An Introduction to Reconfigurable Computing

Katherine Compton
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL USA
kati@ece.nwu.edu

Scott Hauck
Department of Electrical Engineering
University of Washington
Seattle, WA USA
hauck@ee.washington.edu

Abstract

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 introduction to reconfigurable computing, we give an overview of the hardware architectures of reconfigurable computing machines, and the software that targets these machines, such as compilation tools. Finally, we consider the issues involved in run-time reconfigurable systems, which re-use the configurable hardware during program execution.

Introduction

There are two primary methods in traditional computing for the execution of algorithms. The first is to use an Application Specific Integrated Circuit, or ASIC, to perform the operations in hardware. Because these ASICs are designed specifically to perform a given computation, they are very fast and efficient when executing the exact computation for which they were designed. However, after fabrication the circuit cannot be altered. Microprocessors are 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 suffers, and is far below that of an ASIC. The processor must read each instruction from memory, determine its meaning, and only then execute it. This results in a high execution overhead for each individual operation. 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.

This paper presents a brief overview 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 FPGAs in general, followed by an exploration 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 tradeoffs 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 overview does not seek to cover every technique and research project in the area of reconfigurable computing. Instead, it hopes to serve as a brief 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 other techniques and systems elsewhere [1, 2, 3, 4]

Field-Programmable Gate Arrays

FPGAs were originally created to serve as a hybrid device between PALs and Mask-Programmable Gate Arrays (MPGAs). Like PALs, they are fully electrically programmable, meaning that the Non-Recurring Engineering (NRE) costs are amortized, and they can be customized nearly instantaneously. Like MPGAs they can implement very complex computations on a single chip, with million gate devices currently in production. Because of these features, FPGAs are often primarily viewed as glue-logic replacement and rapid-prototyping vehicles. However, as we will show throughout this paper, 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 are SRAM-programmable (Figure 1 left). This means that SRAM 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 as easily as a standard static RAM. To configure the routing on an FPGA, typically a passgate structure is employed (see Figure 1 middle). 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.

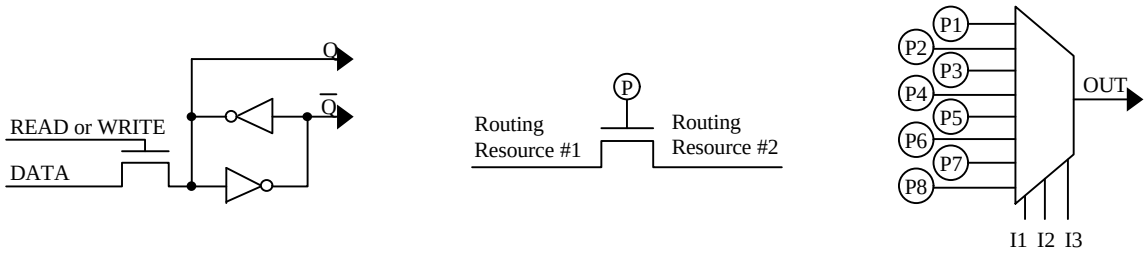

Figure 1: Programming bit for SRAM-based FPGAs [5] (left), a programmable routing connection (middle) and a 3-input LUT (right).

Lookup-tables (LUTs), which are essentially small memories provided for computing arbitrary logic functions, are generally used as the computational structures in an FPGA. These elements can compute any function of N inputs (where N is the number of control signals for the LUT's mux) by programming the 2^N programming bits with the truth table of the desired function (see Figure 1 right). 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. The typical FPGA has a logic block with one 4-input LUT, an optional D flip-flop (DFF), and some form of fast carry logic. 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.

Routing structures for FPGAs have generally focused on island-style layouts. In this type of structure, the logic blocks are surrounded by general routing channels, running both horizontally and vertically. The input and output signals of the blocks are connected to the channels through programmable connection blocks. Switchboxes are used at the juncture of horizontal and vertical wires to allow signals to change routing direction at those points. Using these structures, relatively arbitrary interconnections can be achieved.

Hardware

There are many different architectures designed for use in reconfigurable computing. One of the primary variations between these is the degree of coupling (if any) with a host microprocessor. Programmable logic tends to be inefficient at implementing certain types of operations, such as loop and branch control. In order to most efficiently run an application in a reconfigurable computing system, 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 (see Figure 2).

