between automatic and manual design.

reconfigurable system employed, as well as a significant amount of design time. On the other end of the spectrum, an automatic compilation system provides a quick and easy way to program for reconfigurable systems. It therefore makes the use of reconfigurable hardware more accessible to general application programmers, but quality may suffer.

Both for manual and automatic circuit creation, the design process proceeds through a number of distinct phases, as indicated in Figure 11. Circuit specification is the process of describing the functions that are to be placed on the reconfigurable hardware. This can be done as simply as by writing a program in C that represents the functionality of the algorithm to be implemented in hardware. On the other hand, this can also be as complex as specifying the inputs, outputs, and operation of each basic building block in the reconfigurable system. Between these two methods is the specification of the circuit using generic complex components, such as adders and multipliers, which will be mapped to the actual hardware later in the design process. For descriptions in a high-level language (HLL), such as C/C++ or Java, or ones using complex building blocks, this code must be compiled into a netlist of gate-level components. For the HLL implementations, this involves

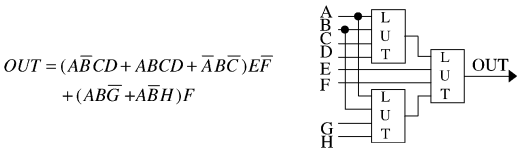

Fig. 12. A wide function implemented with multiple LUTs.

generating computational components to perform the arithmetic and logic operations within the program, and separate structures to handle the program control, such as loop iterations and branching operations. Given a structural description, either generated from a HLL or specified by the user, each complex structure is replaced with a network of the basic gates that perform that function.

Once a detailed gate- or element-level description of the circuit has been created, these structures must be translated to the actual logic elements of the reconfigurable hardware. This stage is known as technology mapping, and is dependent upon the exact target architecture. For a LUT-based architecture, this stage partitions the circuit into a number of small subfunctions, each of which can be mapped to a single LUT [Brown et al. 1992a; Abouzeid et al. 1993; Sangiovanni-Vincentelli et al. 1993; Hwang et al. 1994; Chang et al. 1996; Hauck and Agarwal 1996; Yi and Jhon 1996; Chowdhary and Hayes 1997; Lin et al. 1997; Cong and Wu 1998; Pan and Lin 1998; Togawa et al. 1998; Cong et al. 1999]. Some architectures, such as the Xilinx 4000 series [Xilinx 1994], contain multiple LUTs per logic cell. These LUTs can be used either separately to generate small functions, or together to generate some wider-input functions [Inuani and Saul 1997; Cong and Hwang 1998]. By taking advantage of multiple LUTs and the internal routing within a single logic cell, functions with more inputs than can be implemented using a single LUT can efficiently be mapped into the FPGA architecture. Figure 12 shows one example of a wide function mapped to a multi-LUT FPGA logic cell.

For reconfigurable structures that include embedded memory blocks, the mapping stage may also consider using these memories as logic units when they are not being used for data storage. The memories act as very large LUTs, where the number of inputs is equal to the number of address lines. In order to use these memories as logic, the mapping software must analyze how much of the memory blocks are actually used as storage in a given mapping. It must then determine which are available in order to implement logic, and what part or parts of the circuit are best mapped to the memory [Cong and Xu 1998; Wilton 1998].

After the circuit has been mapped, the resulting blocks must be placed onto the reconfigurable hardware. Each of these blocks is assigned to a specific location within the hardware, hopefully close to the other logic blocks with which it communicates. As FPGA capacities increase, the placement phase of circuit mapping becomes more and more time consuming. Floorplanning is a technique that can be used to alleviate some of this cost. A floorplanning algorithm first partitions the logic cells into clusters, where cells with a large amount of communication are grouped together. These clusters are then placed as units onto regions of the reconfigurable hardware. Once this global placement is complete, the actual placement algorithm performs detailed placement of the individual logic blocks within the boundaries assigned to the cluster [Sankar and Rose 1999].

The use of a floorplanning tool is particularly helpful for situations where the circuit structure being mapped is of a datapath type. Large computational components or macros that are found in datapath circuits are frequently composed of highly regular logic. These structures are placed as entire units, and their component cells are restricted to the floorplanned location [Shi and Bhatia 1997; Emmert and Bhatia 1999]. This encourages the placer to find a very regular placement of these logic cells, resulting in a higher performance layout of the circuit. Another technique for the mapping and placement of datapath elements is to perform both of these steps simultaneously [Callahan et al. 1998].

This method also exploits the regularity of the datapath elements to generate mappings and placements quickly and efficiently.

Floorplanning is also important when dealing with hierarchically structured reconfigurable designs. In these architectures, the available resources have been grouped by the logic or routing hierarchy of the hardware. Because performance is best when routing lengths are minimized, the cells to be placed should be grouped such that cells that require a great deal of communication or which are on a critical path are placed together within a logic cluster on the hardware [Krupnova et al. 1997; Senouci et al. 1998].

After floorplanning, the individual logic blocks are placed into specific logic cells. One algorithm that is commonly used is the simulated annealing technique [Shahookar and Mazumder 1991; Betz and Rose 1997; Sankar and Rose 1999]. This method takes an initial placement of the system, which can be generated (pseudo-) randomly, and performs a series of "moves" on that layout. A move is simply the changing of the location of a single logic cell, or the exchanging of locations of two logic cells. These moves are attempted one at a time using random target locations. If a move improves the layout, then the layout is changed to reflect that move. If a move is considered to be undesirable, then it is only accepted a small percentage of the time. Accepting a few "bad" moves helps to avoid any local minima in the placement space. Other algorithms exist that are not so based on random movements [Gehring and Ludwig 1996], although this searches a smaller area of the placement space for a solution, and therefore may be unable to find a solution which meets performance requirements if a design uses a high percentage of the reconfigurable resources.

Finally, the different reconfigurable components comprising the application circuit are connected during the routing stage. Particular signals are assigned to specific portions of the routing resources of the reconfigurable hardware. This can become difficult if the placement causes many connected components to be placed far from one another, as the signals that travel long distances use more routing resources than those that travel shorter ones. A good placement is therefore essential to the routing process. One of the challenges in routing for FPGAs and reconfigurable systems is that the available routing resources are limited. In general hardware design, the goal is to minimize the number of routing tracks used in a channel between rows of computation units, but the channels can be made as wide as necessary. In reconfigurable systems, however, the number of available routing tracks is determined at fabrication time, and therefore the routing software must perform within these boundaries. Thus, FPGA routing concentrates on minimizing congestion within the available tracks [Brown et al. 1992b; McMurchie and Ebeling 1995; Alexander and Robins 1996; Chan and Schlag 1997; Lee and Wu 1997; Thakur et al. 1997; Wu and Marek-Sadowska 1997; Swartz et al. 1998; Nam et al. 1999]. Because routing is one of the more time-intensive portions of the design cycle, it can be helpful to determine if a placed circuit can be routed before actually performing the routing step. This quickly informs the designer if changes need to be made to the layout or a larger reconfigurable structure is required [Wood and Rutenbar 1997; Swartz et al. 1998].

Each of the design phases mentioned above may be implemented either manually or automatically using compiler tools. The operation of some of these individual steps are described in greater depth in the following sections.

4.1. Hardware-Software Partitioning

For systems that include both reconfigurable hardware and a traditional microprocessor, the program must first be partitioned into sections to be executed on the reconfigurable hardware and sections to be executed in software on the microprocessor. In general, complex control sequences such as variable-length loops are more efficiently implemented in software,

while fixed datapath operations may be more efficiently executed in hardware.

Most compilers presented for reconfigurable systems generate only the hardware configuration for the system, rather than both hardware and software. In some cases, this is because the reconfigurable hardware may not be coupled with a host processor, so only a hardware configuration is necessary. For cases where reconfigurable hardware does operate alongside a host microprocessor, some systems currently require that the hardware compilation be performed separately from the software compilation, and special functions are called from within the software in order to configure and control the reconfigurable hardware. However, this requires effort on the part of the designer to identify the sections that should be mapped to hardware, and to translate these into special hardware functions. In order to make the use of the reconfigurable hardware transparent to the designer, the partitioning and programming of the hardware should occur simultaneously in a single programming environment.

For compilers that manage both the hardware and software aspects of application design, the hardware/software partitioning can be performed either manually, or automatically by the compiler itself. When the partitioning is performed by the programmer, compiler directives are used to mark sections of program code for hardware compilation. The NAPA C language [Gokhale and Stone 1998] provides **pragma** statements to allow a programmer to specify whether a section of code is to be executed in software on the Fixed Instruction Processor (FIP), or in hardware on the Adaptive Logic Processor (ALP). Cardoso and Neto [1999] present another compiler that requires the user to specify (using information gained through the use of profiling tools) which areas of code to map to the reconfigurable hardware.

Alternately, the hardware/software partitioning can be done automatically [Chichkov and Almeida 1997; Kress et al. 1997; Callahan et al. 2000; Li et al. 2000a]. In this case, the compiler will use cost functions based upon the amount of acceleration gained through the execution of a code fragment in hardware to determine whether the cost of configuration is overcome by the benefits of hardware execution.

4.2. Circuit Specification

In order to use the reconfigurable hardware, designers must somehow be able to specify the operation of their custom circuits. Before high-level compilation tools are developed for a specific reconfigurable system, this is done through hand mapping of the circuit, where the designer specifies the operation of the components in the configurable system directly. Here, the designers utilize the basic building blocks of the reconfigurable system to create the desired circuit. This style of circuit specification is primarily useful only when a software front-end for circuit design is unavailable, or for the design of small circuits or circuits with very high performance requirements. This is due to the great amount of time involved in manual circuit creation. However, for circuits that can be reasonably hand mapped, this provides potentially the smallest and fastest implementation.

Because not all designers can be intimately familiar with every reconfigurable architecture, some design tools abstract the specifics of the target architecture. Creating a circuit using a structural design language involves describing a circuit using building blocks such as gates, flip-flops and latches [Bellows and Hutchings 1998; Gehring and Ludwig 1998; Hutchings et al. 1999]. The compiler then maps these modules to one or more basic components of the architecture of the reconfigurable system. Structural VHDL is one example of this type of programming, and commercial tools are available for compiling from this language into vendor-specific FPGAs [Synplicity 1999].

However, these two methods require that the designer possess either an intimate knowledge of the targeted reconfigurable hardware, or at least a working knowledge of the concepts involved

in hardware design. In order to allow a greater number of software developers to take advantage of reconfigurable computing, tools that allow for behavioral circuit descriptions are being developed. These systems trade some area and performance quality for greater flexibility and ease of use.

