

Programming Architectures For Run-Time Reconfigurable Systems

Master's Thesis

December 1999

*Katherine Compton
Dept. of ECE
Northwestern University
Evanston, IL USA*

*Advisor:
Scott Hauck
University of Washington
Seattle, WA USA*

Programming Architectures For Run-Time Reconfigurable Systems

Katherine Compton
Department of ECE
Northwestern University
Evanston, IL USA
kati@ece.nwu.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. By mapping the compute-intensive sections of an application to reconfigurable hardware, custom computing systems exhibit significant speedups over traditional microprocessors. However, the number and frequency of these hardware-mapped sections of code are limited by the requirement that the speedups provided must outweigh the considerable time cost of configuration. The ability to relocate and defragment configurations on an FPGA can dramatically decrease the overall configuration overhead incurred by the use of the reconfigurable hardware. This increases the viability of mapping portions of the program that were previously considered to be too costly. We therefore explore the adaptation of a simplified Xilinx 6200 series FPGA for relocation and defragmentation. Because of the complexities involved with this structure, we also present a novel architecture designed from the ground up to provide relocation and defragmentation support with a negligible area increase over a generic partially reconfigurable FPGA.

Introduction

There are two primary methods in traditional computing for the execution of algorithms. One method 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. Instead, a re-design and re-fabrication of the chip is necessary 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.

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. 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, resulting in a high execution overhead for each operation.

Reconfigurable computing allows designers to harness the power of hardware while still providing the flexibility of software. Reconfigurable computing systems use Field Programmable Gate Arrays (FPGAs) or FPGA-like 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. 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.

Run-time reconfiguration expands upon the idea of reconfigurability by providing the ability to change the reconfigurable hardware not only between applications, but also within a single application. Over the course of program execution different configurations can be loaded into the FPGA to perform different hardware-optimized computations at different points in time. This allows a larger percentage of a program to be accelerated in hardware. However, the cost of reconfiguration is quite high. In some situations, configuration overhead can comprise over 98.5% of execution time [Smith99]. This amount of overhead has the potential to eclipse the benefits gained through use of the reconfigurable hardware. Therefore, it is essential to the principle of run-time reconfiguration to reduce the configuration overheads.

This thesis examines the effect of the programming architecture of an FPGA on the configuration overhead encountered in RTR applications. Three primary FPGA types are first compared to determine the relative configuration overhead cost. One of these models, the partially reconfigurable FPGA, is programmed in an addressable fashion. A variation on this type of FPGA is also presented which further lowers reconfiguration overheads. We add the ability to perform relocation of the configurations, which allows the final placement within the FPGA to be determined at runtime, and discuss how this idea may be applied to the Xilinx 6200, a commercial partially reconfigurable FPGA [Xilinx96]. We then present a new FPGA programming architecture that includes relocation as well as providing a mechanism for run-time defragmentation of the configurations within the array to consolidate unused resources. Using relocation and defragmentation together, the configuration overheads encountered when using a partially reconfigurable FPGA design can be further reduced, increasing the efficiency of run-time reconfiguration.

First we provide a short background in reconfigurable computing. Next, we present our area models used in comparison of the FPGA types, as well as the performance results of the comparison. In order to leverage the advantages of relocation, we then examine the refitting of the Xilinx 6200 architecture into a relocation-enabled FPGA. Next we discuss the difficulties involved in actually using the 6200 for relocation and defragmentation. Finally, we will propose a novel architecture designed specifically for partial reconfiguration, relocation and defragmentation.

Background – Reconfigurable Computing

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 type of computing is based upon Field Programmable Gate Arrays (FPGAs). These devices contain an array of computational elements whose functionality is determined through multiple SRAM configuration bits. These elements, also known as logic blocks, are connected using a set of routing resources that are also programmable. In this way, custom circuits can be mapped to the FPGA 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.

Field-Programmable Gate Arrays

Most current FPGAs are SRAM-programmable (Figure 1 left). 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 (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.

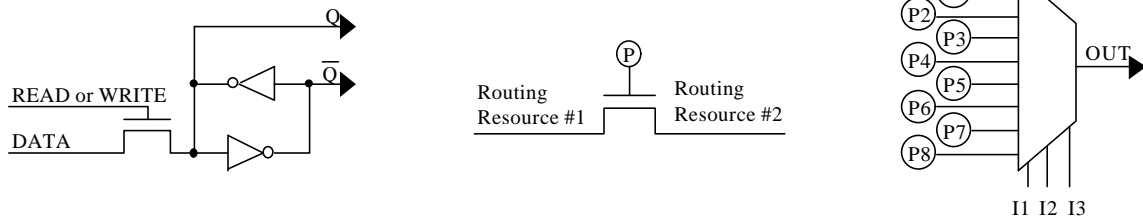


Figure 1: Programming bit for SRAM-based FPGAs [Xilinx94] (left), a programmable routing connection (middle) and a 3-input LUT (right).

In order to implement logic functions there are typically multiplexers with programming bits connected to the control and/or data inputs. These muxes 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 mux selecting whether to forward the latched or unlatched signal value. Thus, for systems that require stateholding the programming bits controlling the mux 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 can choose between other on-chip functionalities, such as fixed logic computation elements, memories, carry chains, or other functions.

Lookup-tables (LUTs), which are essentially small memories provided for computing arbitrary logic functions, can also be included. 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 an OR.

Routing structures for FPGAs have typically 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. Figure 2 illustrates the basics of this type of routing design. Using these structures, relatively arbitrary interconnections can be achieved.

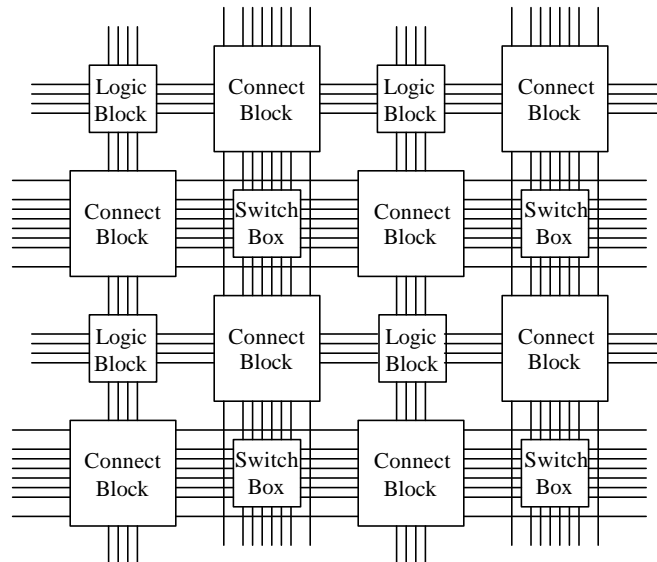


Figure 2: A generic island-style FPGA routing architecture.

Reconfigurable Hardware

There are many different architectures designed for use in reconfigurable computing. One of the primary variations between these architectures 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 variable-length loop and branch control. In order to most efficiently run an application in a reconfigurable computing system, the areas of the program that cannot easily be mapped to the reconfigurable logic are executed on a host microprocessor. Meanwhile, the areas 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 3).

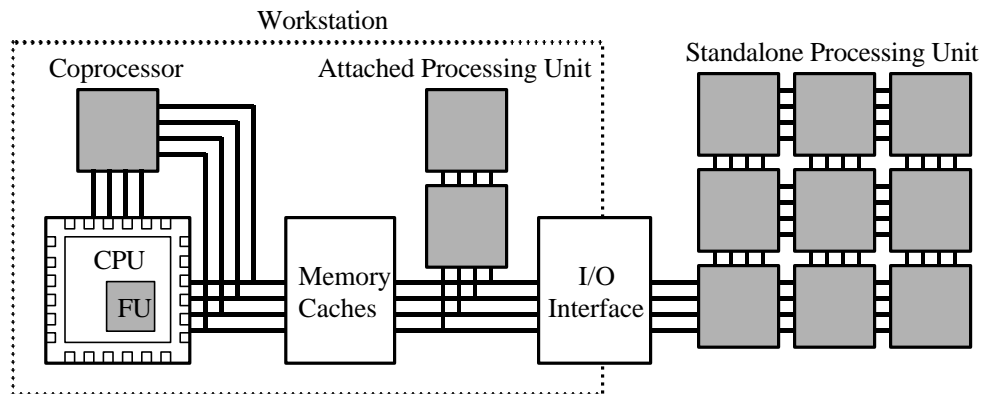


Figure 3: Different levels of coupling in a reconfigurable system. Reconfigurable logic is shaded.

First, reconfigurable hardware can be used solely to provide reconfigurable functional units within a host processor [Razdan94, Hauck97]. 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 [Wittig96, Hauser97, Miyamori98, Rupp98]. 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 coprocessor performs the actual computations independently of the main processor, and returns the results after completion. Although there is more communication overhead for the computation data and result values than with the reconfigurable functional unit, the coprocessor model can greatly reduce the frequency with which this overhead is encountered for long or repeated computations. Coupling with this method can also increase parallelism by permitting the host processor and the reconfigurable logic to execute simultaneously.

Third, an attached reconfigurable processing unit [Annapolis98, Laufer99] 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 [Quickturn99a, Quickturn99b]. This type of reconfigurable hardware communicates infrequently with a host processor (if present). The standalone processing unit 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 a 3-bit LUT is an example of a very fine grained computational element, and a 4-bit ALU is an example of a very 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) [Xilinx96, Altera98] are useful for bit-level manipulation of data, as can frequently be found in encryption and image processing applications. Several reconfigurable systems use a medium-sized granularity of logic block [Xilinx94, Hauser97, Haynes98, Lucent98, Marshall99]. 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. Very coarse-grained architectures [Ebeling96, Miyamori98, Moritz98] 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.

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 [Xilinx94]. 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 longer-distance routing at connection blocks. Hierarchical routing [Aggarwal94, Lai97] 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.

For any FPGA-based reconfigurable hardware that is coupled with a host processor, there are three distinct phases of operation: compilation, configuration and execution. The compilation step translates hardware circuit descriptions, (in the form of high level language programs, gate level diagrams, or somewhere in between) into actual configuration information to be written to the programming bits of the FPGA for execution. The first major phase of compilation requires technology mapping to translate the description into computation blocks that can be executed by the FPGA's logic elements. Next, the location of these blocks in the actual FPGA structure is determined in the placement stage. Finally, the routing phase determines how to connect these blocks together to reform the original computation. The resulting configuration information contains all the logic and routing programming information to place the configuration at the proper location within the FPGA.

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 the next section. Finally, the 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.

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, as shown in Figure 4. This concept is known as run-time reconfiguration.

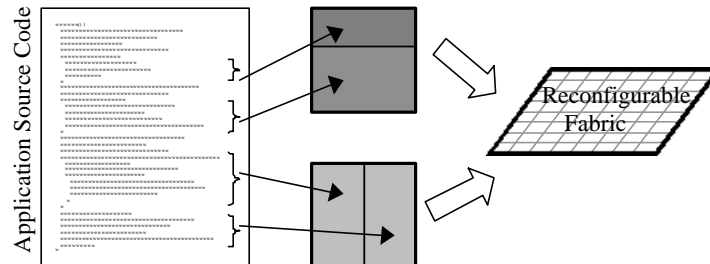


Figure 4: Some applications have more configurations than can fit in the available hardware. In this case, we would like to re-program the reconfigurable logic during run-time to allow all configurations to be executed in hardware in a time-multiplexed fashion.

Run-time reconfiguration is similar to the concept of virtual memory, and therefore the term “virtual hardware” is sometimes applied to this concept. 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 the contents of the reconfigurable hardware can be changed during runtime, more areas of an application can be mapped to the hardware. This increased usage of the reconfigurable hardware as a hardware accelerator leads to an overall improvement in performance.

There are a few different configuration memory styles that can be used with reconfigurable systems. A single context device is programmed using a serial stream of configuration information, and requires a complete reconfiguration in order to change any of the programming bits. A multi-context device has multiple layers of programming bits, each of which 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. 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. These programming architectures are described in more depth in a later section.