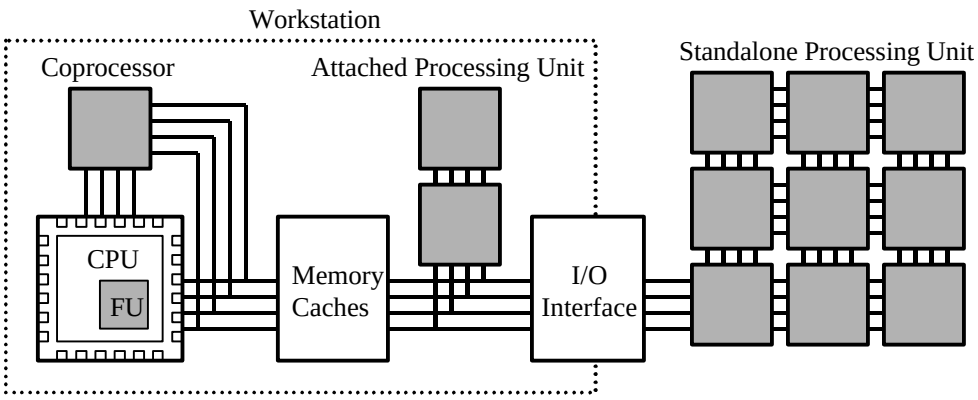

Figure 2: Different levels of coupling in a reconfigurable system. Reconfigurable logic is shaded.

First, reconfigurable hardware can be used solely to provide reconfigurable functional units within a host processor. This allows for a traditional programming environment with the 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. 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.

Third, an attached reconfigurable processing unit behaves as if it is an additional processor in a multi-processor system. 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. 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 standalone processing unit. 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.

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.

In addition to the level of coupling, 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 look up 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.

Very fine-grained logic blocks (such as those operating only on 2 or 3 one-bit values) are useful for 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, 128 bit addition, etc). Performing these types of computation on a traditional microprocessor wastes calculation effort for the case of very small operands, and incurs multi-instruction overhead for the case of very large operands. The reconfigurable logic performs exactly the calculation needed.

Several reconfigurable systems use a medium-sized granularity of logic block. A number of these architectures operate on two or more 4-bit wide data words, in particular. This increases the total number of input lines to the circuit, and provides more efficient computational structures for more complex problems. 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 also be used to efficiently implement more complex operations such as finite state machines.

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 [6], is composed of 16-bit adders, multipliers, and registers. If only three 1-bit values are required, then the use of this architecture suffers an unnecessary area and speed overhead, as all 16 bits 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.

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.

There are two primary methods to provide both local and global routing resource. The first is the use of segmented routing. In segmented routing, short wires accommodate local communications traffic. These short wires can be connected together using switchboxes to emulate longer wires. Optionally, longer wires may also be included, and signals may transfer between local and distance routing at connection blocks. Hierarchical routing provides local routing within a cluster, and longer wires at the boundaries connect the different clusters together. Hierarchical structures are optimized for situations where the most communication should be local and only a limited amount of communication will traverse long distances.

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 [2], including meshes, crossbars, and variants on these structures. Because of the need for efficient communication between the FPGAs, the determining the inter-chip routing topology is a very important step in the design of a multi-FPGA system.

Once a circuit has been configured onto the reconfigurable hardware, it is ready to be used by the host processor during program execution. The runtime operation of a reconfigurable system occurs in two distinct phases: configuration and execution. The configuration 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 startup 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. Some systems suspend the execution of the host processor during execution on the reconfigurable hardware. Others allow for simultaneous execution with techniques similar to the use of fork/join primitives in multiprocessor programming.

Software

Although reconfigurable hardware has been shown to have significant performance benefits in program execution, 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 for 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 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, and therefore makes the use of reconfigurable hardware more accessible to general application programmers. For a more detailed description of software and compilation issues in reconfigurable computing refer to [1, 7].

Both for manual and automatic circuit creation, the design process must proceed through a number of distinct phases. 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 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.

When targeting 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 loops are more efficiently implemented in software, while fixed datapath operations may be more optimally executed in hardware. This can be performed manually, by delineating sections of the program within the source code to be executed on the hardware. Alternately, an automatic compilation tool can determine where areas of the code should be executed.

For descriptions in a high level language (HLL), such as C/C++ or Java, or ones using complex building blocks, the code targeted to the reconfigurable hardware must be compiled into a netlist of gate-level components. For the HLL implementations this involves 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-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 sub-functions, each of which can be mapped to a single LUT.

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. This technique therefore separates the overall placement problem into a set of smaller localized sub-problems for the detailed placer to solve, reducing the overall workload.

After floorplanning, the individual logic blocks are placed into specific logic cells. One algorithm that is commonly used is the simulated annealing technique [8]. This method takes an initial placement of the system, which can be generated 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 at each iteration. 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.

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, and concentrate on minimizing congestion within the available tracks. 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.

When reconfigurable systems use more than one FPGA to form the complete reconfigurable hardware, there are additional compilation issues to deal with. The design must first be partitioned into the different FPGA chips. This is generally done by placing each highly connected portion 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. 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 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.

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 helpful to swap different configurations in and out of the reconfigurable hardware as they are needed during program execution, performing a run-time reconfiguration of the hardware. 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 in the run-time reconfigurable systems. This can lead to an overall improvement in performance.

There are a few different configuration memory styles that can be used with reconfigurable systems (Figure 3). A single context device is a serially programmed chip that requires a complete reconfiguration in order to change any of the programming bits. Most commercial FPGAs are of this variety. To implement run-time reconfiguration on this type of device, configurations must be grouped into full contexts, and the complete contexts are swapped in and out of the hardware as needed.

