Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores

Noha Kafafi, Kimberly Bozman, Steven J.E. Wilton
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, B.C., Canada
stevew@ece.ubc.ca

ABSTRACT

As integrated circuits become more and more complex, the ability to make post-fabrication changes will become more and more attractive. This ability can be realized using programmable logic cores. Currently, such cores are available from vendors in the form of a "hard" layout. In this paper, we focus on an alternative approach: vendors supply a synthesizable version of their programmable logic core (a "soft" core) and the integrated circuit designer synthesizes the programmable logic fabric using standard cells. Although this technique suffers increased speed, density, and power overhead, the task of integrating such cores is far easier than the task of integrating "hard" cores into an ASIC. For very small amounts of logic, this ease of use may be more important than the increased overhead. This paper presents two synthesizable programmable logic core architectures, describes the associated place and route CAD tools, and compares the two architectures to each other, and to a "hard" programmable logic core. It also shows how these cores can be made more efficient by creating a non-rectangular architecture, an option not available to "hard" core vendors.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles – *VLSI (very large scale integration)*

General Terms

Design

Keywords

FPGA, Programmable Logic Cores, Standard Cells, System-on-Chip Design

1. INTRODUCTION

Recent years have seen impressive improvements in the achievable density of integrated circuits. In order to maintain this rate of improvement, designers need new techniques to handle the increased complexity inherent in these large chips. One such emerging technique is the System-on-a-Chip (SoC) design methodology. In this methodology, pre-designed and pre-verified blocks, often called cores or intellectual property (IP), are obtained from internal sources or third-parties, and combined onto a single chip. These cores may include embedded processors, memory blocks, or circuits that handle specific processing functions. The SoC designer, who would have only limited knowledge of the structure of these cores, could then combine them onto a chip to implement complex functions.

No matter how seamless the SoC design flow is made, and no matter how careful an SoC designer is, there will always be some chips that are designed, manufactured, and then deemed unsuitable. This may be due to design errors not detected by simulation or it may be due to a change in requirements. This problem is not unique to chips designed using the SoC methodology. However, the SoC methodology provides an elegant solution to the problem: one or more programmable logic cores can be incorporated into the SoC. The programmable logic core is a flexible logic fabric that can be customized to implement any digital circuit after fabrication. Before fabrication, the designer embeds a programmable fabric (consisting of many uncommitted gates and programmable interconnects between the gates). After the fabrication, the designer can then program these gates and the connections between them.

Several companies already provide programmable logic cores [1,2,3,4]. Yet, the use of these cores is still far from mainstream. There are a number of reasons for this:

1. Positioning the programmable logic core and connecting the core to the rest of the chip is not easy, using existing computer-aided design and simulation tools. Although the tools can allow designers to do this, the flow and design tradeoffs are not well understood by most integrated circuit designers. This is somewhat of a chicken-and-egg problem: existing tools and flows will not be enhanced to support the easy incorporation of programmable logic cores until this design technique becomes mainstream, and the design technique will not become mainstream until the tools are enhanced to support programmable logic cores.
2. Often, an integrated circuit would prefer to have many very small regions of programmable logic, rather than a single (or handful of) large programmable logic regions. Often, circuits contain control logic which coordinate the operation of the rest of the chip. It would be beneficial to map selected parts of this control logic to programmable logic, rather than the entire control logic subcircuit.
3. Programmable Logic Cores come in fixed sizes. The integrated circuit designer must choose a programmable logic

FPGA'03, February 23-25, 2003, Monterey, California, USA.

core that is closest to the desired size; this could lead to wastage of chip area.

4. The integrated circuit designer can not modify the internal structure of the programmable logic core.

In this paper, we describe an alternate method for incorporating programmable logic cores into an SoC. Rather than providing "hard" layouts, core vendors would provide "soft" descriptions of their programmable logic cores. These descriptions would typically be written in VHDL or Verilog. The integrated circuit designer could then incorporate the HDL description into their own HDL (for the fixed part of the chip) and synthesize the entire chip using existing synthesis techniques [5,6]. The technique will be described in more detail in Section 2.

Section 3 and 4 will describe new architectures and CAD tools for these cores. Since the cores are intended to be synthesized using standard synthesis tools, it is unlikely that traditional FPGA architectures, that have been optimized for full-custom layout, will be appropriate. Section 5 gives experimental results comparing the architectural alternatives presented in this paper. Finally, Section 6 shows how these cores can be made slightly more efficient by creating a non-rectangular architecture, an option not available to "hard" core vendors.

2. SYNTHESIZABLE PROGRAMMABLE LOGIC CORES

As described in the introduction, integrated circuit designers that wish to use a programmable logic core typically receive a "hard core" which contains the actual physical transistor layout information. The size and shape of the core is fixed; the only freedom the designer has is where to position the core on the chip. Using the alternate scheme described in [5,6], however, the designer receives the core in the form of a "soft core". A "soft core" is one in which the designer obtains a description of the behaviour of the core, written in a hardware description language. Note that this is distinct from the behaviour of the circuit to be implemented in the core, which is determined after fabrication. Here, we are referring to the behaviour of the programmable logic core itself.

Since the designer receives only a description of the behaviour of the core, he or she must use synthesis tools to map the behaviour to gates. These synthesis tools can be the same ones that are used to synthesize the fixed (ASIC) portions of the chip.

The details are as follows:

1. The integrated circuit designer partitions the design into functions that will be implemented using fixed logic and programmable logic, and describes the fixed functions using a hardware description language.
2. The integrated circuit designer obtains a description of the behaviour of a programmable logic core. This behaviour is specified in a hardware description language.
3. The integrated circuit designer merges the behavioural description of the fixed part of the integrated circuit (from step 1) and the behavioural description of the programmable logic core (from step 2), creating a behavioural description of the entire chip.
4. Standard ASIC synthesis, place, and route tools are then used to implement the behavioural description from step 3. In this way, both the programmable logic core and fixed logic are implemented simultaneously.
5. The integrated circuit is fabricated.
6. The integrated circuit designer configures the programmable logic core.