Fast Configuration

Because run-time reconfigurable systems involve reconfiguration during program execution, the reconfiguration must be done as efficiently and as quickly as possible in order to ensure that the overhead of the reconfiguration does not outweigh 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% [Wirthlin96] to 71% [Wirthlin95] of execution time is spent in reconfiguration, while in the UCLA ATR work this figure can rise to over 98.5% [Smith99]. 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.

Some of the previous work in fast reconfiguration is presented in the following section. Our work is also aimed at the reduction of configuration overheads, but instead through the examination of optimized programming architectures. These programming architectures can reduce not only the number of times a reconfiguration is necessary, but also potentially the amount of data sent in each communication.

Previous Methods

A number of different tactics for reducing configuration overhead have been explored. 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 reconfigurable hardware. Third, 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 faster reconfiguration for commonly used configurations.

Configuration Prefetching

Overlapping the actual configuration of the hardware with computations performed by the host processor can help to mask the requires milliseconds to seconds required for the reconfiguration. This overlapping 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 [Hauck98b] 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.

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 can be reduced by minimizing the amount of data transferred from the host processor to the reconfigurable array. Configuration compression can be used to compact this configuration information [Hauck98a, Hauck99, Li99].

One form of configuration compression has already been implemented in a commercial system. The Xilinx 6200 series of FPGA [Xilinx96] is a partially reconfigurable structure that contains wildcarding hardware, which provides a method to program multiple logic cells with a single address and data value. A special register is set 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 are 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. [Hauck98a] discusses the benefits of this hardware, while [Li99] covers 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 effectively compress the amount of data sent to the reconfigurable hardware. A configuration can possibly re-use configuration information already present on the array, such that only the areas differing in configuration values must be re-programmed. Therefore, configuration time can be reduced through the identification of these common components and the calculation of the incremental configurations that must be loaded [Luk97, Shirazi98].

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 as the reading of the configuration data from a file or main memory, a configuration cache can potentially reduce the costs of reconfiguration [Deshpande99]. By storing the configurations in fast memory near to the reconfigurable array, the data transfer during reconfiguration is accelerated, and the overall configuration time required is reduced.

Our Work

This research focuses on the actual programming architecture of the FPGA, and how it affects the configuration overhead. We first study the overheads shown by the three major programming paradigms for FPGAs: single context, partially reconfigurable and multi-context. Next we adapt the partially reconfigurable FPGA to include two new optimizations, relocation and defragmentation. We will demonstrate how these optimizations can reduce the configuration overhead even further over the partially reconfigurable FPGA, which is already an improvement over the single context, serially programmed FPGA.

Basic Programming Architectures

Traditional FPGA structures have been single-context, allowing only 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 and/or slow to efficiently implement run-time reconfiguration. The following discussion defines the single-context device, and further considers newer FPGA designs (multi-context and partially reconfigurable), along with their impact on run-time reconfiguration.

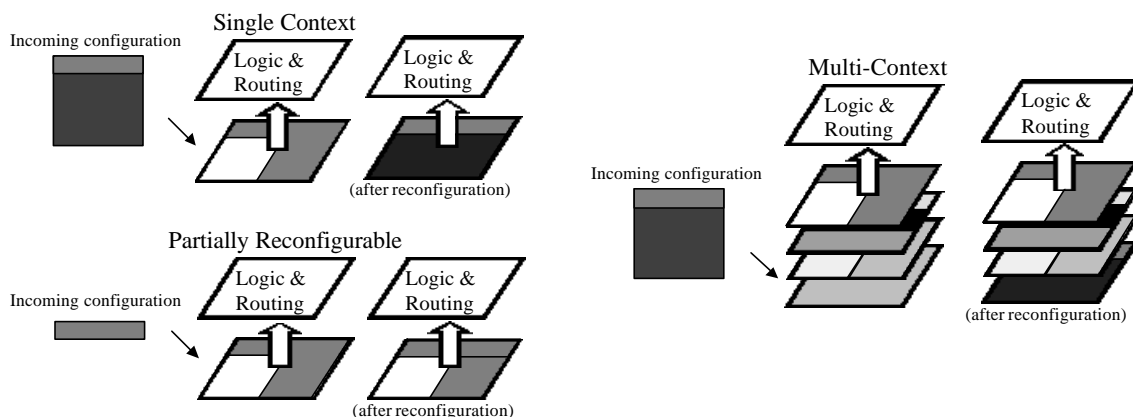


Figure 5: The different basic models of reconfigurable computing: single context, multi-context, and partially reconfigurable. Each of these designs is shown performing a reconfiguration.

Single Context

A single context FPGA is 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. This type of FPGA is therefore more suited for applications that can benefit from reconfigurable computing without run-time reconfiguration. Most current commercial FPGAs are of this style, including the Xilinx 4000 series [Xilinx94], the Altera Flex10K series [Altera98], and Lucent's Orca series [Lucent98]. A single context FPGA is depicted in Figure 5, top left.

In order to implement run-time reconfiguration using 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.

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 supported by a single context 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 changed at a given point in time, the entire array does not require reprogramming for each incoming configuration. Additionally, some applications require the updating of only a portion of a mapped circuit, while the rest should remain intact, as shown in Figure 5, bottom left. For example, in a filtering operation in signal processing, a set of constant values that change slowly over time may be re-initialized to a new value. But 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 RaPiD [Ebeling96], Chimaera [Hauck97], PipeRench [Cadambi98], and NAPA [Rupp98].

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. A full reconfiguration of the entire array is therefore slower than with 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, the fast configurations methods presented in a previous section can help reduce the configuration data traffic requirements.

Multi-Context

A multi-context FPGA includes multiple memory bits for each programming bit location. These memory bits can be thought of as multiple planes of configuration information, as shown in Figure 5 right. 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 multi-context device can be considered a multiplexed set of single-context devices, which requires that a context be fully reprogrammed to perform any modification. This systems 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 6 shows a multi-context memory bit, as used in [Trimberger97].

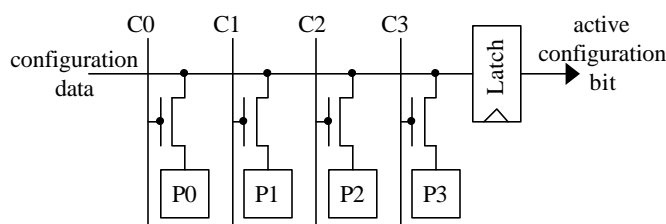


Figure 6: A four-bit multi-contexted programming bit [Trimberger97]. P0-P3 are the stored programming bits, while C0-C3 are the chip-wide control lines which select the context to program or activate.

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 in the 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 multi-context device at the same time.

Area Models

In order to compare the costs vs. benefits of the three basic programming architectures, we must compute the performance of each design given a fixed area resource. We have created area models for each of the programming architectures, which can be used to determine the area requirements. Because the area requirements of each programming model differ, the number of programming bits which can fit within the fixed area vary with the architecture. Therefore, the number of programming bits available to each programming architecture is used as an input parameter for the simulation program that determines the configuration overhead for each model for each of the benchmark programs.

To find the area requirements for the different programming models, we consider the major components of the programming architecture, such as actual programming memory, row decoders, and input/output tri-state buffers. Each of these components can be generated by replicating smaller tileable structures. We use tileable structures to allow us to parameterize the architecture and determine the areas in terms of the number of programming bits. These tiles were created by hand, and their sizes in λ^2 were tabulated (Figure 7). The full sizes of the components were then calculated by multiplying the size of the relevant tiles by the number of those tiles are required, given a particular number of rows and columns of programming bits. The layouts of the tiles were created using the Magic VLSI layout program.

Because we wish to determine how the structure of the programming architecture affects the total chip area of the FPGA, we also must account for the area occupied by the logic and routing. We use the assumption that for a single-context device, the programming structures occupy 25% of the total chip area [Trimberger98]. Therefore, three times the area required by the single context programming structure is used as an approximation of the fixed area occupied by the logic, routing, and I/O structures of the FPGA. Once we have the area in terms of the programming bits, we can set the area to a fixed size and compute the number of programming bits that will fit within that area for a given programming architecture.

Prog. Bit	Area (λ^2)
Serial	2275.5
Partial	1309.0
Multi (2)	9075.0
Multi (4)	13075.0
Multi (8)	21075.0

Decoders	Area (λ^2)
1Row decoder (per row), ≥ 2 address lines	$476 + 392 * \langle \# \text{ address lines} \rangle$
Column decoder (per column), < 3 address lines	3445
Column decoder (per column), ≥ 3 address lines	$2177.5 + 487.5 * \langle \# \text{ address lines} \rangle$
Context decoder (per context), 1 address line	1148
Context decoder (per context), ≥ 2 address lines	$476 + 392 * \langle \# \text{ address lines} \rangle$

Input / Output Tri-states	Area (λ^2)
Serial	0.0
Partial	11407.5
Multi (2)	14722.5
Multi (4)	11992.5
Multi (8)	11407.5

Figure 7: Tables of tileable component sizes

Structure Details

The single context FPGA area model is composed of a two-phase shift chain of programming bits, which forms the path for the input of configuration data. No other support structures are needed for this particular architecture. For N_{Row} rows and N_{Col} 32-bit columns of programming bits, the area of the programming structure in λ^2 for the single-context FPGA is:

$$\begin{aligned}
\text{single context area} &= \langle \# \text{ prog. bits} \rangle * \langle \text{prog. bit size} \rangle \\
&= NRow * (NCol * 32) * 2275.5 \\
&= NRow * NCol * 72816
\end{aligned}$$

The partially reconfigurable architecture is based upon a traditional addressable RAM structure. The programming bits are held in 5-transistor SRAM cells. The row and column decoders used to select the destination location of the configuration data are both pseudo-NMOS with precharging. Large output tri-state drivers enabled by the column decoder are required to magnify the weak signals provided by the SRAM cells when reading the configuration data off of the array. The input of data in 32-bit words is controlled by tri-state drivers enabled by the column decoder in conjunction with a write enable signal. Figure 8 demonstrates how the tileable structures were used to estimate the area of this design. For $NRow$ ($NRow \geq 4$) rows and $NCol$ ($NCol \geq 8$) 32-bit columns of programming bits, the area of the programming structure in λ^2 for the partially reconfigurable FPGA is:

$$\begin{aligned}
\text{partially reconfigurable area} &= \langle \# \text{ prog. bits} \rangle * \langle \text{prog. bit area} \rangle + \langle \text{row decoder area} \rangle \\
&\quad + \langle \text{column decoder area} \rangle + \langle \text{I/O tri-states area} \rangle \\
&= NRow * (NCol * 32) * 1309 + NRow * (476 + 392 \text{ ceil}(\lg NRow)) \\
&\quad + NCol * (2177.5 + 487.5 * \text{ceil}(\lg NCol)) + (NCol * 32) * 11407.5 \\
&= NRow * NCol * 41888 + NRow * 476 + NRow * \text{ceil}(\lg NRow) * 392 \\
&\quad + NCol * 367217.5 + NCol * \text{ceil}(\lg NCol) * 487.5
\end{aligned}$$