Behavioral circuit design is similar to software design because the designer indicates the steps a hardware subsystem must go through in order to perform the desired computation rather than the actual composition of the circuit. These behavioral descriptions can be either in a generic hardware description language such as VHDL or Verilog, or a general-purpose high-level language such as C/C++ or Java. The eventual goal of this type of compilation is to allow users to write programs in commonly used languages that compile equally well, without modification, to both a traditional software executable and to an executable which leverages reconfigurable hardware.

Working towards this direction, Transmogrifier C [Galloway 1995] allows a subset of the C language to be used to describe hardware circuits. While multiplication, division, pointers, arrays, and a few other C language specifics are not supported, this system provides a behavioral method of circuit description using a primitive form of the C language. Similarly, the C++ programming environment used for the P1 system [Vuillemin et al. 1996] provides a hybrid method of description, using a combination of behavioral and structural design. Synopsys' CoCentric compiler [Synopsys 2000], which can be targeted to the Xilinx Virtex series of FPGA, uses SystemC to provide for behavioral compilation of C/C++ with the assistance of a set of additional hardware-defining classes. Other compilers, such as Nimble [Li et al. 2000a] and the Garp compiler [Callahan et al. 2000], are fully behavioral C compilers, handling the full set of the ANSI C language.

Although behavioral description, and HLL description in particular, provides a convenient method for the programming of reconfigurable systems, it does suffer from the drawback that it tends to produce larger and slower designs than those generated by a structural description or hand-mapping. Behavioral descriptions can leave many aspects of the circuit unspecified. For example, a compiler that encounters a **while** loop must generate complicated control structures in order to allow for an unspecified number of iterations. Also, in many HLL implementations, optimizations based upon the bit width of operands cannot be performed. The compiler is generally unaware of any application-specific limitations on the operand size; it only sees the programmer's choice of data format in the program. Problems such as these might be solved through additional programmer effort to replace **while** loops whenever possible with **for** loops, and to use compiler directives to indicate exact sizes of operands [Galloway 1995; Gokhale and Stone 1998]. This method of hardware design falls between structural description and behavioral description in complexity, because although the programmers do not need to know a great deal about hardware design, they are required to follow additional guidelines that are not required for software-only implementations.

4.3. Circuit Libraries

The use of circuit or macro libraries can greatly simplify and speed the design process. By predesigning commonly used structures such as adders, multipliers, and counters, circuit creation for configurable systems becomes largely the assembly of high-level components, and only application-specific structures require detailed design. The actual architecture of the reconfigurable device can be abstracted, provided only library components are used, as these low-level details will already have been encapsulated within the library structures. Although the users of the circuit library may not know the intricacies of the destination architecture, they are still able to make use of architecture-specific optimizations, such as specialized carry

chains. This is because designers very familiar with the details of the target architecture create the components within a circuit library. They can take advantage of architecture specifics when creating the modules to make these components faster and smaller than a designer unfamiliar with the architecture likely would. An added benefit of the architecture abstraction is that the use of library components can also facilitate design migration from one architecture to another, because designers are not required to learn a new architecture, but only to indicate the new target for the library components. However, this does require that a circuit library contain implementations for more than one architecture.

One method for using library components is to simply instantiate them within an HDL design [Xilinx 1997; Altera 1999]. However, circuit libraries can also be used in general language compilers by comparing the dataflow graph of the application to the dataflow graphs of the library macros [Cadambi and Goldstein 1999]. If a dataflow representation of a macro matches a portion of the application graph, the corresponding macro is used for that part of the configuration.

Another benefit of circuit design with library macros is that of fast compilation. Because the library structures may have been premapped, preplaced, and prerouted (at least within the macro boundaries), the actual compile time is reduced to the time required to place the library components and route between them. For example, fast configuration was one of the main motivations for the creation of libraries for circuit design in the DISC reconfigurable image processing system [Hutchings 1997].

4.4. Circuit Generators

Circuit generators fulfill a role similar to circuit libraries, in that they provide optimized high-level structures for use within larger applications. Again, designers are not required to understand the low-level details of particular architectures. However, circuit generators create semicustomized high-level structures automatically at compile time, as opposed to circuit libraries that only provide static structures. For example, a circuit generator can create an adder structure of the exact bit width required by the designer, whereas a circuit library is likely to contain a limited number of adder structures, none of which may be of the correct size. Circuit generators are therefore more flexible than circuit libraries because of the customization allowed.

Some circuit generators, such as MacGen [Yasar et al. 1996], are executed at the command line using custom description files to generate physical design layout data files. Newer circuit generators, however, are functions or methods called from high-level language programs. PAM-Blox [Mencer et al. 1998], for example, is a set of circuit generators executed in C++ that generate structures for use with the PCI Pamette reconfigurable processing board. The circuit generator presented by Chu et al. [1998] contains a number of Java classes to allow a programmer to generate arbitrarily sized arithmetic and logical components for a circuit. Although the examples presented in that paper were mapped to a Xilinx 4000 series FPGA, the generator uses architecture specific libraries for module generation. The target architecture can therefore be changed through the use of a different design library. The Carry Look-Ahead circuit generator described by Stohmann and Barke [1996] is also retargetable, because it maps to an FPGA logic cell architecture defined by the user.

One drawback of the circuit generators is that they depend on a regular logic and routing structure. Hierarchical routing structures (such as those present in the Xilinx 6200 series [Xilinx 1996]) and specialized heterogeneous logic blocks are frequently not accounted for. Therefore, some optimized features of a particular architecture may be unused. For these cases, a circuit macro from a library may provide a more highly optimized structure than one created with a circuit generator,

provided that the library macro fits the needs of the application.

4.5. Partial Evaluation

Functions that are to be implemented on the reconfigurable array should occupy as little area as possible, so as to maximize the number of functions that can be mapped to the hardware. This, combined with the minimization of the delay incurred by each circuit, increases the overall acceleration of the application. Partial evaluation is the process of reducing hardware requirements for a circuit structure through optimization based upon known static inputs. Specifically, if an input is known to be constant, that value can potentially be propagated through one or more gates in the structure at compile time, and only the portions of a circuit that depend on time-varying inputs need to be mapped to the reconfigurable structure.

One example of the usefulness of this operation is that of constant coefficient multipliers. If one input to a multiplier is constant, a multiplier object can be reduced from a general-purpose multiplier to a set of additions with static-length shifts between them corresponding to the locations of 1s in the binary constant. This type of reduction leads to a lower area requirement for the circuit, and potentially higher performance due to fewer gate delays encountered on the critical path. Partial evaluation can also be performed in conjunction with circuit generation, where the constants passed to the generator function are used to simplify the created hardware circuit [Wang and Lewis 1997; Chu et al. 1998]. Other examples of this type of optimization for specific algorithms include the partial evaluation of DES encryption circuits [Leonard and Mangione-Smith 1997], and the partial evaluation of constant multipliers and fixed polynomial division circuits [Payne 1997].

4.6. Memory Allocation

As with traditional software programs, it may be necessary in reconfigurable computing to allocate memories to hold variables and other data. Off-chip memories may be added to the reconfigurable system. Alternately, if a reconfigurable system includes memory blocks embedded into the reconfigurable logic, these may be used, provided that the storage requirements do not surpass the available embedded memory. If multiple off-chip memories are available to a reconfigurable system, variables used in parallel should be placed into different memory structures, such that they can be accessed simultaneously [Gokhale and Stone 1999]. When smaller embedded memory units are used, larger memories can be created from the smaller ones. However, in this case, it is desirable to ensure that each smaller memory is close to the computation that most requires its contents [Babb et al. 1999]. As mentioned earlier, the small embedded memories that are not allocated for data storage may be used to perform logic functions.

4.7. Parallelization

One of the benefits of reconfigurable computing is the ability to execute multiple operations in parallel. In cases where circuits are specified using a structural hardware description language, the user specifies all structures and timing, and therefore either implicitly or explicitly specifies any parallel operation. However, for behavioral and HLL descriptions, there are two methods to incorporate parallelism: manual parallelization through special instructions or compiler directives, and automatic parallelization by the compiler.

To manually incorporate parallelism within an application, the programmer can specifically mark sections of code that should run as parallel threads, and use similar operations to those used in traditional parallel compilers [Cronquist et al. 1998; Gokhale and Stone 1998]. For example, a signal/wait technique can be used to perform synchronization of the different threads of the computation. The RaPiD-B language [Cronquist et al. 1998] is one that uses this methodology.

Although the NAPA C compiler [Gokhale and Stone 1998] requires programmers to mark the areas of code for executing the host processor and the reconfigurable hardware in parallel, it also detects and exploits fine-grained parallelism within computations destined for the reconfigurable hardware.

Automatic parallelization of inner loops is another common technique in reconfigurable hardware compilers to attempt to maximize the use of the reconfigurable hardware. The compiler will select the innermost loop level to be completely unrolled for parallel execution in hardware, potentially creating a heavily pipelined structure [Cronquist et al. 1998; Weinhardt and Luk 1999]. For these cases, outer loops may not have multiple iterations executing simultaneously. Any loop reordering to improve the parallelism of the circuit must be done by the programmer. However, some compiler systems have taken this procedure a step further and focus on the parallelization of all loops within the program, not just the inner loops [Wang and Lewis 1997; Budiu and Goldstein 1999]. This type of compiler generates a control flow graph based upon the entire program source code. Loop unrolling is used in order to increase the available parallelism, and the graph is then used to schedule parallel operations in the hardware.

4.8. Multi-FPGA System Software

When reconfigurable systems use more than one FPGA to form the complete reconfigurable hardware, there are additional compilation issues to deal with [Hauck and Agarwal 1996]. The design must first be partitioned into the different FPGA chips [Hauck 1995; Acock and Dimond 1997; Vahid 1997; Brasen and Saucier 1998; Khalid 1999]. This is generally done by placing each highly connected portions of a circuit into a single chip. Multi-FPGA systems have a limited number of I/O pins that connect the chips together, and therefore their use must be minimized in the overall circuit mapping. Also, by minimizing the amount of routing required between the FPGAs, the number of paths with a high (inter-chip) delay is reduced, and the circuit may have an overall higher performance. Similarly, those sections of the circuit that require a short delay time must be placed upon the same chip. Global placement then determines which of the actual FPGAs in the multi-FPGA system will contain each of the partitions.

After the circuit has been partitioned into the different FPGA chips, the connections between the chips must be routed [Mak and Wong 1997; Ejnioui and Ranganathan 1999]. A global routing algorithm determines at a high level the connections between the FPGA chips. It first selects a region of output pins on the source FPGA for a given signal, and determines which (if any) routing switches or additional FPGAs the signal must pass through to get to the destination FPGA. Detailed routing and pin assignment [Slimane-Kade et al. 1994; Hauck and Borriello 1997; Mak and Wong 1997; Ejnioui and Ranganathan 1999] are then used to assign signals to traces on an existing multi-FPGA board, or to create traces for a multi-FPGA board that is to be created specifically to implement the given circuit.

