Configuration Relocation and Defragmentation for Reconfigurable Computing

Katherine Compton, James Cooley, Stephen Knol

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

Scott Hauck

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

Abstract

Custom computing systems exhibit significant speedups over traditional microprocessors by mapping compute-intensive sections of a program to reconfigurable logic [Hauck98]. 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 cost of configuration. Research has shown that the ability to relocate and defragment configurations on an FPGA dramatically decreases the overall configuration overhead [Li00]. This increases the viability of mapping portions of the program that were previously considered to be too costly. We therefore explore the adaptation of the Xilinx 6200 series FPGA for relocation and defragmentation. Due to some 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

One application of FPGAs is that of reconfigurable computing – the use of a run-time reprogrammable device operating as a customizable coprocessor or functional unit alongside a main microprocessor [Hauck98]. This reconfigurable logic is used to emulate custom hardware for the acceleration of one or more compute-intensive portions of a program. Unfortunately, the speedups attainable by the use of reconfigurable logic are limited by the large configuration overheads incurred each time a function is loaded into the array [Li00].

Li compared the total configuration overheads exhibited during program execution by different FPGA types acting as the reconfigurable coprocessor. Two of these types are the serial and the partially reconfigurable FPGAs. The traditional serial FPGA loads configuration information for the entire chip in a bit or byte serial fashion, while the partially reconfigurable FPGA, such as the Xilinx 6200, can be selectively programmed during runtime in an addressable manner. The partially reconfigurable FPGA was determined to have the potential to improve configuration overhead by more than a factor of 7 over the serial FPGA.

Variations on the partially reconfigurable FPGA were also examined. Relocation, the ability to determine at runtime the actual location in the array of a pre-compiled configuration, improved the configuration overhead of the generic partially reconfigurable FPGA by a factor of up to 1.5, and the serial FPGA by nearly a factor of 8. Furthermore, the ability to defragment the configurations already present on the array to consolidate unused computation area decreased the overhead by a factor of 1.5 over the partially reconfigurable FPGA with relocation [Li00]. This leads to more than an overall factor of 11 improvement in configuration overhead over the serially programmed FPGA.

These increases in efficiency can affect the number of program areas suitable for FPGA acceleration. The configuration overhead of a serial or even a basic partially reconfigurable FPGA might outweigh the speedups obtained through the use of the reconfigurable logic for a particular portion of the program. In this case, this section should not be mapped to the reconfigurable coprocessor. However, with the lower configuration cost of the relocation and defragmentation FPGAs, the guidelines for approving a function for acceleration in hardware are relaxed, increasing the potential for sections of a program to qualify for translation to the reconfigurable coprocessor, and increasing the overall speedup attainable.

In order to leverage the advantages of relocation, we examine the refitting of the Xilinx 6200 into a relocation-enabled FPGA. Later we discuss the issues
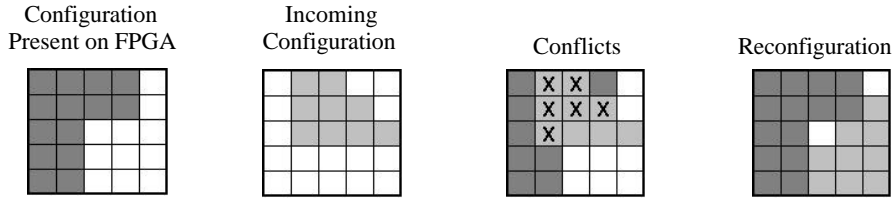
Figure 1: In some situations an incoming configuration maps to the same location as an existing configuration. If the incoming mapping is relocated, it may be possible to allow both configurations to be present in the FPGA concurrently.

in applying the idea of defragmentation to the updated relocation 6200. Finally, we will propose a novel architecture designed specifically from the ground up for partial configuration, relocation and defragmentation.

Example of Relocation

Although the partially reconfigurable FPGA design is powerful, it faces limitations imposed by configuration locations determined at compile time. If two different configurations were mapped at compile time to overlapping locations in the FPGA, only one of these configurations can be present in the array at any given moment. They cannot operate simultaneously. However, if somehow the final FPGA location could be determined at runtime, one or both of these overlapping configurations could be shifted to a new location that was previously unused to allow for simultaneous use.

Figure 1 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.

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 1 shows the steps taken to relocate the second configuration to available cells.

Xilinx 6200 for Relocation

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 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 homogeneity. 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.

As we will demonstrate later in this paper, the relocation of a configuration requires modifications to the programming data and/or programming addresses on a cell-by-cell basis. Although the main CPU could perform these manipulations, it would require effort proportional to the size of the configuration.