$$NRow * (476 + 392 * \lg NRow) \lambda^2$$

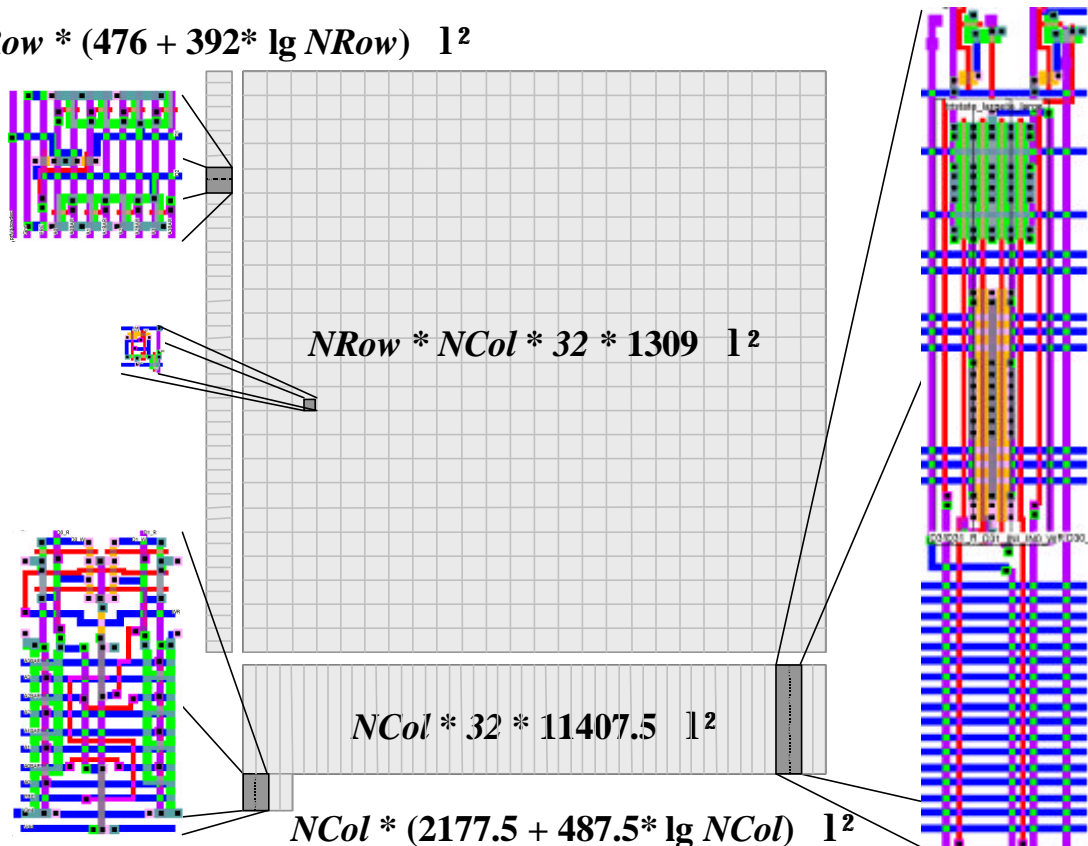


Figure 8: A diagram of how the estimation of the programming structure of the partially reconfigurable FPGA was calculated. The upper left structure is a row decoder. The small structure below it is the 5-transistor SRAM cell. The column decoder is in the lower left, and the right hand side shows the I/O tri-state buffers.

The multi-context design is based on the structure shown in Figure 6. Because we found (see Table 2) that the partially reconfigurable FPGA design required less total area than a single context design composed of a shift chain, we built each of the contexts in the multi-context device as a partially reconfigurable plane. In order to configure a particular location in the multi-context device, not only the row and column addresses but also the context address must be specified. The context decoder is the same basic structure as the row column decoder. A D latch holds the “active” programming bit value.

The device changes contexts by first disabling the transistor which connects the global configuration lines to the short local bus connecting the different contexts within a given programming bit structure. Next, the new context is enabled. To write to a context, the data is placed on the global lines by the input tri-state drivers that are enabled by the column decoder, and the row decoder enables the transistor connecting the global and local lines. The destination context must also be enabled by the context decoder in order to allow the data to be written to a storage bit.

The transistor connecting the local lines to the global lines of a programming bit is in series with the SRAM read/write enable transistor and the input tri-state driver. Therefore, this transistor needs to be quite large to ensure that the data contained in the SRAM cells are properly overwritten during a write period. We used logical effort [Sutherland99] to size the transistors appropriately. The SRAM read/write enable transistors are 12λ high with 2λ wide polysilicon. As much as possible, we have absorbed the size of the local/global connection transistor within the programming bits without increasing the programming bit size. In essence, the size of the transistors is amortized over the number of contexts. The input tri-state driver sizes were then calculated based upon these values.

For $NRow$ ($NRow \geq 4$) rows and $NCol$ ($NCol \geq 8$) 32-bit columns of programming bits, the area of the programming structure in λ^2 for a multi-contexted FPGA with 2 contexts is:

$$\begin{aligned}
\text{multi2 area} &= \langle \# \text{ prog. bits} \rangle * \langle \text{prog. bit area} \rangle + \langle \text{row decoder area} \rangle \\
&+ \langle \text{column decoder area} \rangle + \langle \text{context decoder area} \rangle \\
&+ \langle \text{I/O tri-states area} \rangle \\
&= NRow * (NCol * 32) * 9075 + NRow * (476 + 392 * \text{ceil}(\lg NRow)) \\
&+ NCol * (2177.5 + 487.5 * \text{ceil}(\lg NCol)) + 2 * 1148 \\
&+ (NCol * 32) * 14722.5 \\
&= NRow * NCol * 290400 + NRow * 476 + NRow * \text{ceil}(\lg NRow) * 392 \\
&+ NCol * 473297.5 + NCol * \text{ceil}(\lg NCol) * 487.5 + 2296
\end{aligned}$$

Adjusting the areas of the programming bit, the context decoder, and the I/O tri-states, the areas computed for the 4 and 8 context FPGAs are:

$$\begin{aligned}
\text{multi4 area} &= NRow * NCol * 418400 + NRow * 476 + NRow * \text{ceil}(\lg NRow) * 392 \\
&+ NCol * 385937.5 + NCol * \text{ceil}(\lg NCol) * 487.5 + 5040 \\
\text{multi8 area} &= NRow * NCol * 674400 + NRow * 476 + NRow * \text{ceil}(\lg NRow) * 392 \\
&+ NCol * 367217.5 + NCol * \text{ceil}(\lg NCol) * 487.5 + 13216
\end{aligned}$$

Using the area of the single context FPGA, we find that the corresponding area for the logic and the routing is $NRow * NCol * 218448$. Therefore, we add this value to the equations calculated above in order to find the total chip area for the different FPGA structures. Limiting the number of rows ($NRow$) and the number of 32-bit columns ($NCol$) to a power of 2, we summarize the areas of the different programming styles in Table 1. Figure 9 illustrates on a logarithmic scale how the area of each programming architecture increases exponentially with the number of programming bits.

Programming Structure	Area Equation (l²)
Serial	$NRow * NCol * 291264$
Partial	$NRow * NCol * 260336 + NRow * 476 + NRow \lg NRow * 392 + NCol * 367217.5 + NCol \lg NCol * 487.5$
Multi (2)	$NRow * NCol * 508848 + NRow * 476 + NRow \lg NRow * 392 + NCol * 473297.5 + NCol \lg NCol * 487.5 + 2296$
Multi (4)	$NRow * NCol * 636848 + NRow * 476 + NRow \lg NRow * 392 + NCol * 385937.5 + NCol \lg NCol * 487.5 + 5040$
Multi (8)	$NRow * NCol * 892848 + NRow * 476 + NRow \lg NRow * 392 + NCol * 367217.5 + NCol \lg NCol * 487.5 + 13216$

Table 1: The equations to determine the area of an entire FPGA given the number of rows $NRow$ and the number of 32-bit columns $NCol$ of configuration bits.

Programming Structure	Area For 2²⁰ Programming Bits
Serial	$9.544 \times 10^9 \text{ ̳}$
Partial	$8.547 \times 10^9 \text{ ̳}$
Multi (2)	$16.694 \times 10^9 \text{ ̳}$
Multi (4)	$20.885 \times 10^9 \text{ ̳}$
Multi (8)	$29.273 \times 10^9 \text{ ̳}$

Table 2: The areas required by the different programming architectures given a square layout of 1024 x 1024 programming bits (1024 rows, 32 32-bit columns).

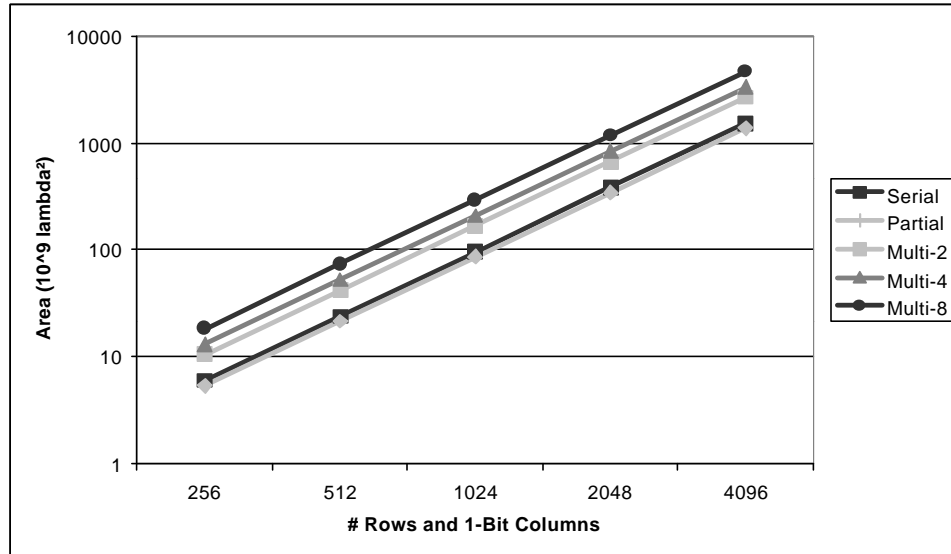


Figure 9: A graph of the area requirements for the different FPGA programming architectures. The programming bits here are organized in a square formation, with the number of rows equaling the number of 1-bit columns. Note that this is on a logarithmic scale.

Model Performance

Using the full chip area requirements for the different programming architectures, the area occupied by each can be set equal, the number of available programming bits can be calculated, and performance can be measured for a fixed area resource. In order to measure performance, real-time caching algorithms were designed to control when configurations should be swapped in and out of the array for each of the FPGA programming architectures [Li00]. To demonstrate their validity, these algorithms were compared to lower bound algorithms that were also developed. The configuration overheads incurred by the different FPGA types given a fixed area resource were calculated by simulating 8 different benchmark programs, and calculating the mean overhead encountered. These benchmarks include some compiled for the Garp system [Hauser97], and some created for the Chimaera System [Hauck97]. The base area for this comparison was computed to be slightly larger than the largest configuration in order to ensure that each configuration would fit within the array. Other sizes, such as 1.5 times this area and 2.0 times this area, were also considered.

A normalized graph of configuration overheads vs. area is shown in Figure 10. The results using both the lower bound and real-time algorithms are given. The real-time algorithms developed for this comparison are an upper bound guideline on what performance is possible for the given architecture. The best performance achievable in real-time therefore resides somewhere between this upper bound and the lower bound.

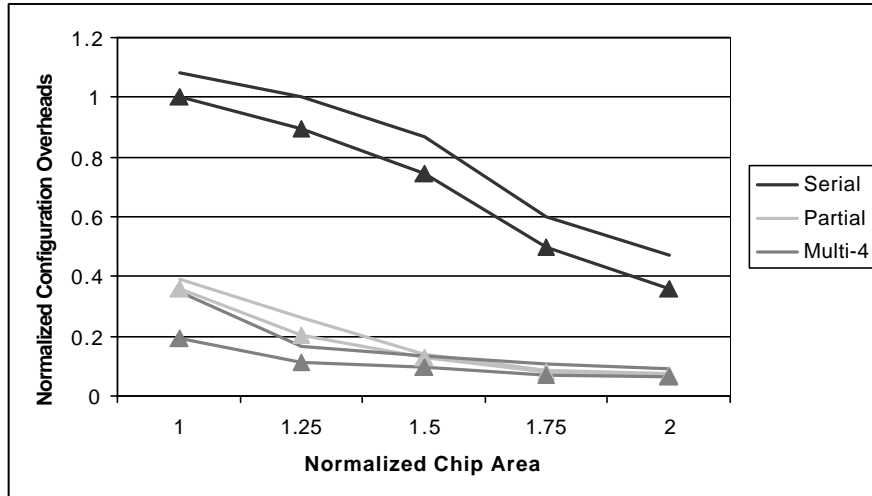


Figure 10: A graph of normalized configuration overheads given a fixed area resource for the different FPGA programming architectures: single context, partially reconfigurable, and multi-context with 4 contexts. The plain lines represent the upper bound algorithms, while the lines with the triangles represent the lower bound algorithms.