Because multi-FPGA systems use interchip connections to allow the circuit partitions to communicate, they frequently require a higher proportion of I/O resources vs. logic in each chip than is normally required in single-FPGA use. For this reason, some research has focused on methods to allow pins of the FPGAs to be reused for multiple signals. This procedure is referred to as Virtual Wires [Babb et al. 1993; Agarwal 1995; Selvidge et al. 1995], and allows for a flexible trade-off between logic and I/O within a given multi-FPGA system. Signals are multiplexed onto a single wire by using multiple virtual clock cycles, one per multiplexed signal, within a user clock cycle, thus pipelining the communication. In this manner, the I/O requirements of a circuit can be reduced, while the logic requirements (because of the added circuitry used for the multiplexing) are increased.

4.9. Design Testing

After compilation, an application needs to be tested for correct operation before deployment. For hardware configurations that have been generated from behavioral descriptions, this is similar to the debugging of a software application. However, structurally and manually created circuits must be simulated and debugged with techniques based upon those from the design of general hardware circuits. For these structures, simulation and debugging are critical not only to ensure proper circuit operation, but also to prevent possible incorrect connections from causing a short within the circuit, which can damage the reconfigurable hardware.

There are several different methods of observing the behavior of a configuration during simulation. The contents of memory structures within the design can be viewed, modified, or saved. This allows on-the-fly customization of the simulated execution environment of the reconfigurable hardware, as well as a method for examining the computation results. The input and output values of circuit structures and substructures can also be viewed either on a generated schematic drawing or with a traditional waveform output. By examining these values, the operation of the circuit can be verified for correctness, and conflicts on individual wires can be seen. A number of simulation and debugging software systems have been developed that use some or all of these techniques [Arnold et al. 1992; Buell et al. 1996; Gehring and Ludwig 1996; Lysaght and Stockwood 1996; Bellows and Hutchings 1998; Hutchings et al. 1999; McKay and Singh 1999; Vasilko and Cabanis 1999].

4.10. Software Summary

Reconfigurable hardware systems require software compilation tools to allow programmers to harness the benefits of reconfigurable computing. On one end of the spectrum, circuits for reconfigurable systems can be designed manually, leveraging all application-specific and architecture-specific optimizations available to generate a high-performance application. However, this requires a great deal of time and effort on the part of the designer. At the opposite end of the spectrum is fully automatic compilation of a high-level language. Using the automatic tools, a software programmer can transparently utilize the reconfigurable hardware without the need for direct intervention. The circuits created using this method, while quickly and easily created, are generally larger and slower than manually created versions. The actual tools available for compilation onto reconfigurable systems fall at various points within this range, where many are partially automated but require some amount of manual aid. Circuit designers for reconfigurable systems therefore face a trade-off between the ease of design and the quality of the final layout.

5. RUN-TIME RECONFIGURATION

Frequently, the areas of a program that can be accelerated through the use of reconfigurable hardware are too numerous or complex to be loaded simultaneously onto the available hardware. For these cases, it is beneficial to be able to swap different configurations in and out of the reconfigurable hardware as they are needed during program execution (Figure 13). This concept is known as run-time reconfiguration (RTR).

Run-time reconfiguration is based upon the concept of virtual hardware, which is similar to virtual memory. Here, the physical hardware is much smaller than the sum of the resources required by each of the configurations. Therefore, instead of reducing the number of configurations that are mapped, we instead swap them in and out of the actual hardware as they are needed. Because run-time reconfiguration allows more sections of an application to be mapped into hardware than can be fit in a non-run-time reconfigurable system, a greater portion of the program can be accelerated. This provides potential for an overall improvement in performance.

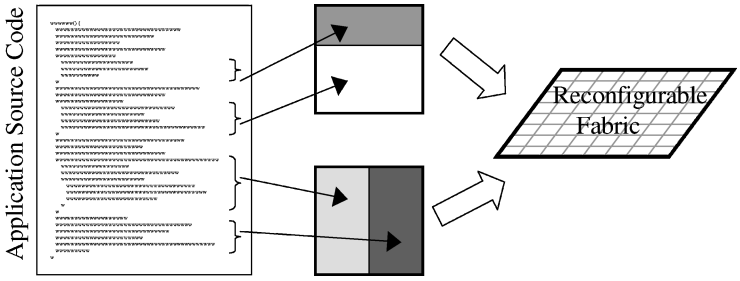

Fig. 13. Applications which are too large to entirely fit on the reconfigurable hardware can be partitioned into two or more smaller configurations that can occupy the hardware at different times.

During a single program's execution, configurations are swapped in and out of the reconfigurable hardware. Some of these configurations will likely require access to the results of other configurations. Configurations that are active at different periods in time therefore must be provided with a method to communicate with one another. Primarily, this can be done through the use of registers [Ebeling et al. 1996; Cadambi et al. 1998; Rupp et al. 1998; Scalera and Vazquez 1998], the contents of which can remain intact between reconfigurations. This allows one configuration to store a value, and a later configuration to read back that value for use in further computations. An alternative for reconfigurable systems that do not include state-holding devices is to write the result back to registers or memory external to the reconfigurable array, which is then read back by successive configurations [Hauck et al. 1997].

There are a few different configuration memory styles that can be used with reconfigurable systems. A single context device is a serially programmed chip that requires a complete reconfiguration in order to change any of the programming bits. A multicontext device has multiple layers of programming bits, each of which can be active at a different point in time. Devices that can be selectively programmed without a complete reconfiguration are called partially reconfigurable. These different types of configuration memory are described in more detail later. An advantage of the multicontext FPGA over a single context architecture is that it allows for an extremely fast context switch (on the order of nanoseconds), whereas the single context may take milliseconds or more to reprogram. The partially reconfigurable architecture is also more suited to run-time reconfiguration than the single context, because small areas of the array can be modified without requiring that the entire logic array be reprogrammed.

For all of these run-time reconfigurable architectures, there are also a number of compilation issues that are not encountered in systems that only configure at the beginning of an application. For example, run-time reconfigurable systems are able to optimize based on values that are known only at run-time. Furthermore, compilers must consider the run-time reconfigurability when generating the different circuit mappings, not only to be aware of the increase in time-multiplexed capacity, but also to schedule reconfigurations so as to minimize the overhead that they incur. These software issues, as well as an overview of methods to perform fast configuration, will be explored in the sections that follow.

5.1. Reconfigurable Models

Traditional FPGA structures have been single context, only allowing one full-chip configuration to be loaded at a time. However, designers of reconfigurable systems have found this style of configuration to be too limiting or slow to efficiently implement run-time reconfiguration. The

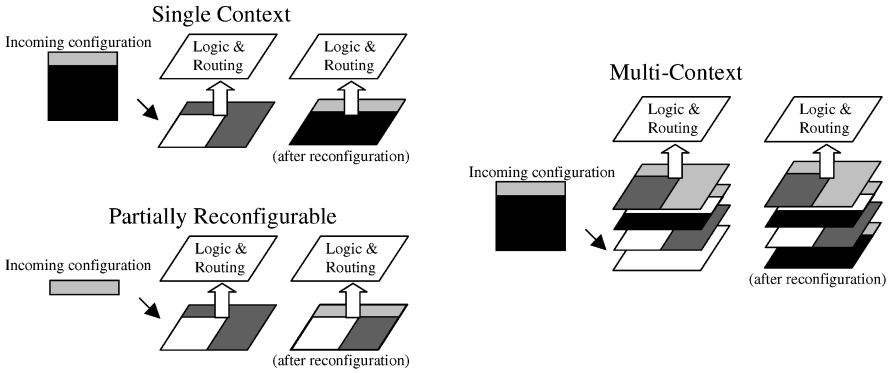

Fig. 14. The different basic models of reconfigurable computing: single context, multicontext, and partially reconfigurable. Each of these designs is shown performing a reconfiguration.

following discussion defines the single context device, and further considers newer FPGA designs (multicontext and partially reconfigurable), along with their impact on run-time reconfiguration.

5.1.1. Single Context. Current single context FPGAs are programmed using a serial stream of configuration information. Because only sequential access is supported, any change to a configuration on this type of FPGA requires a complete reprogramming of the entire chip. Although this does simplify the reconfiguration hardware, it does incur a high overhead when only a small part of the configuration memory needs to be changed. Many commercial FPGAs are of this style, including the Xilinx 4000 series [Xilinx 1994], the Altera Flex10K series [Altera 1998], and Lucent's Orca series [Lucent 1998]. This type of FPGA is therefore more suited for applications that can benefit from reconfigurable computing without run-time reconfiguration. A single context FPGA is depicted in Figure 14.

In order to implement run-time reconfiguration onto a single context FPGA, the configurations must be grouped into contexts, and each full context is swapped in and out of the FPGA as needed. Because each of these swap operations involve reconfiguring the entire FPGA, a good partitioning of the configurations between contexts is essential in order to minimize the total reconfiguration delay. If all the configurations used within a certain time period are present in the same context, no reconfiguration will be necessary. However, if a number of successive configurations are each partitioned into different contexts, several reconfigurations will be needed, slowing the operation of the run-time reconfigurable system.

5.1.2. Multicontext. A multicontext FPGA includes multiple memory bits for each programming bit location [DeHon 1996; Trimberger et al. 1997a; Scalera and Vazquez 1998; Chameleon 2000]. These memory bits can be thought of as multiple planes of configuration information, as shown in Figure 14. One plane of configuration information can be active at a given moment, but the device can quickly switch between different planes, or contexts, of already-programmed configurations. In this manner, the multicontext device can be considered a multiplexed set of single context devices, which requires that a context be fully reprogrammed to perform any modification. This system does allow for the background loading of a context, where one plane is active and in execution while an inactive place is in the process of being programmed. Figure 15 shows a multicontext memory bit, as used in [Trimberger et al. 1997a]. A commercial product that uses this technique is the CS2000 RCP series from Chameleon, Inc [Chameleon 2000]. This device provides

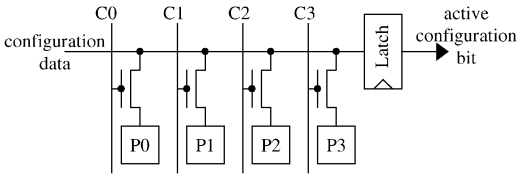

Fig. 15. A four-bit multicontexted programming bit [Trimberger et al. 1997a]. P0-P3 are the stored programming bits, while C0-C3 are the chip-wide control lines that select the context to program or activate.

two separate planes of programming information. At any given time, one of these planes is controlling current execution on the reconfigurable fabric, and the other plane is available for background loading of the next needed configuration.