Alternatively, we can create the logic necessary to implement the manipulations in hardware placed in or near the FPGA chip itself. Instead of changing the bitstream before it is output to the FPGA, the CPU could also send relocation instructions with the configuration bitstream to the reconfigurable logic. This message would contain a high-level description of which alterations should be made to the entire configuration. The relocation logic would then calculate the actual changes to the bitstream as the configuration information enters the FPGA. The relocation hardware will be able to move, flip, and rotate multi-cell mappings to make the most efficient use of the cell array. This minimizes the effort required on the part of the CPU to efficiently use the reconfigurable logic, leaving it available for other computing tasks.

In order to create such reconfiguration hardware, it is convenient to consider a somewhat idealized FPGA similar to the 6200 [Xilinx96]. Like the 6200, this idealized FPGA allows random access to any cell in its array. However, we will assume that its long-distance
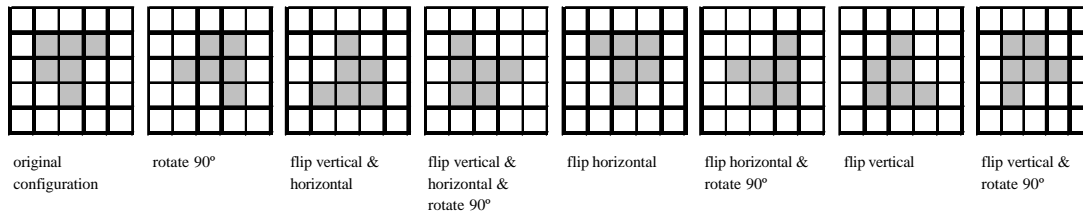
Figure 2: The eight permutations of a configuration.

routing is flexible and can be configured to and from any cell. This removes the irregularity of the 6200 hierarchical routing. We will first determine the basic needs of relocation hardware by examining this abstract model. Later, we will use this model to discuss an actual reconfiguration hardware design for the 6200.

Abstract Relocation

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 2. These seven manipulations can be decomposed into combinations of three distinct basic movements: a vertical flip, a horizontal flip, and a rotation of 90 degrees. With combinations of these movements, any basic manipulation shown in Figure 2 can be achieved.

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 degrees 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. 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, changing input and output directions based on the manipulation or manipulations being performed. This can be performed using either combinational logic or lookup 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
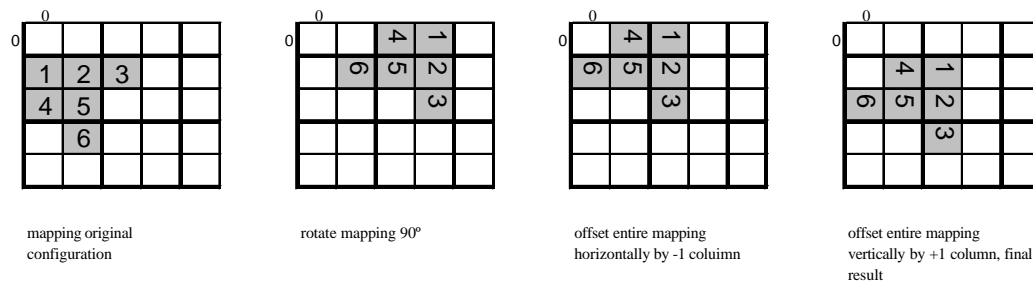

Figure 3: An example relocation using a 90 degree rotation and an offset.

Type	Old Location	New Location
Vertical Flip	$\langle c, r\rangle$	$\langle c, maxrow\text{-}r\rangle$
Horizontal Flip	$\langle c, r\rangle$	$\langle maxcol\text{-}c, r\rangle$
Rotate 90°	$\langle c, r\rangle$	$\langle maxcol\text{-}r, c\rangle$
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 1: The equations to determine the relocated coordinates for a cell.

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 matter 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.

We show an example of a rotation and an offset operation in Figure 3 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 $\langle maxcol\text{–}r, c\rangle$. 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 fully relocated.

With the ability to do the three complex movements and the two offset operations, any reconfiguration of a cell mapping is possible in our idealized FPGA. Table 1 details the position equations for these five manipulations. Any reconfiguration hardware that we design will take an incoming mapping, pass each cell of it through a pipeline of these five stages, and output a fully reconfigured mapping. Figure 4 shows this pipeline and its operation on the example of Figure 1.