Both the partially reconfigurable and the multi-context designs perform far better for a given area than the single context. For the median case, where the area of the chip is approximately 1.5 times the area of the largest configuration, using partial reconfiguration reduces the overhead by a factor of 6.3 over the serial version. The multi-context design reduces the overhead by a factor of 6.6. Going to one size larger, the partially reconfigurable shows an improvement of a factor of as much as 7.1 over the serial single-context design, whereas the improvement shown by the multi-context design decreases to a factor of 5.8.

Because the single context FPGA is the most widespread in commercial use, this indicates that more development is needed in FPGAs before they can effectively be used for run-time reconfigurable applications. The multi-context device (with four contexts for this comparison) performs slightly better than the partially reconfigurable device for smaller areas, as it can switch quickly between contexts. At larger areas, the configuration bounds on overhead encountered in partially reconfigurable designs fall entirely within the bounds of the overheads seen with the multi-context device. However, the configuration overhead of the partially reconfigurable FPGA can be further lowered by adding two further hardware optimizations: relocation and defragmentation.

Optimization: Relocation / Defragmentation

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 have been placed during compilation onto 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 ping-pong 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 [Compton00, Li00]. 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 [Hauck97], Garp [Hauser97], and PipeRench [Cadambi98].

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 could 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 in addition to relocation 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 [Diessel97, Compton00]. The collected unused area can then be used by incoming configurations, potentially without overwriting any of the moved configurations.

Example of 2D Relocation

Figure 11 illustrates a situation in which relocation could be used. The darkly shaded mapping is already present on the FPGA. The lightly shaded mapping is a new mapping that is also to be placed on the FPGA. However, since the first and second configurations have several cell locations in common, they cannot both be present on a traditional partially reconfigurable FPGA simultaneously.

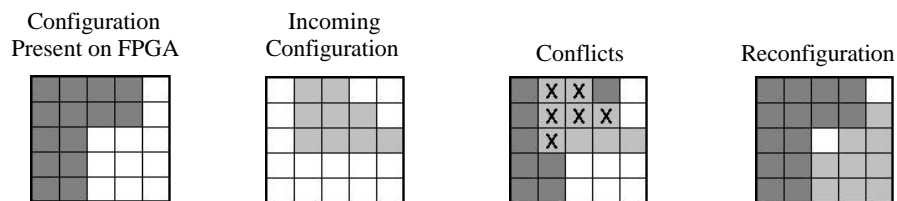


Figure 11: In some situations an incoming configuration maps to the same location as an existing configuration. If the incoming mapping is relocated, it may be possible to allow both configurations to be present and operational on the FPGA simultaneously.

However, an FPGA with relocation ability can modify the second configuration to fit the unused space on the grid, thereby allowing both mappings to be present without one overwriting the other's information. Figure 11 shows the steps taken to relocate the second configuration to available cells.

Modifying the 6200

We have chosen the Xilinx 6200 FPGA [Xilinx96] to adapt for use with configuration relocation because it is a commercial partially reconfigurable FPGA. In addition, the cell layout and local routing are primarily regular. Each cell has the same abilities, regardless of location. These cells are arranged in an island-style layout. The local routing is in the form of nearest-neighbor connections. Longer distance routing is provided in a hierarchical format, which is where we lose heterogeneity. A 4x4 group of logic elements (cells) forms a cluster in which length 4 wires span four logic elements (cells). Signals may only be transferred onto these lines at the border of the 4x4 block. The next level of the routing hierarchy includes a 4x4 group of the smaller 4x4 blocks. These groups have length 16 wires that span the block. Again, these lines may only be written at the border of the group of 4x4 blocks. Additionally, cells are only able to access nearest neighbor and length 4 wires, so the signals must also be transferred to more local routing for reading. This hierarchy continues up until a chip-sized block is formed that includes chip-length wires.

In order to create the reconfiguration hardware, it is convenient to consider a somewhat idealized FPGA similar to the 6200. Like the 6200, this idealized FPGA allows random access to any cell in its array. However, we will assume that its long-distance routing is flexible and can be configured to and from any cell. The irregularity of the 6200 hierarchical routing is therefore removed. We first determine the basic needs of relocation hardware by examining this abstract model. A more in-depth technical discussion of the application of relocation to the Xilinx 6200 is presented elsewhere [Compton00].

Applying 2D Relocation to the 6200

Each configuration has eight distinct permutations of its structure. This does not include simple offset operations to shift the entire configuration to a new location without altering its orientation. An example configuration and its seven permutations are shown in Figure 12. These seven manipulations can be decomposed into combinations of three distinct basic movements: a vertical flip, a horizontal flip, and a rotation of 90°. As shown in Figure 12, with combinations of these movements any basic manipulation can be achieved.

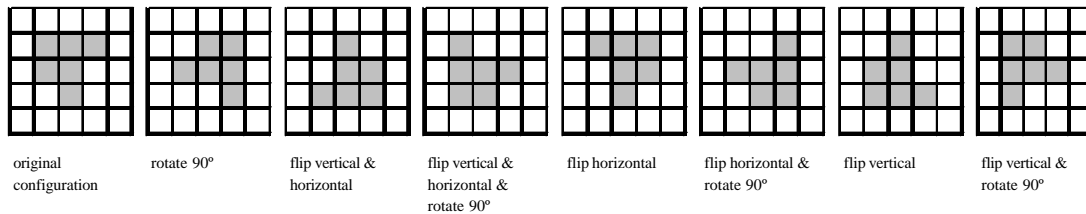


Figure 12: The seven primary permutations of a configuration.

When relocating a mapping, there are a few requirements that we need to meet in order for its functionality to be preserved. First, the routing programmed into each cell must be changed to reflect the overall rotation or flip of the configuration. Each cell in a mapping can have routing to and from its four immediate neighbor cells that must be maintained relative to those neighbors when the mapping is moved. For example, if a cell routes to its neighbor to the east and a horizontal flip is performed, the original cell must now route to that same neighbor which is now found to its west. Alternately, a cell that routes to a cell to the north and belongs to a configuration that is then rotated 90° clockwise would be changed to route to the east.

Second, a cell must also be shifted by the same horizontal and vertical offsets as the entire configuration being relocated. If a mapping is to be moved one column to the east and two rows to the north, each individual cell must be relocated one column to the east and two columns to the north. Additionally, each cell must maintain its position relative to the others so that all routes between cells are preserved. In the rotation example given previously, the northern neighbor must be moved so as to become the eastern neighbor to preserve the correct routing structure.

Third, the relative routing between cells within a configuration must remain intact. The reconfiguration hardware can operate on a cell-by-cell basis to ensure this, changing input and output directions based on the manipulation or manipulations being performed. This can be done using either combinational logic or by looking up the final result in pre-computed tables. Performing translation (shift) operations also involves very little computation. The row and column offsets are simply added to the original row and column addresses of each individual cell. No other manipulations are required for this operation on our idealized 6200 FPGA.

Finally, the relative position of a cell within a configuration must be maintained. While this is easy in a shift operation where the offset is simply applied to all cells within the configuration, it is more complex for the rotate and flip operations. These complex manipulations are easiest to conceptualize as operations performed on one large object. In actuality, however, this one large object is made up of many smaller objects. Each of these must be altered to a different degree in order to preserve the original larger object after the manipulation is complete. In our case, the large object is the full configuration, and the smaller objects are the discrete FPGA cells that form that configuration. Although all of the cells may be flipped or rotated to the same degree as the configuration itself, they each have their own particular offsets to move in order to preserve the relative arrangement between cells within the configuration.

However, if we temporarily consider a configuration to occupy the entire array, these operations are simplified into short equations on a per-cell basis using the original row and column addresses and the maximum row and column addresses. For example, consider a configuration that is to be flipped horizontally. Cells that are in column c will be relocated to column $maxcol - c$. Changing the column address in this manner ensures that each cell is the same distance from the west border as it used to be from the east border, and vice versa. The flip is then followed by a shift of the entire configuration to place it in the desired final location.

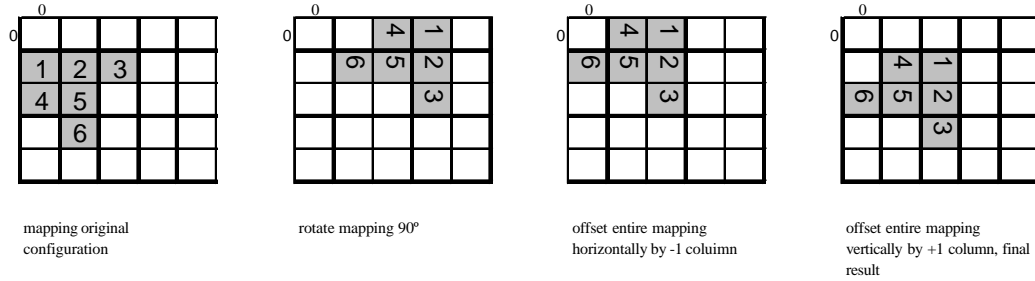


Figure 13: An example relocation using a 90° rotation and an offset.

We show an example of a rotation and an offset operation in Figure 13 that further demonstrates this idea. The cells in the figure are numbered in order to illustrate the location changes for the cells during the relocation of the configuration. In order for a mapping to be successfully manipulated, the relative positions and routing (as represented here by the numbers) should match the original arrangement. The first pane shows an initial mapping.

First the entire array is rotated. In this step, if cell "1" originally routed to cell "2" to the east, it must now be changed to route to cell "2" in the south and its position changes from $\langle 0,1 \rangle$ to $\langle 3,0 \rangle$. If r is the original row position for any cell and c is the original column position, then rotating the mapping changes each cell $\langle c, r \rangle$ to $\langle \text{maxcol}-r, c \rangle$. The next pane shows the entire mapping moved one column to the west. In this case, the position of each cell changes from $\langle c, r \rangle$ to $\langle c+m, r \rangle$ where m is the column translation offset. Finally, the mapping is moved south one row. Here, $\langle c, r \rangle$ becomes $\langle c, r+n \rangle$ where n is the row translation offset. For this example, $m = -1$ and $n = 1$. With a series of simple calculations, a configuration has been repositioned.

Type	Old Location	New Location
Vertical Flip	$\langle c, r \rangle$	$\langle c, \text{maxrow}-r \rangle$
Horizontal Flip	$\langle c, r \rangle$	$\langle \text{maxcol}-c, r \rangle$
Rotate 90°	$\langle c, r \rangle$	$\langle \text{maxcol}-r, c \rangle$
Vertical Offset (by n)	$\langle c, r \rangle$	$\langle c, r+n \rangle$
Horizontal Offset (by m)	$\langle c, r \rangle$	$\langle c+m, r \rangle$

Table 3: The equations to determine the relocated coordinates of a cell.

Table 3 details the equations to compute the new location of a cell for each of the five basic manipulations. These calculations affect the location of each cell, and therefore change the address to which the configuration information is programmed for that cell. These changes do not affect the actual configuration data. However, any manipulations that are needed in order to maintain the relative routing between cells does in fact affect the configuration data.

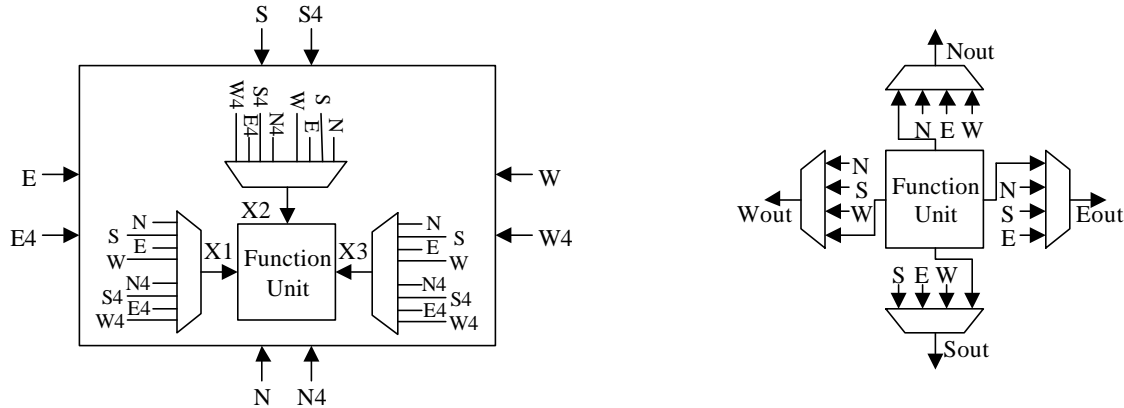


Figure 14: The Xilinx 6200 input structure (left) and output structure (right) [Xilinx96].