A multi-context device has multiple layers of programming bits, where each layer can be active at a different point in time. An advantage of the multi-context 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 multi-context design does allow for background loading, permitting one context to be configuring while another is in execution. Each context of a multi-context device can be viewed as a separate single-context device.

Devices that can be selectively programmed without a complete reconfiguration are called partially reconfigurable. The partially reconfigurable FPGA 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. This allows configurations which occupy only a part of the total area to be configured onto the array without removing all of the configurations already present. Furthermore, individual configurations can be selectively modified based on run-time conditions, such as changing registered constant values or a constant coefficient multiplier structure over time. These small reconfigurations require much less time than a full-chip reconfiguration due to the reduced data traffic.

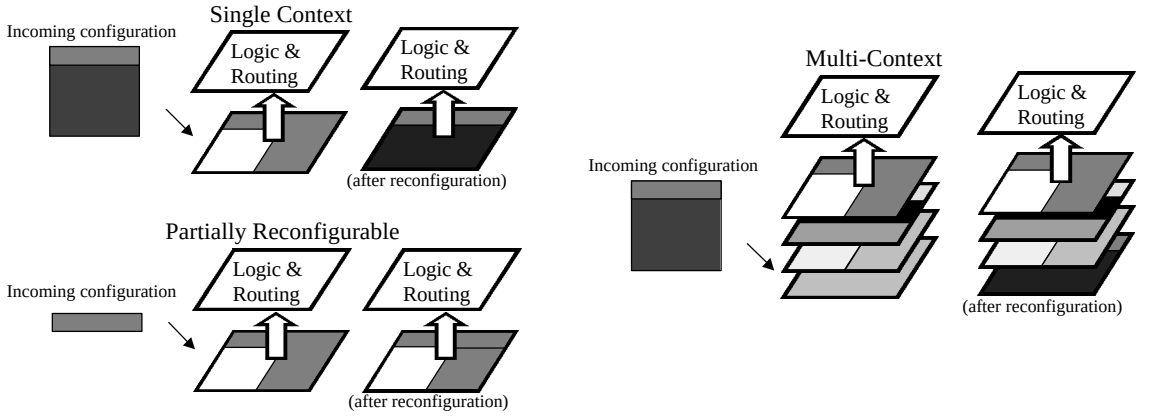

Figure 3: The different basic models of reconfigurable computing: single context, multi-context, and partially reconfigurable. Each of these designs is shown performing a reconfiguration.

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. 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 configuration overhead. 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 some cases over 98% of execution time can be spent in reconfiguration [9]. 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 of the reconfigurable device 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, the number of times a reconfiguration is necessary can be reduced through hardware optimizations that help prevent programmed configurations that will be reused from being unnecessarily replaced by incoming configurations. 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 could provide a faster reconfiguration.

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, the overall application can be greatly accelerated. This is because reconfigurable computing combines 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 execution time of an application. Similar to an ASIC, reconfigurable systems provide a method to map circuits into hardware, achieving far greater performance than software as a result of bypassing the fetch-decode-execute cycle of traditional microprocessors, and parallel execution of multiple operations.

Reconfigurable hardware systems come in many forms, from a configurable functional unit integrated directly into a CPU, to a reconfigurable co-processor 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.

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.

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, the performance of a reconfigurable system can approach or even surpass that of an ASIC.

Reconfigurable computing systems have shown the ability to greatly accelerate program execution, 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.

Acknowledgments

This research was funded in part by DARPA contracts DABT63-97-C-0035 and F30602-98-0144, and a grant from Motorola, Inc. Katherine Compton was supported by an NSF Fellowship.

References

[1] K. Compton, S. Hauck, "Configurable Computing: A Survey of Systems and Software", *Northwestern University, Dept. of ECE Technical Report*, 1999.

[2] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems" *Proceedings of the IEEE*, Vol. 86, No. 4, pp. 615-638, April 1998.

[3] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prasanna, H. A. E. Spaanenburg, "Seeking Solutions in Configurable Computing", *IEEE Computer*, Vol. 30, No. 12, pp. 38-43, December 1997.

[4] J. Rose, A. El Gamal, A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays", *Proceedings of the IEEE*, Vol. 81, No. 7, pp. 1013-1029, July 1993.

[5] *The Programmable Logic Data Book*, San Jose, CA: Xilinx, Inc., 1994.

[6] C. Ebeling, D. C. Cronquist, P. Franklin, "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. Berlin, Germany: Springer-Verlag, pp. 126-135, 1996.

[7] S. Hauck, A. Agarwal, "Software Technologies for Reconfigurable Systems", *Northwestern University, Dept. of ECE Technical Report*, 1996.

[8] K. Shahookar, P. Mazumder, "VLSI Cell Placement Techniques", *ACM Computing Surveys*, Vol. 23, No. 2, pp. 145-220, June 1991.

[9] W. H. Mangione-Smith, "ATR from UCLA", *Personal Communications*, 1999.