Relocation on the 6200

The purpose thus far has been to propose an abstract way of relocating cell-based FPGA mappings. We are in essence designing hardware that takes as input the information for a cell (its configuration and location bits) and changes it according to some master direction from the CPU. Given the desired changes and the configuration data of each cell, our reconfiguration hardware should be able to achieve any relocation in the idealized model of our FPGA. We will now discuss how this can be implemented on the 6200. In
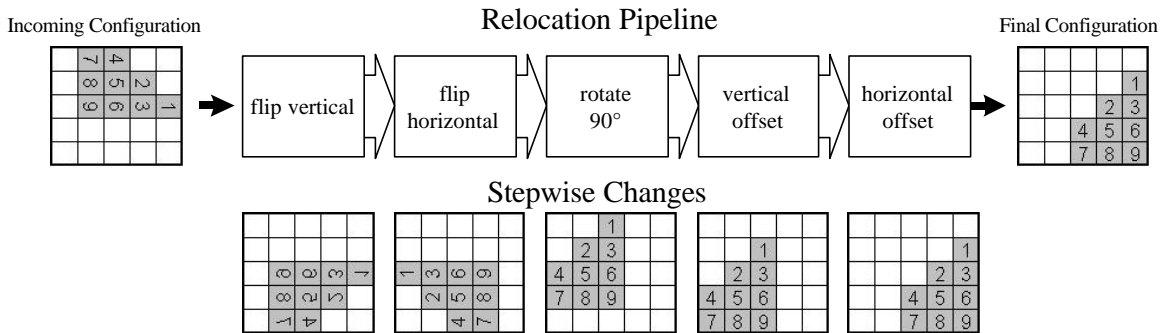

Figure 4: The relocation pipeline and its operation on the example of Figure 1.
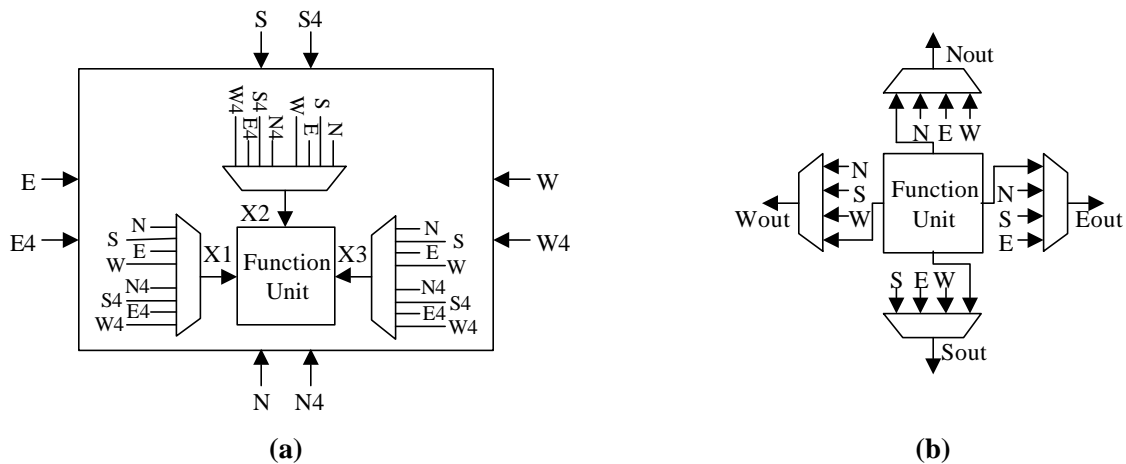
Figure 5: The 6200 cell (a) input structure (b) output structure

particular, we will examine how to change the actual position and routing information of the cells.

Each cell's routing and functionality are controlled by multiplexers, that are in turn selected with SRAM configuration bits local to each cell. Figure 5a shows a diagram of a 6200 cell's inputs. 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. 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 5b.

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 6 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 need to examine 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.

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 and the cell's horizontal position must change from $<c, r>$ to $<maxcol–c, r>$. We similarly change the output routing – north becomes south, south becomes north, the row address r becomes $maxrow – r$, and so forth. For a vertical flip, east/west routing changes do not occur.

A cell's location is determined by the memory address associated with the three data bytes that define its functionality, as shown in Figure 7. This address is

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 6: The three data bytes that control the input and output multiplexers.

Column <5:0>						Column Offset <1:0>		Row <5:0>					
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 7: Address word format for the three programming bytes of Figure 6.