Each cell's routing and functionality are controlled by multiplexers, that are in turn selected with SRAM configuration bits local to each cell. There are three inputs to the function unit within the cell, and these three inputs come from the three multiplexers X1, X2, and X3 respectively (Figure 14 left). The output of these multiplexers can be selected from eight locations. N, S, E, and W are the neighboring cells' outputs to the north, south, east and west, respectively. N4, S4, E4 and W4 are the special long distance routes built into the 6200 and are located in the indicated directions. Outputs of each cell follow similarly and are shown in Figure 14 right.

Cell outputs are chosen from the output of the function unit or from the outputs of other cells (effectively routing through a cell). Two bits of SRAM data for each multiplexer are needed to select from these four possible outputs. Figure 15 shows the configuration information for the cell routing. Although these bytes contain the bits labeled CS, RP, Y2, and Y3 which control the function unit of the cell, we are interested in examining only the bits which control the input and output multiplexers. In order to change a cell's configuration the incoming data destined for these three bytes of SRAM must be altered.

Column Offset <1:0>	DATA BIT							
	7	6	5	4	3	2	1	0
00	North		East		West		South	
01	CS	X1[2:0]			X2[1:0]		X3[1:0]	
10		RP	Y2[1:0]		Y3[1:0]		X3[2]	X2[2]

Figure 15: The three data bytes that control the input and output multiplexers a Xilinx 6200 cell [Xilinx96].

Each mapping manipulation (the rotate 90 degrees and the horizontal and vertical flips) has a distinct set of operations on the routing information that must be made on a cellular level. For instance, to flip a mapping vertically, if a northern input was selected by any of the multiplexers of some cell, it now must be changed to be a southern input. We similarly change the output routing – north becomes south, south becomes north, and so forth. For a vertical flip, east/west routing changes do not occur.

The configuration manipulation operations use the configuration bits given in the format shown in Figure 15, and output new values based on the new orientation of the cell. This can be performed using a different set of logic equations for each bit of each multiplexer [Compton00].

With the ability to do the three complex movements and the two offset operations, any repositioning of a cell mapping is possible in our idealized FPGA. A relocation pipeline of five stages can operate on each cell of an incoming mapping and output a fully relocated mapping. Figure 16 shows this pipeline and its operation on the example of Figure 11.

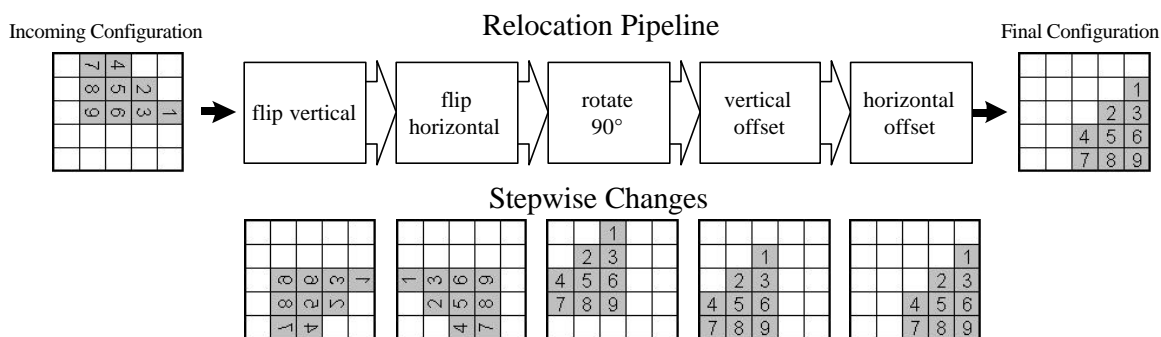


Figure 16: The relocation pipeline and its operation on the example of Figure 11.

Difficulties of the 6200

Using the relocation hardware already discussed, we are potentially able to implement another feature for improved FPGA configuration: defragmentation. The idea of defragmentation is to shift configurations already present on the FPGA in order to consolidate unused area. The unused area can then be used to program additional configurations onto the chip that may not have fit in the previous available space. This is a similar concept to memory defragmentation, although here it is extended to two dimensions.

We can use the hardware and movements that we have described to take configurations that are already loaded onto the cell array and move them elsewhere on the array. If we use the same Relocation Pipeline that we have designed, this operation consists of reading data from the array, running it through the pipeline and writing it back to another location. This is not the quickest way to achieve defragmentation because it involves both a full configuration read and a full configuration write. Alternatively, we could sacrifice some of the flexibility provided by the relocation hardware and employ a defragmentation scheme that simply shifts data directly from cell to cell so that a mapping would be moved horizontally or vertically in single column or row increments. However, this would add a significant amount of routing to a 6200-like FPGA, given that connections would have to be added to relay programming bits from each cell to each of its neighbors. Neither of these two solutions is ideal: one could cause heavy delays due to configuration reads and writes, while the other creates a high area overhead.

Additionally, defragmenting a 2-D array is a complex operation. Essentially, the FPGA must go through a floorplanning stage each time it is defragmented, which is a time-consuming process usually performed in compilation. Although some work has been done on using heuristics to accelerate this operation [Bazargan00], they result in wasted space. Because our aim is to reclaim unused area, this is contrary to our goal. This amount of computation can therefore easily exceed the benefits gained through defragmentation, and cause defragmentation in the 6200 to become unfeasible. A similar difficulty occurs in relocation. If we required that all configurations occupy a rectangular area, we could find free locations without a great deal of difficulty by keeping a list of free rectangles sorted by size. However, odd-shaped configurations would make the search for available space an examination of the FPGA contents on a cell-by-cell basis, which would need to be performed each time a configuration required relocation.

Another consideration is that of I/O. At compile time, the placement and routing tools connect logic blocks to pins for input and output. The pin locations must remain fixed despite relocation because of the board-level connections to the FPGA. Therefore, each time a configuration is moved, the connections between it and the I/O pins it uses need to be re-routed. As routing is an expensive step in the compilation process, it is unlikely that this could be effectively done at run-time. Alternately, we could use the concept of virtualized I/O, which is a bus-based input/output structure that provides a location-independent communication method (this concept is studied in more depth later). However, for two-dimensional virtualized I/O, we would need to provide a method for a configuration to communicate with every pin in the FPGA, which is not practical given the large number of both pins and logic blocks.

A further limitation placed on relocation by the actual 6200 design is that in reality we are not able to make arbitrary movements of mappings. Although the 4-cell spanning routing (N4, E4, etc.) does add some distance routing capability to the 6200 array, it can only be written to near the borders of a 4x4 grouping of cells. This severely limits where we can and cannot move mappings. If a mapping contains 4x4 routing, we are limited to horizontal and vertical movements in multiples of four to preserve this routing. A similar phenomenon occurs at the border of a 16x16 grouping of cells, and so on up a final grouping that is the size of the entire chip.

Although we can create relocation hardware for the simplified 6200 design, introducing the realities of the actual 6200 complicates this hardware significantly. Despite initial appearances, the partially reconfigurable 6200 is not well suited for relocation and defragmentation. While partial reconfigurability is essential to the concept of relocation and defragmentation, there are a number of other notions that are necessary as well. The next sections describe these ideas and how they were used in the design of a new architecture created specifically to feasibly support run-time relocation and defragmentation.

New Relocation / Defragmentation Architecture

We propose a new architecture designed specifically to exploit the benefits of relocation and defragmentation in order to avoid the difficulties presented by the use of the 6200 for this purpose. We will refer to this architecture as the R/D (Relocation / Defragmentation) FPGA. First we examine the guidelines used for the design creation, and then we discuss the details of the actual architecture. Next we show a few examples of the operation of this new FPGA. We also examine a few possible extensions to the R/D architecture. Finally, we give performance results comparing the configuration overhead incurred by our new architecture to that encountered using the serial, partially reconfigurable, and multi-context FPGAs for a given area.

Design Issues

Using a few simple concepts in the design phase of the FPGA, we can ensure that the architecture is suitable for relocation and defragmentation. The first is that of partial reconfiguration. The ability to selectively program portions of the FPGA is critical to the philosophy of relocation and defragmentation, since its addressability provides a way to specify the location of the configuration at run-time. We therefore base the R/D FPGA on a generic partially reconfigurable core, as shown in Figure 17 left.

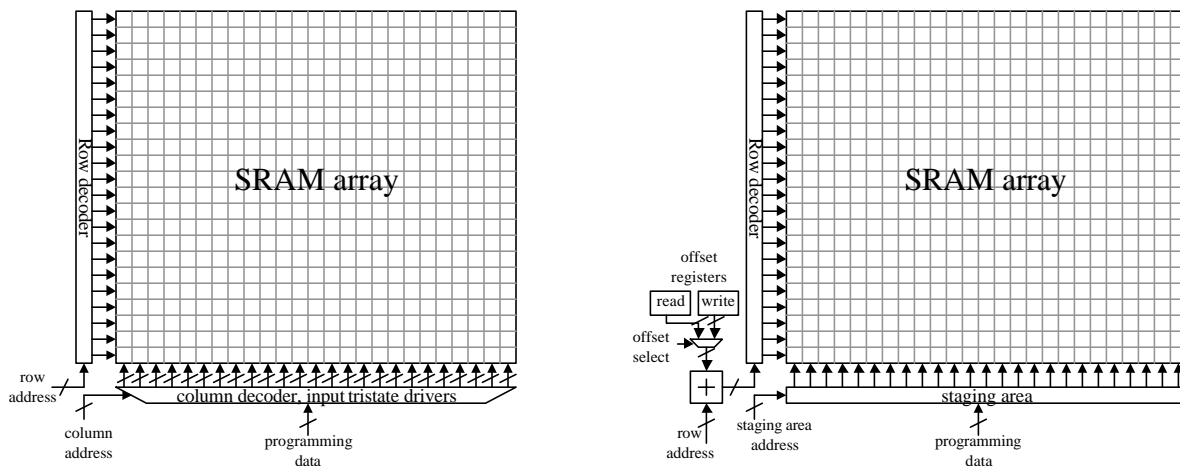


Figure 17: A basic partially reconfigurable FPGA architecture (left), and the Relocation / Defragmentation (R/D) FPGA architecture (right).

The second idea is homogeneity. If each cell in the structure is identical, there are no functional obstacles to moving a configuration from one location to any other location within the boundaries of the array. In the same manner, requiring the routing structure to be homogenous removes any placement limitations for routing reasons. This removes the difficulty that the hierarchical routing structure presents in the 6200. Although the exact structure of

the logic cell and the routing for the R/D FPGA has been left open, we do make homogeneity a requirement. Many current commercial FPGAs are homogeneous, including the Xilinx 4000 [Xilinx94].

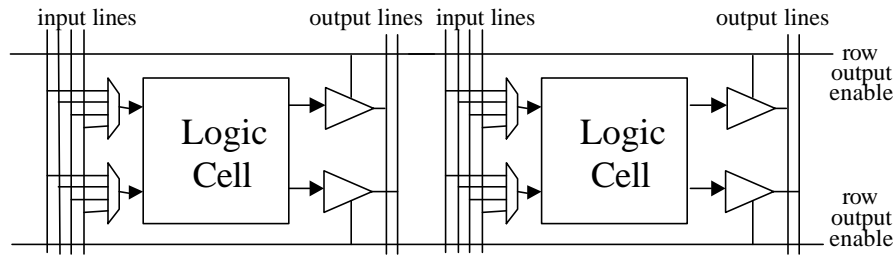


Figure 18: A virtualized I/O structure with four input lines and two output lines. Two cells in one row are shown here. The input and output lines are shared between rows. Although multiple rows may read an input line, only one row at a time may write to any given output line.