Fast switching between contexts makes the grouping of the configurations into contexts slightly less critical, because if a configuration is on a different context than the one that is currently active, it can be activated within an order of nanoseconds, as opposed to milliseconds or longer. However, it is likely that the number of contexts within a given program is larger than the number of contexts available in the hardware. In this case, the partitioning again becomes important to ensure that configurations occurring in close temporal proximity are in a set of contexts that are loaded into the multicontext device at the same time. More aspects involving temporal partitioning for single- and multicontext devices will be discussed in the section on compilers for run-time reconfigurable systems.

5.1.3. Partially Reconfigurable. In some cases, configurations do not occupy the full reconfigurable hardware, or only a part of a configuration requires modification. In both of these situations, a partial reconfiguration of the array is required, rather than the full reconfiguration required by a single- or multicontext device. In a partially reconfigurable FPGA, the underlying programming bit layer operates like a RAM device. Using addresses to specify the target location of the configuration data allows for selective reconfiguration of the array. Frequently, the undisturbed portions of the array may continue execution, allowing the overlap of computation with reconfiguration. This has the benefit of potentially hiding some of the reconfiguration latency.

When configurations do not require the entire area available within the array, a number of different configurations may be loaded into unused areas of the hardware at different times. Since only part of the array is reconfigured at a given point in time, the entire array does not require reprogramming. Additionally, some applications require the updating of only a portion of a mapped circuit, while the rest should remain intact, as shown in Figure 14. For example, in a filtering operation in signal processing, a set of constant values that change slowly over time may be reinitialized to a new value, yet the overall computation in the circuit remains static. Using this selective reconfiguration can greatly reduce the amount of configuration data that must be transferred to the FPGA. Several run-time reconfigurable systems are based upon a partially reconfigurable design, including Chimaera [Hauck et al. 1997], PipeRench [Cadambi et al. 1998; Goldstein et al. 2000], NAPA [Rupp et al. 1998], and the Xilinx 6200 and Virtex FPGAs [Xilinx 1996, 1999].

Unfortunately, since address information must be supplied with configuration data, the total amount of information transferred to the reconfigurable hardware may be greater than what is required with a single context design. This makes a full reconfiguration of the entire array slower than the single context version. However, a partially reconfigurable design is intended for applications in which the size of the configurations is small enough that more than one can fit on the available hardware simultaneously. Plus, as we discuss in subsequent sections, a number of fast configuration methods have been explored for partially reconfigurable systems in order to help reduce the configuration data traffic requirements.

5.1.4. Pipeline Reconfigurable. A modification of the partially reconfigurable FPGA design is one in which the partial

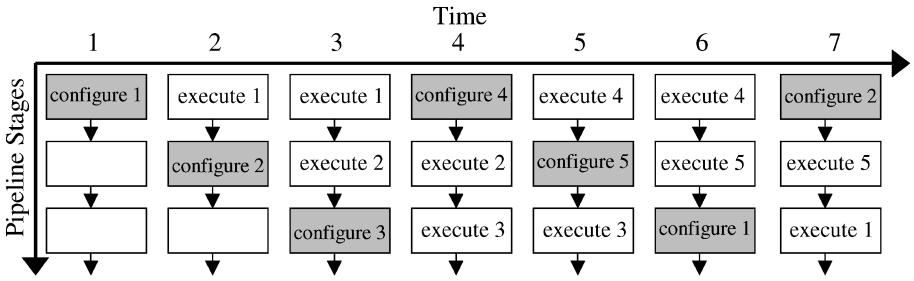

Fig. 16. A timeline of the configuration and reconfiguration of pipeline stages on a pipeline reconfigurable FPGA. This example shows three physical pipeline stages implementing five virtual pipeline stages [Cadambi et al. 1998].

reconfiguration occurs in increments of pipeline stages. This style of reconfigurable hardware is called pipeline reconfigurable, or sometimes a striped FPGA [Luk et al. 1997b; Cadambi et al. 1998; Deshpande and Somani 1999; Goldstein et al. 2000]. Each stage is configured as a whole. This is primarily used in datapath-style computations, where more pipeline stages are used than can fit simultaneously on available hardware. Figure 16 shows an example of a pipeline reconfigurable array implementing more pipeline stages than can fit on the available hardware. In a pipeline-reconfigurable FPGA, there are two primary execution possibilities. Either the number of hardware pipeline stages available is greater than or equal to the number of pipeline stages of the designed circuit (virtual pipeline stages), or the number of virtual pipeline stages will exceed the number of hardware pipeline stages. The first case is straightforward: the circuit is simply mapped to the array, and some hardware stages may go unused. The second case is more complex and is the one that requires run-time reconfiguration. The pipeline stages are configured one by one, from the start of the pipeline, through the end of the available hardware stages (steps 1, 2, and 3 in Figure 16). After each stage is programmed, it begins computation. In this manner, the configuration of a stage is exactly one step ahead of the flow of data. Once the hardware pipeline has been completely filled, reuse of the hardware pipeline stages begins. Configuration of the next virtual stage begins at the first pipeline location in the hardware (step 4), overwriting the first virtual pipeline stage. The reconfiguration of the hardware pipeline stages continues until the last virtual pipeline stage has been programmed (step 7), at which point the first stage of the virtual pipeline is again configured onto the hardware for the next data set. These structures also allow for the overlap of configuration and execution, as one pipeline stage is configured while the others are executing. Therefore, N-1 data values are processed each time the virtual pipeline is fully traversed on an N-stage hardware system.

5.2. Run-Time Partial Evaluation

One of the advantages that a run-time reconfigurable device has over a system that is only programmed at the beginning of an application is the ability to perform hardware optimizations based upon values determined at run-time. Partial evaluation was already discussed in this article in reference to compilation optimizations for general reconfigurable systems. Run-time partial evaluation allows for the further exploitation of "constants" because the configurations can be modified based not only on completely static values, but also those that change slowly over time [Burns et al. 1997; Luk et al. 1997a; Payne 1997; Wirthlin and Hutchings 1997; Chu et al. 1998; McKay and Singh 1999]. This gives reconfigurable circuits the potential to achieve an even higher performance than an ASIC, which must retain generality in these situations. The circuit in the

reconfigurable system can be customized to the application at a given time, rather than to the application as a category. For example, where an ASIC may have to include a generic multiplier, a reconfigurable system could instantiate a constant coefficient multiplier that changes over time. Additionally, partial evaluation can be used in encryption systems [Leonard and Mangione-Smith 1997]. A key-specific reconfigurable encrypter or decrypter is optimized for the particular key being used, but retains the ability to use more than one key over the lifetime of the hardware (unlike a key-specialized ASIC) or during actual run-time.

Although partial evaluation can be used to reduce the overall area requirements of a circuit by removing potentially extraneous hardware within the implementation, occasionally it is preferable to reserve sufficient area for the largest case, and have all mappings occupy that area. This allows the partially evaluated portion of a given configuration to be reconfigured, while leaving the remainder of the circuit intact. For example, if a constant coefficient multiplier within a larger configuration requires that the constant be changed, only the area occupied by the multiplier requires reconfiguration. This is true even if the new constant coefficient multiplier is a larger structure than the previous one, because the reserved area for it is based upon the largest possibility [McKay and Singh 1999]. Although partial evaluation does not minimize the area occupied by the circuit in this case, the speed of configuration is improved by making the multiplier a modular replaceable component. Additionally, this method retains the speed benefits of partial reconfiguration because it still minimizes the logic and routing actually used to implement the structure.

5.3. Compilation and Configuration Scheduling

For some reconfigurable systems, a configuration requires programming the reconfigurable hardware only at the start of its execution. On the other hand, in a run-time reconfigurable system, the circuits loaded on the hardware change over time. If the user must specify by hand the loading and execution of the circuits in the reconfigurable hardware, then the compilers must include methods to indicate these operations. JHDL [Bellows and Hutchings 1998; Hutchings et al. 1999] is one such compiler. It provides for the instantiation of configurations through the use of Java constructors, and the removal of the circuits from the hardware by using a destructor on the circuit objects. This allows the programmer to indicate exactly the loading pattern of the configurations.

Alternately, the compiler can automate the use of the run-time reconfigurable hardware. For a single context or multicontext device, configurations must be temporally partitioned into a number of different full contexts of configuration information. This involves determining which configurations are likely to be used near in time to one another, and which configurations are able to fit together onto the reconfigurable hardware. Ideally, the number of reconfigurations that are to be performed is minimized. By reducing the number of reconfigurations, the proportion of time spent in reconfiguration (compared to the time spent in useful computation) is reduced.

The problem of forming and scheduling single- and multiconfiguration contexts for use in single context or multicontext FPGA designs has been discussed by a number of groups [Chang and Marek-Sadowska 1998; Trimberger 1998; Liu and Wong 1999; Purna and Bhatia 1999; Li et al. 2000a]. In particular, a single circuit that is too large to fit within the reconfigurable hardware may be partitioned over time to form a sequential set of configurations. This involves examining the control flow graph of the circuit and dividing the circuit into distinct computation nodes. The nodes can then be grouped together within contexts, based upon their proximity to one another within the flow control graph. If possible, those configurations that are used in quick succession will be placed within the same group. These groups are finally mapped into full

contexts, to be loaded into the reconfigurable hardware at run-time. Nimble [Li et al. 2000a] is one of the compilers that perform this type of operation. This compiler focuses on mapping core loops within C code to reconfigurable hardware. Hardware models for the candidate loops that will fit within the reconfigurable hardware are first extracted from the C application. Then these loops are grouped into individual configurations using a partitioning method in order to encourage the hardware loops that are used in close temporal proximity to be mapped to the same configuration, reducing configuration overhead.

For partially reconfigurable designs, the compiler must determine a good placement in order to prevent configurations that are used together in close temporal proximity from occupying the same resources. Again, through minimizing the number of reconfigurations, the overall performance of the system is increased, as configuration is a slow process [Li et al. 2000b]. An alternative approach, which allows the final placement of a configuration to be determined at run-time, is also discussed within the Fast Configuration section of this article.

5.4. Fast Configuration

Because run-time reconfigurable systems involve reconfiguration during program execution, the reconfiguration must be done as efficiently and as quickly as possible. This is in order to ensure that the overhead of the reconfiguration does not eclipse the benefit gained by hardware acceleration. Stalling execution of either the host processor or the reconfigurable hardware because of configuration is clearly undesirable. In the DISC II system, from 25% [Wirthlin and Hutchings 1996] to 71% [Wirthlin and Hutchings 1995] of execution time is spent in reconfiguration, while in the UCLA ATR work this figure can rise to over 98.5% [Mangione-Smith 1999]. If the delays caused by reconfiguration are reduced, performance can be greatly increased. Therefore, fast configuration is an important area of research for run-time reconfigurable systems.