		Initial Configuration		Vertical Flip		Horizontal Flip		Rotate 90 Degrees		Vertical Offset		Horizontal Offset		Final Configuration	
Function Unit Inputs	X1	E4	110	E4	110	W4	100	N4	111	N4	111	N4	111	N4	111
	X2	N	011	S	000	S	000	W	001	W	001	W	001	W	001
	X3	S	000	N	011	N	011	E	001	E	001	E	001	E	001
Function Unit Outputs	Nout	F	00	E	10	W	11	W	11	W	11	W	11	W	11
	Eout	N	01	S	11	N	01	N	01	N	01	N	01	N	01
	Sout	E	01	F	00	F	00	E	01	E	01	E	01	E	01
	Wout	S	11	N	10	S	11	F	00	F	00	F	00	F	00
Coords		Col	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col	Row	Col	Row
		4	2	4	2	0	2	2	0	2	1	4	1	4	1

Figure 8: Complex relocation changes for cell #1 in Figure 4. Shaded areas are values that must be recomputed for the operation performed. Arrows indicate exchanges of values due to reorientation of directions.

composed of a word containing 14 bits. Bits 13:8 and 5:0 denote the column and row of the cell respectively. Bits 7:6, the column offset, control which of the three data bytes shown in Figure 6 are to be written to or read from. To move the location of a particular cell, these 14 bits must be changed appropriately.

For the relocation example of Figures 1 and 4, Figure 8 shows the data changes at each stage in order to relocate cell #1. The actual recalculated values are highlighted, and arrows indicate exchanges of the routing information within the cell due to changes in the cell's orientation. Note that the initial routing configuration is arbitrary but is intended to be realistic given the mapping layout.

First we examine the Vertical Flip stage. X1 is initially set to receive E4 and a vertical flipping of a cell does not change the east-west routing directions. Therefore X1 remains unchanged. Since X2 and X3 are set to N and S respectively, their roles swap when the cell is flipped. Additionally, because the Eout and Wout multiplexers output values from the (former) North and South, their outputs are set to the opposite values due to the new orientation. The coordinates of the cell also are changed from <c, r> to <c, maxrow–r>, but in this case r is coincidentally equal to max row – r.

At the next stage, the Horizontal Flip, the output of the X1 multiplexer changes to W4 because of the exchange of the east and west directions. X2 and X3 remain unchanged because their values are South and North, and these directions are unaffected by a horizontal flip. Eout and Wout exchange values, and Nout changes from east to west. Sout is unchanged because it outputs from the function block. The relative position of the cell is maintained by changing its coordinates from <c, r> to <maxcol–c, r>.

The 90 degree rotation is somewhat more complicated. It involves changing routing so that it is associated with the next most clockwise compass point. Westerly inputs become northerly ones. South would become west, east would become south, and north would become east. Cell outputs are also quite complicated. Although Nout was originally west, it remains west because previously Wout was set to South. Similarly, Eout is set to North because before the rotation Nout was set to West, Sout is set to East because previously Eout was set to North, and Wout is set to the Function block output because Sout was originally that output. The coordinates are then changed from <c, r> to <maxcol–r, c>.

Finally, in the Vertical and Horizontal Offset Stages, each cell of the mapping is moved one row to the south (<c, r> becomes <c, r+1>) and 2 columns to the west (<c', r'> becomes <c'+2, r'>).

X1, X3 Multiplexers

Initial State		Final State	
N	011	E	001
S	000	W	010
E	001	S	000
W	010	N	011
N4	111	E4	110
S4	101	W4	100
E4	110	S4	101
W4	100	N4	111

X2 Multiplexer

Initial State		Final State	
N	011	E	010
S	000	W	001
E	010	S	000
W	001	N	011
N4	111	E4	101
S4	110	W4	100
E4	101	S4	110
W4	100	N4	111

Eout Multiplexer

Initial Nout State		Final Eout State	
N	01	E	10
E	10	S	11
W	11	N	01
F	00	F	00

Wout Multiplexer

Initial Sout State		Final Wout State	
S	11	W	01
E	01	S	11
W	10	N	10
F	00	F	00

Sout Multiplexer

Initial Eout State		Final Sout State	
N	01	E	01
S	11	W	10
E	10	S	11
F	00	F	00

Nout Multiplexer

Initial Wout State		Final Nout State	
N	10	E	10
S	11	W	11
W	01	N	01
F	00	F	00

Figure 9: The SRAM bit changes for the input and output multiplexers for the 90 degree relocation operation

X1 Multiplexer

$$X1[0]' = X1[1]\overline{X1[1]}' + \overline{X1[0]}(X1[2]+X1[1])$$
$$X1[1]' = X1[0]X1[2]X1[1] + \overline{X1[0]}\,(\overline{X1[2]X1[1]})$$
$$X1[2]' = X1[2]$$