The third concept is virtualized I/O. Using a bus-based input/output structure provides us with a location-independent method to read in and write out data from the individual configurations. Configurations are therefore not limited by I/O constraints to be placed near the FPGA pins, plus the I/O routing remains unchanged when the configuration is mapped to a new location. Several architectures already support this, including Chimaera [Hauck97], PipeRench [Hauser97], and GARP [Goldstien99]. Alternately, virtualized I/O can be supported without the use of custom hardware provided that all mappings include bus structures such that adjacent mappings have connected busses.

One type of virtualized I/O system for a row-based FPGA is shown in Figure 18. Row-based FPGAs are those in which a row of FPGA cells forms the atomic configuration unit, and therefore is not shared between configurations. This type of FPGA is discussed in more depth in a few paragraphs. The virtualized I/O structure shown includes four global input values per column, and two global output values per column. A cell can select its inputs from the global input lines using a multiplexer. The actual input value read therefore only depends on the setting of the multiplexer. In this structure, cells can only output to a global output line when the corresponding output enable line is set to high for that cell's row. These enable lines are global, and a control structure is required to ensure that only one row at a time may output to any given line.

In the Chimaera system for example, there are Content-Addressable-Memories located next to each row of cells. When the CPU wishes to read the output of a configuration, it sends the configuration number to the array, which checks this value against the CAM values. If a row's CAM is equal to the configuration number sent by the CPU, the output is enabled for that row [Hauck97].

The fourth important idea is that of one-dimensionality. Current commercial FPGA architectures are based on a two-dimensional structure. Movement of configurations in two dimensions for relocation and defragmentation can be quite difficult, as there are many different placement possibilities to consider. These complexities can be removed when the FPGA is designed with a row-based structure similar to Chimaera [Hauck97] and PipeRench [Goldstein99]. These architectures consider a single row of FPGA cells to be an atomic unit when creating a configuration, where each row forms a stage of the computation. The number of cells in a row is arbitrary, but in general assumed to be the same width as the number of bits in a data word in the host processor. This, in essence, reduces the configurations to one-dimensional objects, where the only allowable variation in configuration area is in the number of rows used. Rotation, horizontal or vertical flipping, or horizontal offset operations are no longer necessary. The only operation required for relocating a configuration is to change the vertical offset. Because of the one-dimensionality, the virtualized I/O is also simplified. Instead of including input and output wires along each column and each row of the FPGA, these lines are only necessary for each column, as described earlier in the example corresponding to Figure 18.

Although one-dimensionality does reduce some of the placement generality found in the two-dimensional FPGA architectures, it does provide some benefits for certain applications beyond simplifying relocation and defragmentation. Providing full 2D 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. Using a one-

dimensional routing structure, 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 the majority of it is along a single dimension as well. This type of structure is best suited for datapath operations that operate on data words. These types of calculations are least affected by the limitations of one-dimensionality, while still gaining the advantages of fast placement and routing and simpler I/O and routing structures. This is essential to relocation and defragmentation, which must perform the configuration movements during runtime.

A number of different reconfigurable systems have been designed as one-dimensional architectures. Both Garp [Hauser97] and Chimaera [Hauck97] are structures which 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 [Rupp98] is similar, with a full column of cells acting as the atomic unit for a configuration, as is PipeRench [Cadambi98]. RaPiD [Ebeling96] is a very coarse-grained one-dimensional reconfigurable architecture that operates only on word-width values instead of single bits. Therefore, buses are routed instead of individual values, which also decreases the time required for routing since the bits of a bus can be considered together rather than as separate routes.

Not only does this one-dimensional structure reduce the hardware requirements for the relocation architecture, it also simplifies the software requirements for determining where a configuration can be relocated to. It is no longer a two-dimensional operation. Also, a defragmentation algorithm that operates in two dimensions with possibly odd-shaped configurations could be quite cumbersome. [Diessel97] discusses one such method for performing 2-D defragmentation. However, when the problem is only one-dimensional, an algorithm based on those for memory defragmentation can be applied.

Architecture of the R/D FPGA

We created the design for the R/D FPGA by using each of the guidelines of the previous section. This section describes the major components of this new FPGA programming model. While this design is similar to the partially reconfigurable FPGA in a number of ways that we will discuss, it has a number of additional architectural features.

Similarly to the partially reconfigurable FPGA, the memory array of the R/D FPGA is composed of an array of SRAM bits. These bits are read/write enabled by the decoded row address for the programming data. However, the column decoder, multiplexer, and input tri-state drivers have been replaced with a structure we term the "staging area", as shown in Figure 17 right.

This staging area is a small SRAM buffer, which is essentially a set of memory cells equal in number to one row of programming bits in the FPGA memory array, where a row of logic cells contains a number of rows of configuration bits. Each row, and therefore the staging area, contains several words of data. The staging area is filled in an addressable fashion one word at a time. Once the information for the row is complete in the staging area, the entire staging area is written in a single operation to the FPGA's programming memory at the row location indicated by the row address. In this manner the staging area acts as a small buffer between the master CPU and the reprogrammable logic. This is similar in function to a structure proposed by Xilinx [Trimberger95], and present in their Virtex FPGA [Xilinx99]. More discussion on the application of relocation and defragmentation to the Virtex FPGA appears in a later section.

In the staging area of the R/D FPGA, there is a small decoder that enables addressable writes/reads. The column decoder determines which of the words in the staging area is being referenced at a given moment. No row decoder is required because we construct the staging area such that although there are several columns, there is only one word-sized row. One output tri-state driver per bit in a word is provided to allow for reading from the staging area to the CPU.

The chip row decoder includes a slight modification, namely the addition of two registers, a 2:1 multiplexer to choose between the two registers, and an adder, where these structures are all equal in width to the row address. This allows a vertical offset to be loaded into one or more of the registers to be added to the incoming row address,

which results in the new relocated row address. One of the two offset registers is the "write" offset register, which holds the relocation offset used when writing a configuration. The other offset register is the "read" register, which is used during defragmentation for reading a relocated configuration off of the array. The original row address supplied to the reconfiguration hardware is simply the row address of that particular row within the configuration. Therefore, all configurations are initially "located" starting at address 0 at the top of the array. For simplicity, this version of the R/D hardware does not allow configurations to be relocated horizontally to different columns.

A basic partially reconfigurable FPGA requires a column decoder to determine which data word within a row should be accessed for reading or writing. However, a column decoder between the staging area and the array is not necessary in the R/D design. The staging area is equal in width to the array, and therefore each bit of the staging area is sent out on exactly one column. This provides for a high degree of parallelism when reading from the FPGA configuration memory to the staging area or writing from the staging area to the FPGA memory, as a full row is read or written in a single operation.

Finally, although we have stated that our FPGA contains a homogeneous cell and routing structure, as well as virtualized I/O, the specifics of these structures are not dictated by the memory structure. The particular design is unrestricted because the actual architectures do not influence the discussion of the philosophy and operation of the configuration aspect of the R/D FPGA.

Example of R/D Operation

Figure 19 illustrates the steps involved in writing a row of configuration data to the FPGA SRAM array. Each step shown uses one read/write clock cycle. The words are loaded into the staging area one at a time. Once the words are loaded into the staging area, they are all written in a single write cycle to the memory array itself. Although the figure shows the words loaded in a particular order into the staging area, this is not necessarily the case. The staging area is word-addressable, allowing it to be filled in an arbitrary order. Furthermore, the example shows four words filling the staging area (the number four was chosen for illustrative purposes only). The staging area can be any size, but is expected to be at least several words wide.

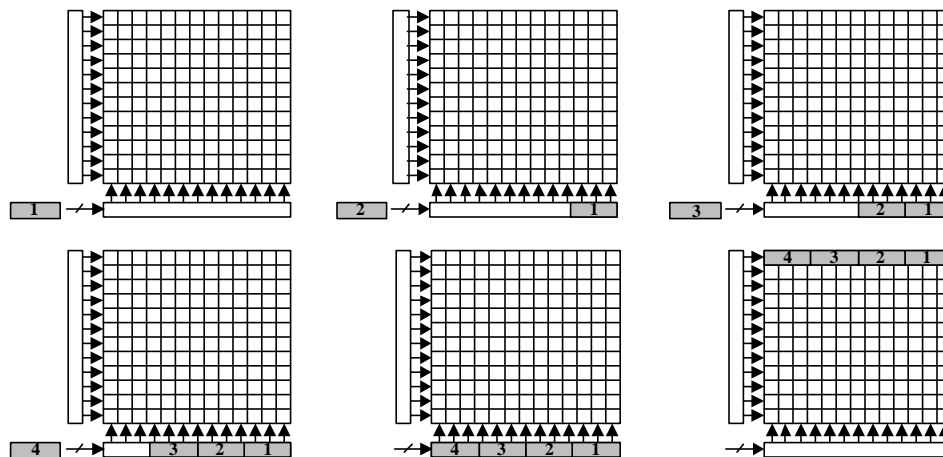


Figure 19: A single row of configuration data is written to the FPGA by performing multiple word-sized writes to the staging area, followed by a single write from the staging area to the array. Each step shows a single write cycle.

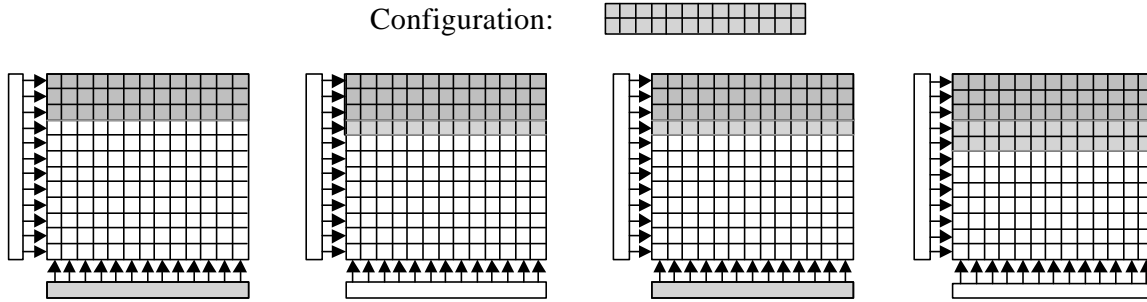


Figure 20: An example of a configuration that is relocated as it is written to the FPGA. Writing the data to the N-word staging area requires N cycles per row, while writing from the staging area to the array requires only one cycle.

Relocation of a configuration is accomplished by altering the row address provided to the row decoder. Using a single row offset and an adder provides a simple way to dynamically relocate individual configurations to fit available free space. Figure 20 shows the steps to relocate a configuration as it is being loaded into the FPGA.

First the offset value required to relocate a configuration is loaded. In this case, a value of "3" is written to the write offset register to force the incoming configuration to be relocated directly beneath the configuration already present in the FPGA. Next, the CPU or the DMA loads each configuration row one data word at a time into the staging area. The entire staging area is then written to the destination row of the FPGA in a single operation. The actual address of this row is determined by adding the write offset register to the destination address for that row. For each row of the configuration there are as many writes to the staging area as there are words in a row, followed by one write from the staging area to the FPGA. This is in addition to the single write to the offset register per configuration in order to relocate a configuration to an empty location. The total number of read/write cycles to write a configuration to the array is therefore:

$$\langle \# \text{ rows} \rangle * (\langle \text{staging area size} \rangle / \langle \text{data word size} \rangle + 1) + 1$$

If we consider a number of full row width configurations that would have been programmed onto a basic partially reconfigurable FPGA, we are only adding $\langle \# \text{ rows} \rangle + 1$ cycles to the configuration time in order to allow relocation, and is only one more write than required for current commercial architectures that already use such a staging area.