There are a number of different tactics for reducing the configuration overhead. First, loading of the configurations can be timed such that the configuration overlaps as much as possible with the execution of instructions by the host processor. Second, compression techniques can be introduced to decrease the amount of configuration data that must be transferred to the system. Third, specialized hardware can be used to adjust the physical location of configurations at run-time based on where the free area on the hardware is located at any given time. Finally, the actual process of transferring the data from the host processor to the reconfigurable hardware can be modified to include a configuration cache, which would provide a faster reconfiguration.

5.4.1. Configuration Prefetching. Performance is improved when the actual configuration of the hardware is overlapped with computations performed by the host processor, because programming the reconfigurable hardware requires from milliseconds to seconds to accomplish. Overlapping configuration and execution prevents the host processor from stalling while it is waiting for the configuration to finish, and hides the configuration time from the program execution. Configuration prefetching [Hauck 1998a] attempts to leverage this overlap by determining when to initiate reconfiguration of the hardware in order to maximize overlap with useful computation on the host processor. It also seeks to minimize the chance that a configuration will be prefetched falsely, overwriting the configuration that is actually used next.

5.4.2. Configuration Compression. Unfortunately, there will always be cases in which the configuration overheads cannot be successfully hidden using a prefetching technique. This can occur when a conditional branch occurs immediately before the use of a configuration, potentially making a 100% correct prefetch prediction impossible, or when multiple configurations or contexts must be loaded in quick succession. In these cases, the delay incurred is minimized when the amount

of data transferred from the host processor to the reconfigurable array is minimized. Configuration compression can be used to compact this configuration information [Hauck et al. 1998b; Hauck and Wilson 1999; Li and Hauck 1999; Dandalis and Prasanna 2001].

One form of configuration compression has already been implemented in a commercial system. The Xilinx 6200 series of FPGA [Xilinx 1996] contains wildcarding hardware, which provides a method to program multiple logic cells with a single address and data value. This is accomplished by setting a special register to indicate which of the address bits should behave as "don't-care" values, resolving to multiple addresses for configuration. For example, suppose two configuration addresses, 00010 and 00110, are both to be programmed with the same value. By setting the wildcard register to 00100, the address value sent is interpreted as 00X10 and both these locations are programmed using either of the two addresses above in a single operation. Hauck et al. [1998b] discuss the benefits of this hardware, while Li and Hauck [1999] cover a potential extension to the concept, where "don't care" values in the configuration stream can be used to allow areas with similar but not identical configuration data values to also be programmed simultaneously.

Within partially reconfigurable systems, there is an added potential to compress effectively the amount of data sent to the reconfigurable hardware. A configuration can possibly reuse configuration information already present on the array, such that only the areas differing in configuration values must be reprogrammed. Therefore, configuration time can be reduced through the identification of these common components and the calculation of the incremental configurations that must be loaded [Luk et al. 1997a; Shirazi et al. 1998].

Alternately, similar operations can be grouped together to form a single configuration that contains extra control circuitry in order to implement the various functions within the group [Kastrup et al. 1999]. By creating larger configurations out of groups of smaller configurations, the configuration overhead of partial reconfiguration is reduced because more operations can be present on chip simultaneously. However, there are some area and execution penalties imposed by this method, creating a trade-off between reduced reconfiguration overhead and faster execution with a smaller area.

5.4.3. Relocation and Defragmentation in Partially Reconfigurable Systems. Partially reconfigurable systems have the advantage over single context systems in that they allow a new configuration to be written to the programmable logic while the configurations not occupying that same area remain intact and available for future use. Because these configurations will not have to be reconfigured onto the array, and because the programming of a single configuration can require the transfer of far less configuration data than the programming of an entire context, a partially reconfigurable system can incur less configuration overhead than a single context FPGA.

However, inefficiencies can arise if two partial configurations are supposed to be located at overlapping physical locations on the FPGA. If these configurations are repeatedly used one after another, they must be swapped in and out of the array each time. This type of conflict could negate much of the benefit achieved by partially reconfigurable systems. A better solution to this problem is to allow the final placement of the configurations to occur at run-time, allowing for run-time relocation of those configurations [Li et al. 2000b; Compton et al. 2002]. Using relocation, a new configuration may be placed onto the reconfigurable array where it will cause minimum conflict with other needed configurations already present on the hardware. A number of different systems support run-time relocation, including Chimaera [Hauck et al. 1997], Garp [Hauser and Wawrzynek 1997], and PipeRench [Cadambi et al. 1998; Goldstein et al. 2000].

Even with relocation, partially reconfigurable hardware can still suffer from some

placement conflicts that could be avoided by using an additional hardware optimization. Over time, as a partially reconfigurable device loads and unloads configurations, the location of the unoccupied area on the array is likely to become fragmented, similar to what occurs in memory systems when RAM is allocated and deallocated. There may be enough empty area on the device to hold an incoming configuration, but it may be distributed throughout the array. A configuration normally requires a contiguous region of the chip, so it would have to overwrite a portion of a valid configuration in order to be placed onto the reconfigurable hardware. A system that incorporates the ability to perform defragmentation of the reconfigurable array, however, would be able to consolidate the unused area by moving valid configurations to new locations [Diessel and El Gindy 1997; Compton et al. 2002]. This area can then be used by incoming configurations, potentially without overwriting any of the moved configurations.

5.4.4. Configuration Caching. Because a great deal of the delay caused by configuration is due to the distance between the host processor and the reconfigurable hardware, as well the reading of the configuration data from a file or main memory, a configuration cache can potentially reduce the costs of reconfiguration [Deshpande et al. 1999; Li et al. 2000b]. By storing the configurations in fast memory near to the reconfigurable array, the data transfer during reconfiguration is accelerated, and the overall time required is reduced. Additionally, a special configuration cache can allow for specialized direct output to the reconfigurable hardware [Compton et al. 2000]. This output can leverage the close proximity of the cache by providing high-bandwidth communications that would facilitate wide parallel loading of the configuration data, further reducing configuration times.

5.5. Potential Problems with RTR

Partial reconfiguration involves selectively programming portions of the reconfigurable array. However, in many architectures, there are some routing resources that traverse long distances, and may traverse areas allocated to different configurations. Care must be taken such that different configurations do not attempt to drive to these wires simultaneously, as multiple drivers to a wire can potentially damage the hardware. Therefore, systems such as the Xilinx 6200 [Xilinx 1996] and Chimaera [Hauck et al. 1997] have specially designed routing resources that prevent multiple drivers. LEGO [Chow et al. 1999b] includes an additional control signal preventing conflicts during the span of time between startup and actual programming of the hardware.

An additional difficulty in using runtime reconfigurable systems occurs when the host processor runs multiple threads or processes. These threads or processes may each have their own sets of configurations that are to be mapped to the reconfigurable hardware. Issues such as the correct use of memory protection and virtual memory must be considered during memory accesses by the reconfigurable hardware [Chien and Byun 1999; Jacob and Chow 1999; Jean et al. 1999]. Another problem can occur when one thread or process configures the hardware, which is then reconfigured by a different thread or process. Threads and processes must be prevented from incorrectly calling hardware functions that no longer appear on the reconfigurable hardware. This requires that the state of the reconfigurable hardware be set to "dirty" on a main processor context switch, or re-loaded with the correct configuration context.

Partially reconfigurable systems must also protect against inter-process or inter-thread conflicts within the array. Even if each application has ensured that their own configurations can safely coexist, a combination of configurations from different applications re-introduces the possibility of inadvertently causing an electrical short within the reconfigurable hardware. This particular issue can be solved through the use of an architecture that does not have "bad" configurations, such as the 6200 series [Xilinx 1996] and

Chimaera [Hauck et al. 1997]. The potential for this type of conflict also introduces the possibility of extremely destructive configurations that can destroy the system's underlying hardware.

5.6. Run-Time Reconfiguration Summary

We have discussed the benefits of using run-time reconfiguration to increase the benefits gained through reconfigurable computing. Different configurations may be used at different phases of a program's execution, customizing the hardware not only for the application, but also for the different stages of the application. Run-time reconfiguration also allows configurations larger than the available reconfigurable hardware to be implemented, as these circuits can be split into several smaller ones that are used in succession. Because of the delays associated with configuration, this style of computing requires that reconfiguration be performed in a very efficient manner. Multicontext and partially reconfigurable FPGAs are both designed to improve the time required for reconfiguration. Hardware optimizations, such as wildcarding, run-time relocation, and defragmentation, further decrease configuration overhead in a partially reconfigurable design. Software techniques to enable fast configuration, including prefetching and incremental configuration calculation, were also discussed.

6. CONCLUSION

Reconfigurable computing is becoming an important part of research in computer architectures and software systems. By placing the computationally intense portions of an application onto the reconfigurable hardware, that application can be greatly accelerated. This is because reconfigurable computing combines many of the benefits of both software and ASIC implementations. Like software, the mapped circuit is flexible, and can be changed over the lifetime of the system or even the lifetime of the application. Similar to an ASIC, reconfigurable systems provide a method to map circuits into hardware. Reconfigurable systems therefore have the potential to achieve far greater performance than software as a result of bypassing the fetch-decode-execute cycle of traditional microprocessors as well as possibly exploiting a greater degree of parallelism.

Reconfigurable hardware systems come in many forms, from a configurable functional unit integrated directly into a CPU, to a reconfigurable coprocessor coupled with a host microprocessor, to a multi-FPGA stand-alone unit. The level of coupling, granularity of computation structures, and form of routing resources are all key points in the design of reconfigurable systems. The use of heterogeneous structures can also greatly add to the overall performance of the final design.

Compilation tools for reconfigurable systems range from simple tools that aid in the manual design and placement of circuits, to fully automatic design suites that use program code written in a high-level language to generate circuits and the controlling software. The variety of tools available allows designers to choose between manual and automatic circuit creation for any or all of the design steps. Although automatic tools greatly simplify the design process, manual creation is still important for performance-driven applications. Circuit libraries and circuit generators are additional software tools that enable designers to quickly create efficient designs. These tools attempt to aid the designer in gaining the benefits of manual design without entirely sacrificing the ease of automatic circuit creation.

Finally, run-time reconfiguration provides a method to accelerate a greater portion of a given application by allowing the configuration of the hardware to change over time. Apart from the benefits of added capacity through the use of virtual hardware, run-time reconfiguration also allows for circuits to be optimized based on run-time conditions. In this manner, performance of a reconfigurable system can approach or even surpass that of an ASIC.