X2 Multiplexer

$$X2[0]' = \overline{X1[1]}(\overline{X1[2]X1[0]}) + X1[0]\overline{X1[1]}'$$
$$X2[1]' = X1[2]\overline{X1[1]} + \overline{X1[2]}X1[0]$$
$$X2[2]' = X1[2]$$

X3 Multiplexer

$$X3[0]' = X3[1]\overline{X3[1]}' + \overline{X3[0]}(X3[2]+X3[1])$$
$$X3[1]' = X3[0]X3[2]X3[1] + \overline{X3[0]}\,(\overline{X3[2]X3[1]})$$
$$X3[2]' = X3[2]$$

Eout Multiplexer

$$Eout\,[0]' = Nout\,[1]$$
$$Eout\,[1]' = Nout\,[1]\overline{Nout\,[0]} + \overline{Nout\,[1]}Nout\,[0]$$

Wout Multiplexer

$$Wout\,[0]' = Sout\,[0]$$
$$Wout\,[1]' = Sout\,[1]\overline{Sout\,[0]} + \overline{Sout\,[1]}Sout\,[0]$$

Nout Multiplexer

$$Nout\,[0]' = Wout\,[0]$$
$$Nout\,[1]' = Wout\,[1]$$

Sout Multiplexer

$$Sout\,[0]' = Eout\,[1]\overline{Eout\,[0]} + \overline{Eout\,[1]}Eout\,[0]$$
$$Sout\,[1]' = Eout\,[1]$$

Figure 10: The logic equations necessary to calculate the individual bit changes of Figure 9. These relocation equations are general, and apply to any 90 degree rotation.

From this type of analysis, a distinct set of logic equations can be derived. Figure 9 lists the changes necessary for the most complicated stage, the rotation of 90 degrees. For instance, if the SRAM bits corresponding to multiplexer X1 are set to "W4" encoded by "100", then it changes to "N4" encoded by "111". The output multiplexers are slightly different. For instance, multiplexer Eout will change its state to match whatever Nout was in the incoming configuration. The table shows all such changes needed in the 6200 for rotations of 90 degrees.

Derived from the table in Figure 9, the equations shown in Figure 10 take as input the current state of the various multiplexers and output what the state would become after a rotation of 90 degrees (shown with the ' notation). For instance, for X1, the rotated X1[0] is dependent on the incoming bits 0, 1, and 3 of X1 but is also dependent on bit 1 of the rotated X1. The rotation of 90 degrees has the most complex equations of the three basic manipulations, yet these equations can be implemented in simple logic. Implementing the row and column changes is also trivial, because it involves simple additions and subtractions.

An overall Relocation Pipeline of these changes can be created for the 6200. Each stage in the pipeline corresponds to one of our basic movements (as illustrated in Figure 4) and incoming configurations pass through each stage either modified or untouched by the relocation hardware depending on simple instructions from the CPU. The CPU itself will require only a constant amount of computation to generate the settings for the relocation hardware, independent of the size of a configuration. However, if we forced the CPU to perform each relocation operation on each FPGA cell, it would require computation time proportional to the number of cells in the configuration. Using the custom relocation hardware frees the CPU for other computing tasks.

Limitations 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 cell 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. Thus, usage of the hierarchical, long-distance routing structures rules out most relocation operations.

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 perform both relocation and defragmentation.
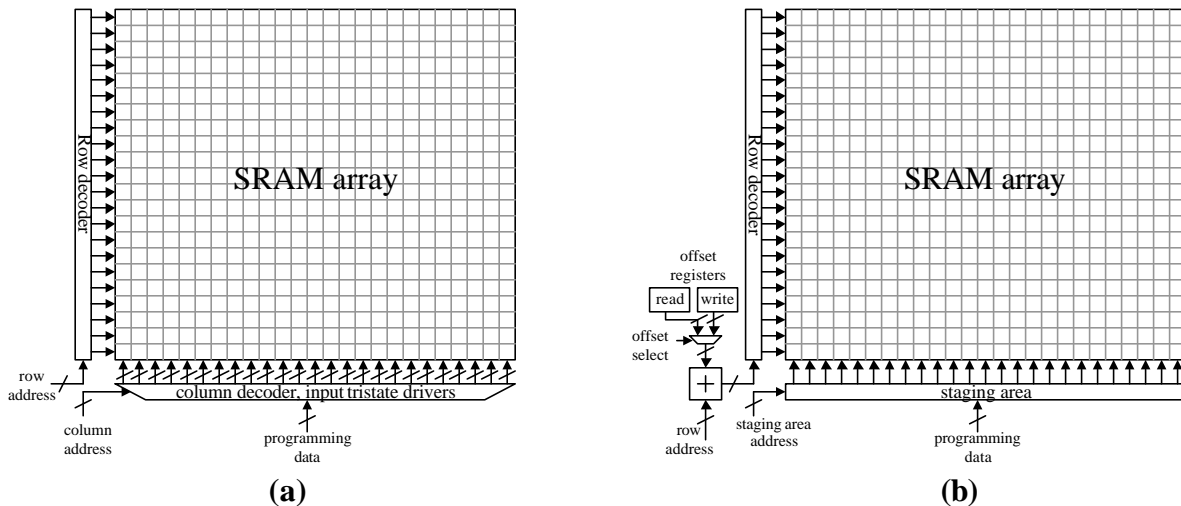