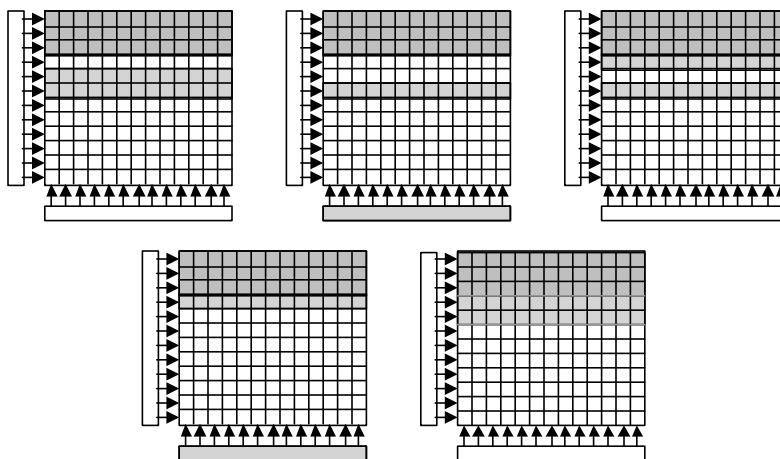


Figure 21: An example of a defragmentation operation. By moving the rows in a top-down fashion for configurations moving upwards in the array, a configuration will not overwrite itself during defragmentation.

Defragmentation of the R/D FPGA is only slightly more complicated than a simple relocation operation. Rows must be moved from existing locations on the FPGA to new locations without overwriting any necessary data. This is particularly apparent when the new location of a configuration partially overlaps the current location. Depending on the order of the row moves, one or more of the rows of information could be lost. In particular, if a configuration is to be moved "up" in the array, the rows should be moved in a topmost-first order. For a configuration that is to be moved "down", the rows should be moved in a bottommost-first order. Figure 21 shows an example of the correct order to move rows in a configuration to prevent loss of data when the configuration is being moved "up" in the array. Because each of these movements are of entire rows of configuration information, defragmentation is performed very quickly.

Here we use both of the offset registers. The read register is used to store the offset of the original location of the configuration. The write register holds the offset of the new configuration location. First, using a row address of 0 and a read offset of 6, the top row of information for the second configuration is read back into the staging area. The row is then written back out to the new location using the same row address, but a write offset of 4. The address sent to the row decoder is incremented (although the contents of the two registers remain unchanged), and the procedure continues with the next row.

Using two registers instead of one allows each row to be moved with a single read and a single write, without having to update the register as to which address to read from or write to. A 1-bit signal controls the 2:1 multiplexer that chooses between the two offsets. There are also two cycles necessary to initialize the two registers. The total number of read/write cycles required to move a configuration is:

$$\langle \# \text{ rows} \rangle * 2 + 2$$

This structure also allows for partial run-time reconfiguration, where most of the structure of a configuration is left as-is, but small parts of it are changed. One example of this type of operation would be a multiply-accumulate with a set of constants that change over time, such as with a time-varying finite impulse response (FIR) filter. A generic example is shown in Figure 22. The changed memory cells are shown in a darker shade.

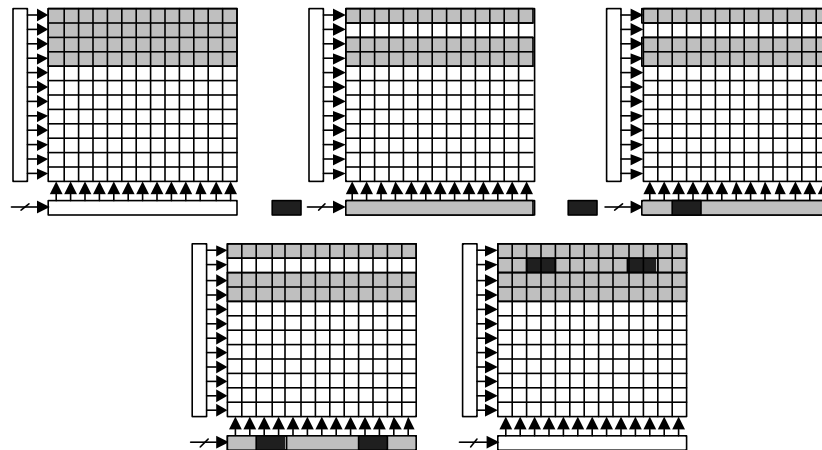


Figure 22: Portions of a configuration can be altered at run-time. This example shows small modifications to a single row of a configuration. Changed configuration information is shown in a darker shade.

First, the row to be partially programmed must be read back into the staging area. Then this row is partially modified (through selectively overwriting the staging area) to include the new configuration. Finally, the modified row is written back to the array, preserving the configuration information already present in the row. This is repeated for each altered row in the configuration.

For each row to be altered in the configuration, there is one read of the original row data, one or more writes to change the data in the staging area, and a single write back to the array from the staging area. This is in addition to a

single write to an offset register for the configuration offset. The total number of read/write cycles required to place a partial-row configuration onto the array is:

$$\langle \# \text{ rows altered} \rangle * 2 + \langle \text{total \# changed words} \rangle + 1$$

Xilinx Virtex for Relocation and Defragmentation

Relocation and defragmentation can also be performed, with some limitations, in one of the current commercial FPGAs. As we have stated previously, the staging area of the R/D FPGA is similar to what is present in Xilinx's Virtex FPGA [Xilinx99]. In this FPGA, this structure is referred to as the Frame Data Input Register, where a frame is a column of configuration information (as opposed to our design, which is organized in rows). The frame register is essentially a shift register that is loaded serially with the configuration information for a frame. This information is then transferred to the FPGA in parallel to a location supplied by the CPU (making the FPGA partially reconfigurable on a frame-by-frame basis). Although the frame register does not contain all of the important features of the R/D FPGA staging area, it can be used in such a way as to provide relocation and defragmentation ability. Instead of performing the relocation of the configuration at the FPGA itself, the CPU would be required to compute the new destination address of each frame, and send this address to the FPGA. Also, because the Virtex architecture does not include virtual I/O hardware, the configurations themselves must include a method to allow input and output values to be placed on wires designated as chip-wide busses for those signals. Each configuration would be required to propagate all of the the busses required in all configurations that could be present on the FPGA at the same time.

However, this method of providing virtualized I/O uses the limited FPGA routing resources that may be required for signals within the actual configuration. Additionally, since the CPU is now responsible for computing the new addresses of each frame, it is performing many bit-wise address calculations that are not required in the R/D model we have presented. Finally, to provide full partial run-time reconfiguration, the frame register should be addressable to allow for the partial run-time reconfiguration shown in the last example in the previous section. Although Xilinx's Virtex FPGA is similar in design to the R/D FPGA, it is lacking a number of features that would provide for easy relocation and defragmentation of configurations. However, the similarity it does share with our design does indicate the feasibility of our proposed programming structure.

Cache for R/D FPGA

An additional method to reduce the CPU time required for configuration operations would be to attach an on-chip cache to the staging area, such as in Figure 23. Rows of configuration information could then be held in the cache. The full details of the actual cache structure are left open. However, the easiest method for uniquely identifying a given row is through the use of a configuration number in conjunction with the position of the row within that configuration.

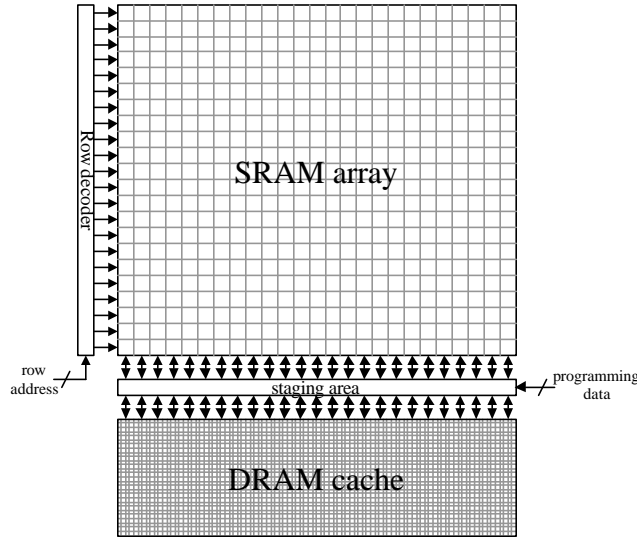


Figure 23: A cache can be attached to the staging area of the R/D FPGA. Entire configuration rows can be fetched from the cache into the staging area, eliminating the per-word loading time required to fill the staging area from the CPU. This cache could be built out of either DRAM or SRAM.

For rows of configuration information that are already present in the cache, the CPU would be freed from the operations necessary to send each word of the row to the staging area. This therefore reduces the latency of retrieving this row from the CPU's memory, and the actual programming of the array would be performed much more quickly. The entire row would be read from the cache in a single operation, rather than the multiple word writes to the staging area from the CPU. Also, the reading of data from the cache could overlap the writing of the previous value. If an entire configuration was held in the cache, the number of read/write cycles required to place it onto the array would only be:

$$\langle \# \text{ rows} \rangle + 2$$

Model Area and Performance

We modeled the sizes of the basic partially reconfigurable FPGA and the R/D FPGA using the same structures presented earlier in this thesis. Again, the sizes are estimated using the sizes of tileable components. We used small modifications to the hardware of a basic partially reconfigurable FPGA design to create our R/D FPGA design.

The column decoder of the partially reconfigurable system was unnecessary in the R/D version because the staging area is exactly the width of the memory array, and was therefore removed for the R/D size model. There were also several additions to the partially reconfigurable FPGA design to create the R/D FPGA. The staging area structure includes the addition of staging area SRAM, output drivers to allow the CPU to read the staging area, and the small decoder for writing to it. Because the row and column decoders serve an identical function but the orientation of the row decoder layout makes it smaller, the row decoder layout is used here instead of the column decoder layout. Additionally, the main row decoder for the array was augmented with two registers, a 2:1 multiplexer for choosing between the registers, and an adder to sum the offset from one of the registers with the incoming row address.

For N_{Row} ($N_{Row} \geq 4$) rows and N_{Col} ($N_{Col} \geq 8$) 32-bit columns of programming bits, the area of the programming structure in λ^2 for the R/D FPGA is:

$$\begin{aligned}
\text{R/D FPGA} &= \langle \# \text{ prog. bits} \rangle * \langle \text{prog. bit area} \rangle + \langle \text{row decoder area} \rangle + \langle \text{I/O tri-state area} \rangle \\
&+ \langle \text{staging area} \rangle + \langle \text{staging row decoder area} \rangle + \langle \text{staging I/O tri-state area} \rangle \\
&+ 2 * \langle \text{row offset register area} \rangle + \langle \text{adder area} \rangle + \langle 2:1 \text{ mux area} \rangle \\
&= NRow * (NCol * 32) * 1309 + NRow * (476 + 392 \text{ ceil}(\lg NRow)) \\
&+ (NCol * 32) * 11407.5 + (NCol * 32) * 1309 + NCol * (476 + 392 * \text{ceil}(\lg NCol)) \\
&+ 32 * 11407.5 + 2 * \text{ceil}(\lg NRow) * 8730 + \text{ceil}(\lg NRow) * 11352 + \text{ceil}(\lg NRow) * 1156 \\
&= NRow * NCol * 41888 + NRow * 476 + NRow * \text{ceil}(\lg NRow) * 392 \\
&+ NCol * 407404 + NCol * \text{ceil}(\lg NCol) * 392 + \text{ceil}(\lg NRow) * 29968 + 365040
\end{aligned}$$

Table 4 lists the full-chip area equation for the R/D FPGA along with the equation for the partially reconfigurable FPGA for comparison.

Programming Structure	Area Equation (1 ²)
Partial	$NRow * NCol * 260336 + NRow * 476 + NRow \lg NRow * 392 + NCol * 367217.5 + NCol \lg NCol * 487.5$
R/D FPGA	$NRow * NCol * 260336 + NRow * 476 + NRow \lg NRow * 392 + NCol * 407404 + NCol \lg NCol * 392 + \lg NRow * 29968 + 365040$

Table 4: The chip area equation for the R/D FPGA. The partially reconfigurable area equation is repeated here for comparison.