Reconfigurable computing systems have shown the ability to accelerate program

execution greatly, providing a high-performance alternative to software-only implementations. However, no one hardware design has emerged as the clear pinnacle of reconfigurable design. Although general-purpose FPGA structures have standardized into LUT-based architectures, groups designing hardware for reconfigurable computing are currently also exploring the use of heterogeneous structures and word-width computational elements. Those designing compiler systems face the task of improving automatic design tools to the point where they may achieve mappings comparable to manual design for even high-performance applications. Within both of these research categories lies the additional topic of run-time reconfiguration. While some work has been done in this field as well, research must continue in order to be able to perform faster and more efficient reconfiguration. Further study into each of these topics is necessary in order to harness the full potential of reconfigurable computing.

REFERENCES

ABOUZEID, P., BABBA, P., DE PAULET, M. C., AND SAUCIER, G. 1993. Input-driven partitioning methods and application to synthesis on table-lookup-based FPGA's. *IEEE Trans. Comput. Aid. Des. Integ. Circ. Syst. 12*, 7, 913–925.

ACOCK, S. J. B. AND DIMOND, K. R. 1997. Automatic mapping of algorithms onto multiple FPGA-SRAM Modules. *Field-Programmable Logic and Applications*, W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Lecture Notes in Computer Science, vol. 1304, Springer-Verlag, Berlin, Germany, 255–264.

ADAPTIVE SILICON, INC. 2001. *MSA 2500 Programmable Logic Cores*. Adaptive Silicon, Inc., Los Gatos, CA.

AGARWAL, A. 1995. *VirtualWires: A Technology for Massive Multi-FPGA Systems.* Available online at http://www.ikos.com/products/virtualwires.ps.

AGGARWAL, A. AND LEWIS, D. 1994. Routing architectures for hierarchical field programmable gate arrays. In *Proceedings of the IEEE International Conference on Computer Design*, 475–478.

ALEXANDER, M. J. AND ROBINS, G. 1996. New performance-driven FPGA routing algorithms. *IEEE Trans. CAD Integ. Circ. Syst. 15*, 12, 1505–1517.

ALTERA CORPORATION. 1998. *Data Book*. Altera Corporation, San Jose, CA.

ALTERA CORPORATION. 1999. *Altera MegaCore Functions*. Available online at http://www.altera.com/html/tools/megacore.html. Altera Corporation, San Jose, CA.

ALTERA CORPORATION. 2001. *Press Release: Altera Unveils First Complete System-on-a-Programmable-Chip Solution at Embedded Systems Conference*. Altera Corporation, San Jose, CA.

ANNAPOLIS MICROSYSTEMS, INC. 1998. *Wildfire Reference Manual*. Annapolis Microsystems, Inc, Annapolis, MD.

ARNOLD, J. M., BUELL, D. A., AND DAVIS, E. G. 1992. Splash 2. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 316–324.

BABB, J., RINARD, M., MORITZ, C. A., LEE, W., FRANK, M., BARUA, R., AND AMARASINGHE, S. 1999. Parallelizing applications into silicon. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 70–80.

BABB, J., TESSIER, R., AND AGARWAL, A. 1993. Virtual wires: Overcoming pin limitations in FPGA-based logic emulators. In *IEEE Workshop on FPGAs for Custom Computing Machines*, 142–151.

BELLOWS, P. AND HUTCHINGS, B. 1998. JHDL—An HDL for reconfigurable systems. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 175–184.

BETZ, V. AND ROSE, J. 1997. VPR: A new packing, placement and routing tool for FPGA research. Lecture Notes in Computer Science 1304—*Field-Programmable Logic and Applications*. W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 213–222.

BETZ, V. AND ROSE, J. 1999. FPGA routing architecture: Segmentation and buffering to optimize speed and density. *ACM/SIGDA International Symposium on FPGAs*, 59–68.

BRASEN, D. R., AND SAUCIER, G. 1998. Using cone structures for circuit partitioning into FPGA packages. *IEEE Trans. CAD Integ. Circ. Syst. 17*, 7, 592–600.

BROWN, S. D., FRANCIS, R. J., ROSE, J., AND VRANESIC, Z. G. 1992a. *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, Boston, MA.

BROWN, S., ROSE, J., AND VRANESIC, Z. G. 1992b. A detailed router for field-programmable gate arrays. *IEEE Trans. Comput. Aid. Desi. 11*, 5, 620–628.

BUDIU, M. AND GOLDSTEIN, S. C. 1999. Fast compilation for pipelined reconfigurable fabrics. *ACM/SIGDA International Symposium on FPGAs*, 195–205.

BUELL, D., ARNOLD, S. M., AND KLEINFELDER, W. J. 1996. *SPLASH 2*: *FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, Los Alamitos, CA.

BURNS, J., DONLIN, A., HOGG, J., SINGH, S., AND DE WIT, M. 1997. A dynamic reconfiguration run-time system. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 66–75.

BUTTS, M. AND BATCHELLER, J. 1991. Method of using electronically reconfigurable logic circuits. *US Patent 5,036,473*.

CADAMBI, S. AND GOLDSTEIN, S. C. 1999. CPR: A configuration profiling tool. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 104–113.

CADAMBI, S., WEENER, J., GOLDSTEIN, S. C., SCHMIT, H., AND THOMAS, D. E. 1998. Managing pipeline-reconfigurable FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 55–64.

CALLAHAN, T. J., CHONG, P., DEHON, A., AND WAWRZYNEK, J. 1998. Fast Module Mapping and Placement for Datapaths in FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 123–132.

CALLAHAN, T. J., HAUSER, J. R., AND WAWRZYNEK, J. 2000. The Garp architecture and C compiler. *IEEE Comput. 3*, 4, 62–69.

CARDOSO, J. M. P. AND NETO, H. C. 1999. Macro-based hardware compilation of Java™ bytecodes into a dynamic reconfigurable computing system. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2–11.

CHAMELEON SYSTEMS, INC. 2000. *CS2000 Advance Product Specification*. Chameleon Systems, Inc., San Jose, CA.

CHAN, P. K. AND SCHLAG, M. D. F. 1997. Acceleration of an FPGA router. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 175–181.

CHANG, D. AND MAREK-SADOWSKA, M. 1998. Partitioning sequential circuits on dynamically reconfigurable FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 161–167.

CHANG, S. C., MAREK-SADOWSKA, M., AND HWANG, T. T. 1996. Technology mapping for TLU FPGA's based on decomposition of binary decision diagrams. *IEEE Trans. CAD Integ. Circ. Syst. 15*, 10, 1226–1248.

CHICHKOV, A. V. AND ALMEIDA, C. B. 1997. An hardware/software partitioning algorithm for custom computing machines. Lecture Notes in Computer Science 1304—*Field-Programmable Logic and Applications*. W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 274–283.

CHIEN, A. A. AND BYUN, J. H. 1999. Safe and protected execution for the morph/AMRM reconfigurable processor. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 209–221.

CHOW, P., SEO, S. O., ROSE, J., CHUNG, K., PÁEZ-MONZÓN, G., AND RAHARDJA, I. 1999a. The design of an SRAM-based field-programmable Gate Array—Part I: Architecture. *IEEE Trans. VLSI Syst. 7*, 2, 191–197.

CHOW, P., SEO, S. O., ROSE, J., CHUNG, K., PÁEZ-MONZÓN, G., AND RAHARDJA, I. 1999b. The design of an SRAM-based field-programmable Gate Array—Part II: Circuit Design and Layout. *IEEE Trans. VLSI Syst. 7*, 3, 321–330.

CHOWDHARY, A. AND HAYES, J. P. 1997. General modeling and technology-mapping technique for LUT-based FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 43–49.

CHU, M., WEAVER, N., SULIMMA, K., DEHON, A., AND WAWRZYNEK, J. 1998. Object oriented circuit-generators in Java. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 158–166.

COMPTON, K., COOLEY, J., KNOL, S., AND HAUCK, S. 2000. Configuration relocation and defragmentation for FPGAs, Northwestern University Technical Report, Available online at http://www.ece.nwu.edu/~kati/publications.html.

COMPTON, K., LI, Z., COOLEY, J., KNOL, S., AND HAUCK, S. 2002. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Trans. VLSI Syst.*, to appear.

CONG, J. AND HWANG, Y. Y. 1998. Boolean matching for complex PLBs in LUT-based FPGAs with application to architecture evaluation. *ACM/SIGDA International Symposium on FPGAs*, 27–34.

CONG, J. AND WU, C. 1998. An efficient algorithm for performance-optimal FPGA technology mapping with retiming. *IEEE Trans. CAD Integr. Circ. Syst. 17*, 9, 738–748.

CONG, J., WU, C., AND DING, Y. 1999. Cut ranking and pruning enabling a general and efficient FPGA mapping solution. *ACM/SIGDA International Symposium on FPGAs*, 29–35.

CONG, J. AND XU, S. 1998. Technology mapping for FPGAs with embedded memory blocks. *ACM/SIGDA International Symposium on FPGAs*, 179–188.

CRONQUIST, D. C., FRANKLIN, P., BERG, S. G., AND EBELING, C. 1998. Specifying and compiling applications for RaPiD. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 116–125.

DANDALIS, A. AND PRASANNA, V. K. 2001. Configuration compression for FPGA-based embedded systems. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 173–182.

DEHON, A. 1996. DPGA Utilization and Application. *ACM/SIGDA International Symposium on FPGAs*, 115–121.

DEHON, A. 1999. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). *ACM/SIGDA International Symposium on FPGAs*, 69–78.

DESHPANDE, D., SOMANI, A. K., AND TYAGI, A. 1999. Configuration caching vs data caching for striped FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 206–214.

DIESSEL, O. AND EL GINDY, H. 1997. Run-time compaction of FPGA designs. Lecture Notes in Computer Science 1304—*Field-Programmable Logic and Applications*. W. Luk, P. Y. K. Cheung, M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 131–140.

DOLLAS, A., SOTIRIADES, E., AND EMMANOUELIDES, A. 1998. Architecture and design of GE1, A FCCM for golomb ruler derivation. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 48–56.

EBELING, C., CRONQUIST, D. C., AND FRANKLIN, P. 1996. RaPiD—Reconfigurable pipelined datapath. Lecture Notes in Computer Science 1142—*Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*. R. W. Hartenstein, M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 126–135.

EJNIOUI, A. AND RANGANATHAN, N. 1999. Multi-terminal net routing for partial crossbar-based multi-FPGA systems. *ACM/SIGDA International Symposium on FPGAs*, 176–184.

ELBIRT, A. J. AND PAAR, C. 2000. An FPGA implementation and performance evaluation of the serpent block cipher. *ACM/SIGDA International Symposium on FPGAs*, 33–40.