Figure 11: (a) a basic partially reconfigurable FPGA architecture (b) the relocation / defragmentation FPGA architecture

New Architecture

We propose a new architecture designed specifically to exploit the benefits of relocation and defragmentation. 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 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 11a.

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. Most current commercial FPGAs are homogeneous, since they are built around a single, replicated tile.

The third concept is virtualized I/O. Using a bus-based input/output structure provides us with a location-independent method to provide external signals to 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. Also, a global configuration can be used to control the I/O connections of the different configurations, as in the DISC system [Wirthlin95].

One type of virtualized I/O system for a row-based FPGA is shown in Figure 12. 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 external inputs and two external output per column. A cell can select its inputs from the external 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.

For example, in the Chimaera system 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
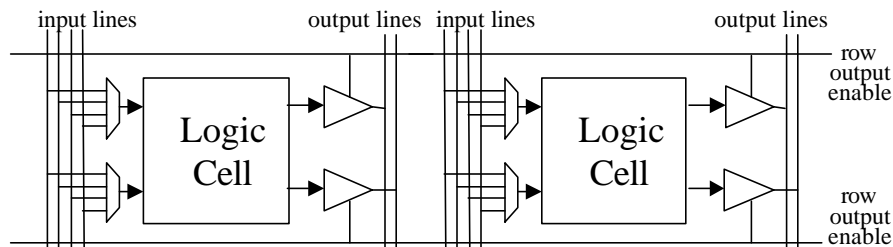

Figure 12: A virtualized I/O structure with four input lines and two output lines. Two cells in one row are shown here. The input and output lines are shared between rows. Although multiple rows may read an input line, only one row at a time may write to any given output line.

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 shown in the example in Figure 12.

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 to where a configuration can be relocated. 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 memory defragmentation techniques can be applied.

Architecture Specifics

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.

Similar 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 11b.

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.

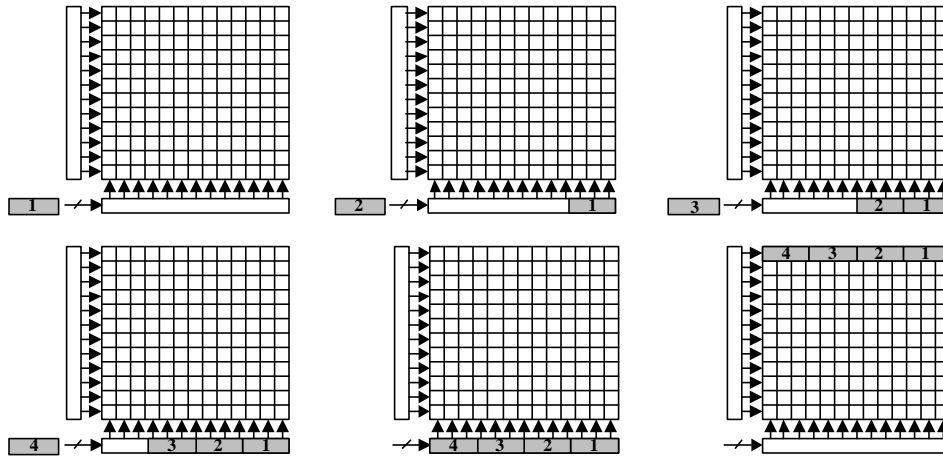The chip row decoder includes a slight modification, namely the addition of two registers, a 2:1 multiplexer

Figure 13: 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.

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.

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 13 illustrates the steps involved in writing a row of configuration data to the FPGA SRAM array. 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. This is for illustrative purposes only. The staging area can be any size, but is expected to be many words wide.