We compared the sizes of the base partially reconfigurable FPGA and the R/D FPGA, each modeled with a megabit (2²⁰ bits) of configuration data in a square layout (# rows = # 1 bit columns). There are 1024 rows, addressed using 10 bits. For the columns there are 32 32-bit columns, addressed by five bits. The area of the partially reconfigurable array was calculated to be 8.547 X 10⁹ ě² while the area of the R/D FPGA was calculated to be 8.549 x 10⁹ ě², a difference of .0002%. According to this comparison, the R/D FPGA has only a negligible size increase over a basic partially reconfigurable FPGA. Figure 24 shows the result of applying relocation and defragmentation to the partially reconfigurable FPGA, comparing it to the partially reconfigurable and the multi-context designs. The R/D FPGA halves the configuration overhead shown by the base partially reconfigurable array. This translates to an improvement by a factor of 11 over the serially-programmed single context FPGA.

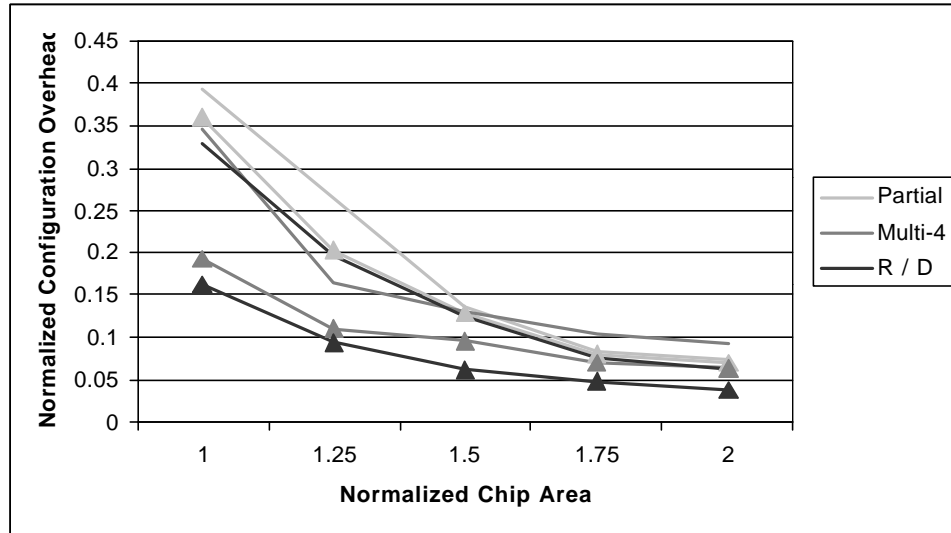


Figure 24: A graph of normalized configuration overheads given a fixed area resource for the different FPGA programming architectures: partially reconfigurable, multi-context with 4 contexts, and the Relocation / Defragmentation (R/D) FPGA. The plain lines indicate upper bounds, while the lines with triangles indicate the lower bounds.

The area of the virtualized I/O was not considered for this area model. The area impact would depend on the number of input and output lines at each column of the array.

Conclusion

Reconfigurable computing is becoming an important part of research in the FPGA and VLSI/CAD communities. By placing the computationally intense portions of an application onto the reconfigurable hardware, that application can be greatly accelerated. Similar to software-only implementations, the mapped circuit is flexible, and can be changed over the lifetime of the system or even the course of execution of the application. Additionally, the computations mapped to the reconfigurable logic are executed in hardware, and therefore have performance similar to an ASIC. This performance stems from bypassing the fetch-decode-execute cycle of traditional microprocessors as well as allowing parallel execution of multiple operations.

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 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.

The use of relocation and defragmentation greatly reduces the configuration overhead encountered in run-time reconfigurable computing. While partial reconfiguration can improve the configuration overhead by more than a factor of 7 over the serially programmed FPGA, the ability to perform relocation and defragmentation of configurations increased this a factor of 11.

We have discussed a method to perform the relocation of configurations on the 6200 that allows horizontal and vertical flips, horizontal and vertical offsets, and 90° rotations. These five operations allow us to perform any valid spatial manipulation of a configuration with a simple pipelined set of steps, minimizing the work required by the CPU. Although a stylized version of the Xilinx 6200 FPGA can be converted to handle relocation and even defragmentation, the re-introduction of some of the realities of the architecture poses significant drawbacks to our modifications. The hierarchical routing structure, for example, places constraints upon our ability to relocate configurations to new locations. The lack of a hardware-based virtual I/O system requires that the connections between the configurations and the I/O pins they use be re-routed for each relocation. The design is also less than

ideally suited to defragmentation. One of our solutions was to read the configuration off of the array and reload it, which could be a time-consuming operation. Alternatively, neighbor-to-neighbor routing for the programming information could be added to allow configurations to be shifted on-chip, but would likely cause large area increases and would prohibit complex operations such as flips or rotations. The time complexity of the calculations involved to compute the new locations is also very high.

We then presented a new architecture design based on the ideas of relocation and defragmentation. This architecture avoids the position constraints imposed by the actual 6200 design by ensuring a homogeneous logic and routing structure. The use of the staging area buffer together with the offset registers and the row address adder provide a quick and simple method for performing relocation and defragmentation of configurations. The one-dimensional nature causes both the reconfiguration hardware and the software that controls it to be simpler than in the 6200 system.

The R/D FPGA exploits the virtues of relocation and defragmentation in order to reduce the overhead of configuration, which is a great concern in run-time reconfigurable applications. The architecture is designed to require little additional run-time effort on the part of the CPU, and requires only a negligible area increase (.0002%) over a basic partially reconfigurable FPGA. Furthermore, because the design shares some key features with a new commercial FPGA, our R/D FPGA design is a feasible next step in the advancement of FPGA programming architectures.

Acknowledgments

The author would like to thank fellow graduate student Zhiyuan Li for the performance comparison of the different FPGA programming architectures, and undergraduates James Cooley and Stephen Knol for their work on modifying the Xilinx 6200 for relocation.

The author would also like to thank her advisor, Scott Hauck, for his assistance in preparing this thesis.

This research was funded in part by a grant from Motorola, Inc. Katherine Compton was supported by an NSF Fellowship.

References

- [Aggarwal94] A. Aggarwal, D. Lewis, "Routing Architectures for Hierarchical Field Programmable Gate Arrays", *Proceedings IEEE International Conference on Computer Design*, pp. 475-478, 1994.
- [Altera98] *Data Book*, Altera Corporation, San Jose, CA: 1998.
- [Annapolis98] *Wildfire Reference Manual*, Annapolis, Maryland: Annapolis Microsystems, Inc., 1998.
- [Bazargan00] K. Bazargan, R. Kastner and M. Sarrafzadeh, "Fast Template Placement for Reconfigurable Computing Systems", to appear in *IEEE Design and Test - Special Issue on Reconfigurable Computing*, January-March 2000.
- [Cadambi98] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, D. E. Thomas, "Managing Pipeline-Reconfigurable FPGAs", *ACM/SIGDA International Symposium on FPGAs*, pp. 55-64, 1998.
- [Compton00] K. Compton, J. Cooley, S. Knol, S. Hauck, "FPGA Hardware Support for Configuration Relocation and Defragmentation", in preparation for *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.
- [Deshpande99] D. Deshpande, A. K. Somani, A. Tyagi, "Configuration Caching Vs Data Caching for Striped FPGAs", *ACM/SIGDA International Symposium on FPGAs*, pp. 206-214, 1999.
- [Diessel97] O. Diessel, H. ElGindy, "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. Berlin, Germany: Springer-Verlag, pp. 131-140, 1997.

- [Ebeling96] 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.
- [Goldstein99] S. C. Goldstein, H. Schmit, M. Moe, M. Budi, S. Cadambi, R. R. Taylor, R. Laufer, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration", *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.
- [Hauck97] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, "The Chimaera Reconfigurable Functional Unit", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.
- [Hauck98a] S. Hauck, Z. Li, E. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.
- [Hauck98b] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors", *ACM/SIGDA International Symposium on FPGAs*, pp. 65-74, 1998.
- [Hauck99] S. Hauck, W. D. Wilson, "Runlength Compression Techniques for FPGA Configurations", *Northwestern University, Dept. of ECE Technical Report*, 1999.
- [Hauser97] J. R. Hauser, J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.
- [Haynes98] S. D. Haynes, P. Y. K. Cheung, "A Reconfigurable Multiplier Array For Video Image Processing Tasks, Suitable for Embedding In An FPGA Structure", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.
- [Lai97] Y.-T. Lai, P.-T. Wang, "Hierarchical Interconnection Structures for Field Programmable Gate Arrays", *IEEE Transactions on VLSI Systems*, Vol. 5, No. 2, pp. 186-196, June 1997.
- [Laufer99] R. Laufer, R. R. Taylor, H. Schmit, "PCI-PipeRench and the SwordAPI: A System for Stream-based Reconfigurable Computing", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.
- [Li99] Z. Li, S. Hauck, "Don't Care Discovery for FPGA Configuration Compression", *ACM/SIGDA International Symposium on FPGAs*, pp. 91-98, 1999.
- [Li00] Z. Li, K. Compton, S. Hauck, "Configuration Caching for FPGAs", in preparation for *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.
- [Lucent98] *FPGA Data Book*, Lucent Technologies, Inc., Allentown, PA: 1998
- [Luk97] W. Luk, N. Shirazi, P. Y. K. Cheung, "Compilation Tools for Run-Time Reconfigurable Designs", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.
- [Marshall99] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, B. Hutchings, "A Reconfigurable Arithmetic Array for Multimedia Applications", *ACM/SIGDA International Symposium on FPGAs*, pp. 135-143, 1999.
- [Miyamori98] T. Miyamori, K. Olukotun, "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.
- [Moritz98] C. A. Moritz, D. Yeung, A. Agarwal, "Exploring Optimal Cost Performance Designs for Raw Microprocessors", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.
- [Quickturn99a] "System Realizer™", <http://www.quickturn.com/products/systemrealizer.htm>, Quickturn, A Cadence Company, San Jose, CA: 1999.

- [Quickturn99b] “Mercury™ Design Verification System Technology Backgrounder”, http://www.quickturn.com/products/mercury_backgrounder.htm, Quickturn, A Cadence Company, San Jose, CA: 1999.
- [Razdan94] R. Razdan, M. D. Smith, “A High-Performance Microarchitecture with Hardware-Programmable Functional Units”, *International Symposium on Microarchitecture*, pp. 172-180, 1994.
- [Rupp98] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, M. Gokhale, "The NAPA Adaptive Processing Architecture", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.
- [Shirazi98] N. Shirazi, W. Luk, P. Y. K. Cheung, "Automating Production of Run-Time Reconfigurable Designs", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.
- [Smith99] W. H. Mangione-Smith, “ATR from UCLA”, *Personal Communications*, 1999.
- [Sutherland99] I. Sutherland, B. Sproul, D. Harris, "Logical Effort: Designing Fast CMOS Circuits", Morgan Kaufmann Publishers, San Francisco, CA: 1999.
- [Trimberger95] S. Trimberger, "Field Programmable Gate Array with Built-In Bitstream Data Expansion", *U.S. Patent 5,426,379*, issued June 20, 1995.
- [Trimberger97] S. Trimberger, D. Carberry, A. Johnson, J. Wong, "A Time-Multiplexed FPGA", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.
- [Trimberger98] S. Trimberger, “Area Requirements of Programming Structures”, *Personal Communications*, 1998.
- [Wirthlin95] M. J. Wirthlin, B. L. Hutchings, “A Dynamic Instruction Set Computer”, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99-107, 1995.
- [Wirthlin96] M. J. Wirthlin, B. L. Hutchings, “Sequencing Run-Time Reconfigured Hardware with Software”, *ACM/SIGDA International Symposium on FPGAs*, pp. 122-128, 1996.
- [Wittig96] R. D. Wittig, P. Chow, “OneChip: An FPGA Processor With Reconfigurable Logic”, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 126-135, 1996.
- [Xilinx94] *The Programmable Logic Data Book*, Xilinx, Inc., San Jose, CA: 1994.
- [Xilinx96] *XC6200: Advance Product Specification*, Xilinx, Inc., San Jose, CA: 1996.
- [Xilinx99] *Virtex™ Configuration Architecture Advanced Users’ Guide*, Xilinx, Inc., San Jose, CA: 1999.