EMMERT, J. M. AND BHATIA, D. 1999. A methodology for fast FPGA floorplanning. *ACM/SIGDA International Symposium on FPGAs*, 47–56.

ESTRIN, G., BUSSEL, B., TURN, R., AND BIBB, J. 1963. Parallel processing in a restructurable computer system. *IEEE Trans. Elect. Comput.* 747–755.

GALLOWAY, D. 1995. The transmogrifier C hardware description language and compiler for FPGAs. *IEEE Symposium on FPGAs for Custom Computing Machines*, 136–144.

GEHRING, S. AND LUDWIG, S. 1996. The trianus system and its application to custom computing. Lecture Notes in Computer Science 1142—*Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*. R. W. Hartenstein and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 176–184.

GEHRING, S. W. AND LUDWIG, S. H. M. 1998. Fast integrated tools for circuit design with FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 133–139.

GOKHALE, M. B. AND STONE, J. M. 1998. NAPA C: Compiling for a hybrid RISC/FPGA architecture. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 126–135.

GOKHALE, M. B. AND STONE, J. M. 1999. Automatic allocation of arrays to memories in FPGA processors with multiple memory banks. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 63–69.

GOLDSTEIN, S. C., SCHMIT, H., BUDIU, M., CADAMBI, S., MOE, M., AND TAYLOR, R. 2000. PipeRench: A Reconfigurable Architecture and Compiler, *IEEE Computer*, vol. 33, No. 4.

GRAHAM, P. AND NELSON, B. 1996. Genetic algorithms in software and in hardware—A performance analysis of workstations and custom computing machine implementations. *IEEE Symposium on FPGAs for Custom Computing Machines*, 216–225.

HAUCK, S. 1995. Multi-FPGA systems. Ph.D. dissertation, Univ. Washington, Dept. of C.S.&E.

HAUCK, S. 1998a. Configuration prefetch for single context reconfigurable coprocessors. *ACM/SIGDA International Symposium on FPGAs*, 65–74.

HAUCK, S. 1998b. The roles of FPGAs in reprogrammable systems. *Proc. IEEE 86*, 4, 615–638.

HAUCK, S. AND AGARWAL A. 1996. Software technologies for reconfigurable systems. Dept. of ECE Technical Report, Northwestern Univ. Available online at http://www.ee.washington.edu/faculty/hauck/publications.html.

HAUCK, S. AND BORRIELLO, G. 1997. Pin assignment for multi-FPGA systems. *IEEE Trans. Comput. Aid. Desi. Integ. Circ. Syst. 16*, 9, 956–964.

HAUCK, S., BORRIELLO, G., AND EBELING, C. 1998a. Mesh routing topologies for multi-FPGA systems. *IEEE Trans. VLSI Syst. 6*, 3, 400–408.

HAUCK, S., FRY, T. W., HOSLER, M. M., AND KAO, J. P. 1997. The Chimaera reconfigurable functional unit. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 87–96.

HAUCK, S., LI, Z., AND SCHWABE, E. 1998b. Configuration compression for the Xilinx XC6200 FPGA. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 138–146.

HAUCK, S. AND WILSON, W. D. 1999. Runlength compression techniques for FPGA configurations. Dept. of ECE Technical Report, Northwestern Univ. Available online at http://www.ee.washington.edu/faculty/hauck/publications.html.

HAUSER, J. R. AND WAWRZYNEK, J. 1997. Garp: A MIPS processor with a reconfigurable coprocessor. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 12–21.

HAYNES, S. D. AND CHEUNG, P. Y. K. 1998. A reconfigurable multiplier array for video image processing tasks, suitable for embedding in an FPGA structure. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 226–234.

HEILE, F. AND LEAVER, A. 1999. Hybrid product term and LUT based architectures using embedded memory blocks. *ACM/SIGDA International Symposium on FPGAs*, 13–16.

HUANG, W. J., SAXENA, N., AND MCCLUSKEY, E. J. 2000. A reliable LZ data compressor on reconfigurable coprocessors. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 249–258.

HUELSBERGEN, L. 2000. A representation for dynamic graphs in reconfigurable hardware and its application to fundamental graph

algorithms. *ACM/SIGDA International Symposium on FPGAs*, 105–115.

HUTCHINGS, B. L. 1997. Exploiting reconfigurability through domain-specific systems. Lecture Notes in Computer Science 1304—*Field-Programmable Logic and Applications*. W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 193–202.

HUTCHINGS, B., BELLOWS, P., HAWKINS, J., HEMMERT, S., NELSON, B., AND RYTTING, M. 1999. A CAD suite for high-performance FPGA design. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 12–24.

HWANG, T. T., OWENS, R. M., IRWIN, M. J., AND WANG, K. H. 1994. Logic synthesis for field-programmable gate arrays. *IEEE Trans. Comput. Aid. Des. Integ. Circ. Syst. 13*, 10, 1280–1287.

INUANI, M. K. AND SAUL, J. 1997. Technology mapping of heterogeneous LUT-based FPGAs. Lecture Notes in Computer Science 1304—*Field-Programmable Logic and Applications*. W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 223–234.

JACOB, J. A. AND CHOW, P. 1999. Memory interfacing and instruction specification for reconfigurable processors. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 145–154.

JEAN, J. S. N., TOMKO, K., YAVAGAL, V., SHAH, J., AND COOK R. 1999. Dynamic reconfiguration to support concurrent applications. *IEEE Trans. Comput. 48*, 6, 591–602.

KASTRUP, B., BINK, A., AND HOOGERBRUGGE, J. 1999. ConCISe: A compiler-driven CPLD-based instruction set accelerator. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 92–101.

KHALID, M. A. S. 1999. Routing architecture and layout synthesis for multi-FPGA systems. Ph.D. dissertation, Dept. of ECE, Univ. Toronto.

KHALID, M. A. S. AND ROSE, J. 1998. A hybrid complete-graph partial-crossbar routing architecture for multi-FPGA systems. *ACM/SIGDA International Symposium on FPGAs*, 45–54.

KIM, H. J. AND MANGIONE-SMITH, W. H. 2000. Factoring large numbers with programmable hardware. *ACM/SIGDA International Symposium on FPGAs*, 41–48.

KIM, H. S., SOMANI, A. K., AND TYAGI, A. 2000. A reconfigurable multi-function computing cache architecture. *ACM/SIGDA International Symposium on FPGAs*, 85–94.

KRESS, R., HARTENSTEIN, R. W., AND NAGELDINGER, U. 1997. An operating system for custom computing machines based on the Xputer paradigm. Lecture Notes in Computer Science 1304—*Field-Programmable Logic and Applications*. W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 304–313.

KRUPNOVA, H., RABEDAORO, C., AND SAUCIER, G. 1997. Synthesis and floorplanning for large hierarchical FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 105–111.

LAI, Y. T. AND WANG, P. T. 1997. Hierarchical interconnection structures for field programmable gate arrays. *IEEE Trans. VLSI Syst. 5*, 2, 186–196.

LAUFER, R., TAYLOR, R. R., AND SCHMIT, H. 1999. PCI-PipeRench and the SwordAPI: A system for stream-based reconfigurable computing. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 200–208.

LEE, Y. S. AND WU, A. C. H. 1997. A performance and routability-driven router for FPGA's considering path delays. *IEEE Trans. CAD Integ. Circ. Syst. 16*, 2, 179–185.

LEONARD, J. AND MANGIONE-SMITH, W. H. 1997. A case study of partially evaluated hardware circuits: Key-specific DES. Lecture Notes in Computer Science 1304—*Field-Programmable Logic and Applications*. W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 151–160.

LEUNG, K. H., MA, K. W., WONG, W. K., AND LEONG, P. H. W. 2000. FPGA Implementation of a microcoded elliptic curve cryptographic processor. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 68–76.

LEWIS, D. M., GALLOWAY, D. R., VAN IERSSEL, M., ROSE, J., AND CHOW, P. 1997. The Transmogrifier-2: A 1 million gate rapid prototyping system. *ACM/SIGDA International Symposium on FPGAs*, 53–61.

LI, Y., CALLAHAN, T., DARNELL, E., HARR, R., KURKURE, U., AND STOCKWOOD, J. 2000a. Hardware-software co-design of embedded reconfigurable architectures. *Design Automation Conference*, 507–512.

LI, Z., COMPTON, K., AND HAUCK, S. 2000b. Configuration caching for FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 22–36.

LI, Z. AND HAUCK, S. 1999. Don't care discovery for FPGA configuration compression. *ACM/SIGDA International Symposium on FPGAs*, 91–98.

LIN, X., DAGLESS, E., AND LU, A. 1997. Technology mapping of LUT based FPGAs for delay optimisation. Lecture Notes in Computer Science 1304—*Field-Programmable Logic and Applications*. W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 245–254.

LIU, H. AND WONG, D. F. 1999. Circuit partitioning for dynamically reconfigurable FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 187–194.

LUCENT TECHNOLOGIES, INC. 1998. *FPGA Data Book*. Lucent Technologies, Inc., Allentown, PA.

LUK, W., SHIRAZI, N., AND CHEUNG, P. Y. K. 1997a. Compilation tools for run-time reconfigurable

designs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 56–65.

LUK, W., SHIRAZI, N., GUO, S. R., AND CHEUNG, P. Y. K. 1997b. Pipeline morphing and virtual pipelines. Lecture Notes in Computer Science 1304—*Field-Programmable Logic and Applications*. W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 111–120.

LYSAGHT, P. AND STOCKWOOD, J. 1996. A simulation tool for dynamically reconfigurable field programmable gate arrays. *IEEE Trans. VLSI Syst. 4*, 3, 381–390.

MAK, W. K. AND WONG, D. F. 1997. Board-level multi net routing for FPGA-based logic emulation. *ACM Trans. Des. Automat. Elect. Syst. 2*, 2, 151–167.

MANGIONE-SMITH, W. H. 1999. ATR from UCLA. *Personal Commun.*

MANGIONE-SMITH, W. H., HUTCHINGS, B., ANDREWS, D., DEHON, A., EBELING, C., HARTENSTEIN, R., MENCER, O., MORRIS, J., PALEM, K., PRASANNA, V. K., AND SPAANENBURG, H. A. E. 1997. Seeking solutions in configurable computing. *IEEE Comput. 30*, 12, 38–43.

MARSHALL, A., STANSFIELD, T., KOSTARNOV, I., VUILLEMIN, J., AND HUTCHINGS, B. 1999. A reconfigurable arithmetic array for multimedia applications. *ACM/SIGDA International Symposium on FPGAs*, 135–143.

MCKAY, N. AND SINGH, S. 1999. Debugging techniques for dynamically reconfigurable hardware. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 114–122.