Relocation of a configuration is accomplished by altering the row address provided to the row decoder. This allows for a simple way to dynamically locate individual configurations to fit available free space. Figure 14 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. Each configuration is considered to start at row "0", so the offset indicates exactly which row the configuration should be placed at.

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
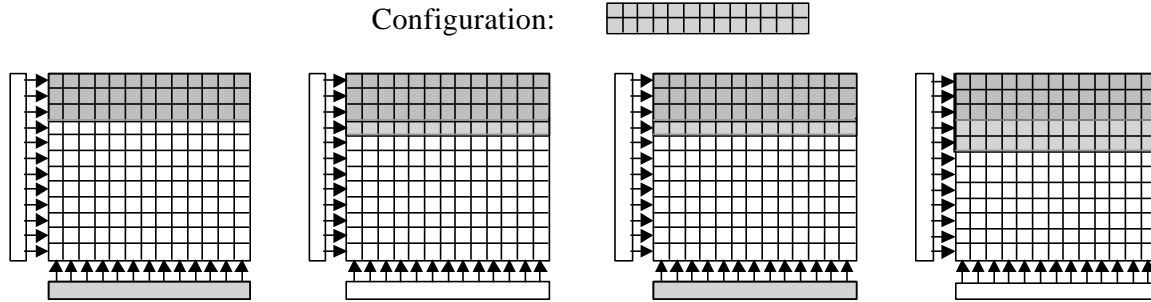
11

Configuration:

Figure 14: An example of a configuration that is relocated as it is written to the FPGA. The actual loading is done one data word at a time, but is shown here as one step for simplicity.

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 plus 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:

$$<\#rows> * (<staging\,area\,size> / <data\,word\,size> + 1) + 1$$

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 $<\#\,rows> + 1$ cycles to the configuration time in order to allow relocation.

Defragmentation of the R/D FPGA requires more steps 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 15 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.

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.
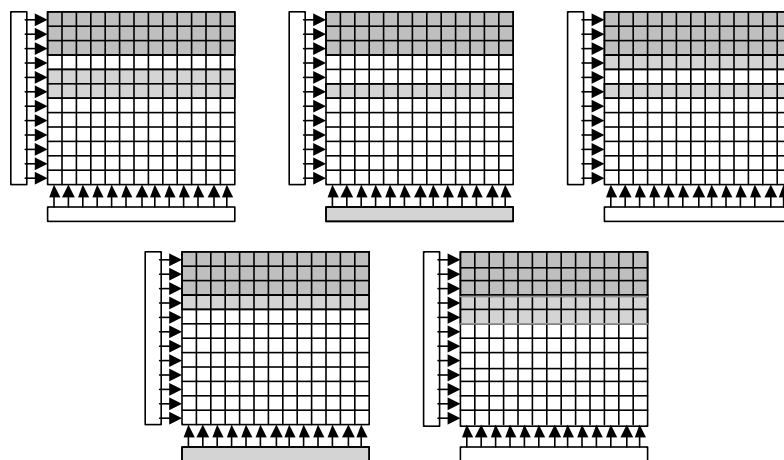

Figure 15: An example of a defragmentation operation. By moving the rows in a top-down fashion for configurations moving upwards in the array, a configuration will not overwrite itself during defragmentation.
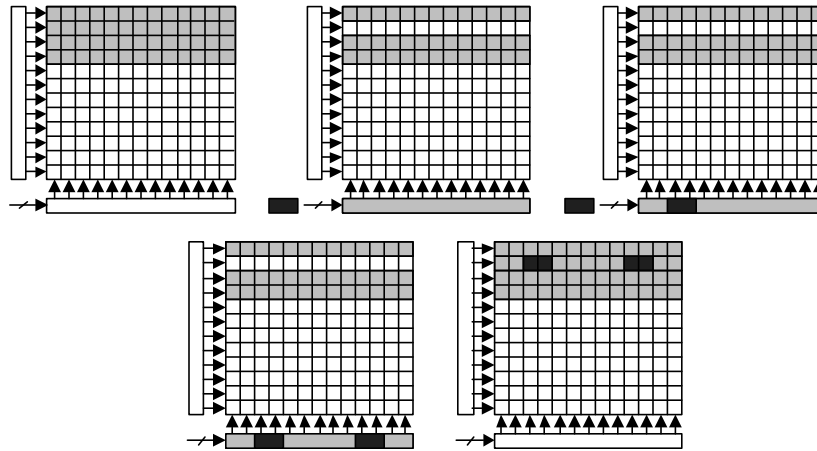
Figure 16: Portions of a configuration can be altered at run-time. This example shows small modifications to a single row of a configuration.

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 signals. There are also two cycles necessary to initialize the two registers. The total number of read/write cycles required to move a configuration is:

$$<\# \ rows> \ * \ 2 + 2$$

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 16. The changed memory cells are 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. This preserves 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:

$$<\# \ rows \ altered> \ * \ 2 + <total \ \# \ changed \ words> + 1$$

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 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.

Also, 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 17. 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.

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:

$$<\# \ rows> + 2$$

Estimated Size Comparison

We modeled the sizes of the basic partially reconfigurable FPGA and the R/D FPGA using the same structures used in [Li00]. The sizes are estimated using the areas of tileable components.

In order to create the area model for the R/D FPGA, we modified the hardware of a basic partially reconfigurable 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.
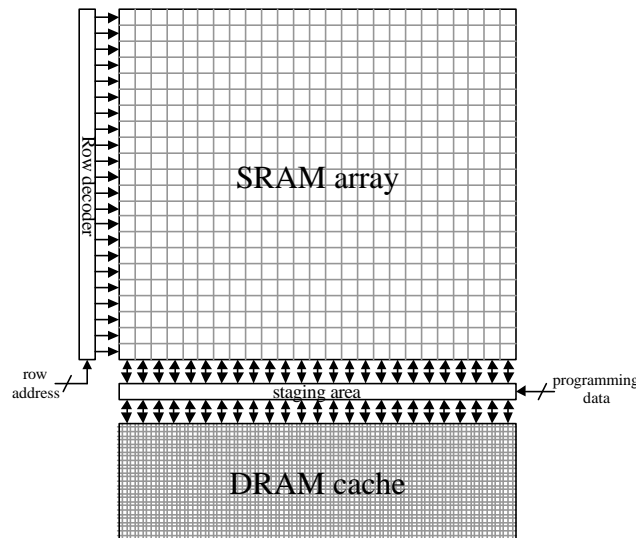

Figure 17: A cache can be attached to the staging area of the R/D FPGA. Entire configuration rows can be fetched from the cache into the staging area, eliminating the per-word loading time required to fill the staging area from the CPU. This cache could be composed of either SRAM or DRAM.

We compared the sizes of the two different styles of FPGA using the base partially reconfigurable FPGA from [Li00], and the R/D FPGA as a base partially reconfigurable FPGA with the modifications listed above. For this size evaluation, we modeled each with a megabit (2^{20} bits) of configuration data in a square layout (# rows = # columns). There are 1024 rows, addressed using 10 bits. For the columns there are 32 32-bit columns, addressed by five bits.

Using the method presented in Li's paper, we consider that the configuration memory area only comprises 25% of the total FPGA chip area. We used the serial traditional FPGA design in order to compute the area of the other 75% of the chip and added this value to our area totals. The area of the partially reconfigurable array was calculated to be 8.547 X 10^9 lambda^2, while the area of the R/D FPGA was calculated to be 8.549 x 10^9 lambda^2, a difference of .0002%. According to this comparison, the R/D FPGA has only a negligible size increase over a basic partially reconfigurable FPGA.

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.

Conclusions

The use of relocation and defragmentation greatly reduces the configuration overhead encountered in reconfigurable computing. In fact, configuration overhead is reduced by as much as a factor of 11 over a serially programmed FPGA when these concepts are used [Li00]. 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 degree 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.

References

[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.

[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. Budiu, 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 FPGAs for Custom Computing Machines*, pp. 87-96, 1997.

[Hauck98]   S. Hauck, "The Roles of FPGAs in Reprogrammable Systems", *Proceedings of the IEEE*, Vol. 86, No. 4, pp. 615-638, April 1998.

[Hauser97]   J. R. Hauser, J. Wawrzynek, ``Garp: A MIPS Processor with a Reconfigurable Coprocessor," *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 12-21, 1997.

[Li 00]   Z. Li, K. Compton, S. Hauck, "Configuration Caching for FPGAs", in preparation for *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[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.

[Trimberger95] S. Trimberger, "Field Programmable Gate Array with Built-In Bitstream Data Expansion", *U.S. Patent 5,426,379*, issued June 20, 1995.

[Wirthlin95]   M.J. Wirthlin, B. L. Hutchings. "A Dynamic Instruction Set Computer", *IEEE Workshop on FPGAs for Custom Computing Machines*, pp 99-107, 1995.

[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^{TM} Configuration Architecture Advanced Users' Guide, Xilinx, Inc., San Jose, CA: 1999.