MCMURCHIE, L. AND EBELING, C. 1995. Pathfinder: A negotiation-based performance-driven router for FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 111–117.

MENCER, O., MORF, M., AND FLYNN, M. J. 1998. PAM-blox: High performance FPGA design for adaptive computing. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 167–174.

MIYAMORI, T. AND OLUKOTUN, K. 1998. A quantitative analysis of reconfigurable coprocessors for multimedia applications. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2–11.

MORITZ, C. A., YEUNG, D., AND AGARWAL, A. 1998. Exploring optimal cost performance designs for Raw microprocessors. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 12–27.

NAM, G. J., SAKALLAH, K. A., AND RUTENBAR, R. A. 1999. Satisfiability-based layout revisited: detailed routing of complex FPGAs via search-based boolean SAT. *ACM/SIDGA International Symposium on FPGAs*, 167–175.

PAN, P. AND LIN, C. C. 1998. A new retiming-based technology mapping algorithm for LUT-based FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 35–42.

PAYNE, R. 1997. Run-time parameterised circuits for the Xilinx XC6200. Lecture Notes in Computer Science 1304—*Field-Programmable Logic and Applications*. W. Luk, P. Y. K. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 161–172.

PURNA, K. M. G. AND BHATIA, D. 1999. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Trans. Comput. 48*, 6, 579–590.

QUICKTURN, A CADENCE COMPANY. 1999a. *System Realizer*™. Available online at http://www.quickturn.com/products/systemrealizer.htm. Quickturn, A Cadence Company, San Jose, CA.

QUICKTURN, A CADENCE COMPANY. 1999b. *Mercury*™ *Design Verification System Technology Backgrounder*. Available online at http://www.quickturn.com/products/mercury_backgrounder.htm. Quickturn, A Cadence Company, San Jose, CA, 1999.

RAZDAN, R. AND SMITH, M. D. 1994. A high-performance microarchitecture with hardware-programmable functional units. *International Symposium on Microarchitecture*, 172–180.

RENCHER, M. AND HUTCHINGS, B. L. 1997. Automated target recognition on SPLASH2. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 192–200.

ROSE, J., EL GAMAL, A., AND SANGIOVANNI-VINCENTELLI, A. 1993. Architecture of field-programmable gate arrays. *Proc. IEEE 81*, 7, 1013–1029.

RUPP, C. R., LANDGUTH, M., GARVERICK, T., GOMERSALL, E., HOLT, H., ARNOLD, J. M., AND GOKHALE, M. 1998. The NAPA adaptive processing architecture. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 28–37.

SANGIOVANNI-VINCENTELLI, A., EL GAMAL, A., AND ROSE, J. 1993. Synthesis methods for field programmable gate arrays. *Proc. IEEE 81*, 7, 1057–1083.

SANKAR, Y. AND ROSE, J. 1999. Trading quality for compile time: Ultra-fast placement for FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 157–166.

SCALERA, S. M. AND VAZQUEZ, J. R. 1998. The design and implementation of a context switching FPGA. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 78–85.

SELVIDGE, C., AGARWAL, A., DAHL, M., AND BABB J. 1995. TIERS: Topology IndependEnt Pipelined Routing and Scheduling for VirtualWire™ Compilation. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 25–31.

SENOUCI, S. A., AMOURA, A., KRUPNOVA, H., AND SAUCIER, G. 1998. Timing driven floorplanning on programmable hierarchical targets. *ACM/SIGDA International Symposium on FPGAs*, 85–92.

SHAHOOKAR, K. AND MAZUMDER, P. 1991. VLSI cell placement techniques. *ACM Comput. Surv. 23*, 2, 145–220.

SHI, J. AND BHATIA, D. 1997. Performance driven floorplanning for FPGA based designs. *ACM/SIGDA International Symposium on FPGAs*, 112–118.

SHIRAZI, N., LUK, W., AND CHEUNG, P. Y. K. 1998. Automating production of run-time reconfigurable designs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 147–156.

SLIMANE-KADI, M., BRASEN, D., AND SAUCIER, G. 1994. A fast-FPGA prototyping system that uses inexpensive high-performance FPIC. *ACM/SIGDA Workshop on Field-Programmable Gate Arrays*.

SOTIRIADES, E., DOLLAS, A., AND ATHANAS, P. 2000. Hardware-software codesign and parallel implementation of a Golomb ruler derivation engine. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 227–235.

STOHMANN, J. AND BARKE, E. 1996. An universal CLA adder generator for SRAM-based FPGAs. Lecture Notes in Computer Science 1142—*Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*. R. W. Hartenstein and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 44–54.

SWARTZ, J. S., BETZ, V., AND ROSE, J. 1998. A fast routability-driven router for FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 140–149.

SYNOPSYS, INC. 2000. *CoCentric System C Compiler*. Synopsys, Inc., Mountain View, CA.

SYNPLICITY, INC. 1999. *Synplify User Guide Release 5.1*. Synplicity, Inc., Sunnyvale, CA.

TAKAHARA, A., MIYAZAKI, T., MUROOKA, T., KATAYAMA, M., HAYASHI, K., TSUTSUI, A., ICHIMORI, T., AND FUKAMI, K. 1998. More wires and fewer LUTs: A design methodology for FPGAs. *ACM/SIGDA International Symposium on FPGAs*, 12–19.

THAKUR, S., CHANG, Y. W., WONG, D. F., AND MUTHUKRISHNAN, S. 1997. Algorithms for an FPGA switch module routing problem with application to global routing. *IEEE Trans. CAD Integ. Circ. Syst. 16*, 1, 32–46.

TOGAWA, N., YANAGISAWA, M., AND OHTSUKI, T. 1998. Maple-OPT: A performance-oriented simultaneous technology mapping, placement, and global gouting algorithm for FPGA's. *IEEE Trans. CAD Integ. Circ. Syst. 17*, 9, 803–818.

TRIMBERGER, S. 1998. Scheduling designs into a time-multiplexed FPGA. *ACM/SIGDA International Symposium on FPGAs*, 153–160.

TRIMBERGER, S., CARBERRY, D., JOHNSON, A., AND WONG, J. 1997a. A time-multiplexed FPGA. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 22–28.

TRIMBERGER, S., DUONG, K., AND CONN, B. 1997b. Architecture issues and solutions for a high-capacity FPGA. *ACM/SIGDA International Symposium on FPGAs*, 3–9.

TSU, W., MACY, K., JOSHI, A., HUANG, R., WALKER, N., TUNG, T., ROWHANI, O., GEORGE, V., WAWRZYNEK, J., AND DEHON, A. 1999. HSRA: High-speed, hierarchical synchronous reconfigurable array. *ACM/SIGDA International Symposium on FPGAs*, 125–134.

VAHID, F. 1997. I/O and performance tradeoffs with the FunctionBus during multi-FPGA partitioning. *ACM/SIGDA International Symposium on FPGAs*, 27–34.

VARGHESE, J., BUTTS, M., AND BATCHELLER, J. 1993. An efficient logic emulation system. *IEEE Trans. VLSI Syst. 1*, 2, 171–174.

VASILKO, M. AND CABANIS, D. 1999. Improving simulation accuracy in design methodologies for dynamically reconfigurable logic systems. *IEEE Sympos. Field-Prog. Cust. Comput. Mach.* 123–133.

VUILLEMIN, J., BERTIN, P., RONCIN, D., SHAND, M., TOUATI, H., AND BOUCARD, P. 1996. Programmable active memories: Reconfigurable systems come of age. *IEEE Trans. VLSI Syst. 4*, 1, 56–69.

WANG, Q. AND LEWIS, D. M. 1997. Automated field-programmable compute accelerator design using partial evaluation. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 145–154.

WEINHARDT, M. AND LUK, W. 1999. Pipeline vectorization for reconfigurable systems. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 52–62.

WILTON, S. J. E. 1998. SMAP: Heterogeneous technology mapping for area reduction in FPGAs with embedded memory arrays. *ACM/SIGDA International Symposium on FPGAs*, 171–178.

WIRTHLIN, M. J. AND HUTCHINGS, B. L. 1995. A dynamic instruction set computer. *IEEE Symposium on FPGAs for Custom Computing Machines*, 99–107.

WIRTHLIN, M. J. AND HUTCHINGS, B. L. 1996. Sequencing run-time reconfigured hardware with software. *ACM/SIGDA International Symposium on FPGAs*, 122–128.

WIRTHLIN, M. J. AND HUTCHINGS, B. L. 1997. Improving functional density through run-time constant propagation. *ACM/SIGDA International Symposium on FPGAs*, 86–92.

WITTIG, R. D. AND CHOW, P. 1996. OneChip: An FPGA processor with reconfigurable logic. *IEEE Symposium on FPGAs for Custom Computing Machines*, 126–135.

WOOD, R. G. AND RUTENBAR, R. A. 1997. FPGA routing and routability estimation via Boolean satisfiability. *ACM/SIGDA International Symposium on FPGAs*, 119–125.

WU, Y. L. AND MAREK-SADOWSKA, M. 1997. Routing for array-type FPGA's. *IEEE Trans. CAD Integ. Circ. Syst. 16*, 5, 506–518.

XILINX, INC. 1994. *The Programmable Logic Data Book*. Xilinx, Inc., San Jose, CA.

XILINX, INC. 1996. *XC6200: Advance Product Specification*. Xilinx, Inc., San Jose, CA.

XILINX, INC. 1997. *LogiBLOX: Product Specification*. Xilinx, Inc., San Jose, CA.

XILINX, INC. 1999. *Virtex^{TM} 2.5 V Field Programmable Gate Arrays: Advance Product Specification*. Xilinx, Inc., San Jose, CA.

XILINX, INC. 2000. *Press Release: IBM and Xilinx Team to Create New Generation of Integrated Circuits*. Xilinx, Inc., San Jose, CA.

XILINX, INC. 2001. *Virtex-II 1.5V Field Programmable Gate Arrays: Advance Product Specification*. Xilinx, Inc., San Jose, CA.

YASAR, G., DEVINS, J., TSYRKINA, Y., STADTLANDER, G., AND MILLHAM, E. 1996. Growable FPGA macro generator. Lecture Notes in Computer Science 1142—*Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*. R. W. Hartenstein and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 307–326.

YI, K. AND JHON, C. S. 1996. A new FPGA technology mapping approach by cluster merging. Lecture Notes in Computer Science 1142—*Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*. R. W. Hartenstein and M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 366-370.

ZHONG, P., MARTINOSI, M., ASHAR, P., AND MALIK, S. 1998. Accelerating Boolean satisfiability with configurable hardware. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 186–195.

Received May 2000; revised October 2001 and January 2002; accepted February 2002