The primary advantage of the new method is that existing ASIC tools can be used to implement the chip. No modifications to the tools are required, and the flow follows a standard integrated circuit design flow that designers are familiar with. This will significantly reduce the design time of chips containing these cores. A second advantage is that this technique allows small blocks of programmable logic to be positioned very close to the fixed logic that connects to the programmable logic. The use of a "hard core" requires that all the programmable logic be grouped into a small number of relatively large blocks. A third advantage is that the new technique allows users to customize the programmable logic core to support his or her needs precisely. This is because the description of the behaviour of the programmable logic core is a text file which can be edited and understood by the user. Finally, it is easy to migrate the circuit to new technologies; new programmable logic cores from the core vendors are not required.

The primary disadvantage of the proposed technique is that the area, power, and speed overhead will be significantly increased, compared to implementing programmable logic using a hard core. Thus, for large amounts of circuitry, this technique would not be suitable. It only makes sense if the amount of programmable logic required is small. An envisaged application might be the next state logic in a state machine. In Section 5, we will quantify this tradeoff.

3. ARCHITECTURES

In this section, we describe two alternative architectures for a synthesizable programmable logic core. The first architecture is very similar to a standard FPGA. As will be shown in Section 5.0, we can improve the density of our core by removing flexibility; the second architecture contains fewer programmable switches and hence is more area-efficient, yet contains enough flexibility to implement small circuits.

3.1 Architecture 1: "Directional Architecture"

The most straightforward way to implement a synthesizable programmable logic core is to describe the behaviour of a standard FPGA at the RTL level using a hardware description language. In doing so, we can make the following observations:

Observation 1: Synthesizable programmable logic cores only make sense for very small amounts of programmable logic. An envisaged application would be the next state logic in a state machine.

Observation 2: Many synthesis tools (the tools that will be used to synthesize the programmable logic core along with the fixed part of the chip) have problems with combinational loops.

These observations motivate us to modify a standard FPGA architecture. First consider Observation 1. Since we are targeting small amounts of logic, we have decided that our architecture will only implement combinational logic, allowing us to remove all flip-flops. Flip-flops can be added at the inputs and outputs of the programmable logic core by the IC designer if desired. Removing flip-flops reduces area and simplifies timing analysis.

Observation 2 is a problem, since an unprogrammed FPGA contains many combinational loops (a good designer will rarely configure the FPGA to contain combinational loops, but before configuration, these loops exist). Recall that one of the primary requirements of our programmable logic core is that it be synthesizable by standard tools. Thus, we have created a "directional" architecture, in which the flow between logic blocks can only flow from left to right. Since our architecture is only to implement combinational circuits, this will not cause a problem; any feedbacks that are required can be implemented outside of the core.

Based on these observations, we have created the architecture shown in Figure 1(a). Each switch block is a standard switch block, with the right-to-left connections removed, as shown in Figure 1(b). The choice of a 3-LUT (as opposed to a 4-LUT or 5-LUT) was based on the observation that the ratio of logic area divided by routing area is larger in a synthesized core than a hand-optimized core; thus, we would expect a smaller LUT to be more efficient (we have performed experiments to confirm this).

3.2 Architecture 2: "Gradual Architecture"

We can attempt to create a more efficient architecture by making the following additional observations:

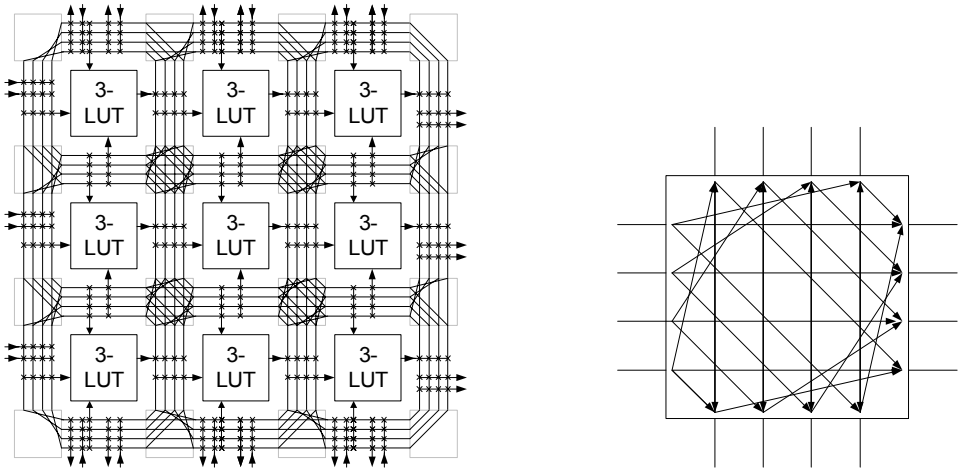
Observation 3: Since we are implementing such small circuits, we can remove some flexibility

Observation 4: Since the core will be hardwired into a fixed-function chip, we still need lots of flexibility on the inputs and outputs.

Observation 5: Unlike a hard FPGA layout, it is not critical that each tile is identical. In a hard layout, FPGA vendors do not wish to layout multiple tiles; in our case, the tiles are synthesized and laid out automatically by CAD tools.

These observations lead to the architecture in Figure 2, which we call the "Gradual Architecture". Like the Directional Architecture, signals in the Gradual Architecture flow from left to right, and the logic resources consist only of 3-LUTs. The horizontal routing channels gradually increase in width from left to right. The vertical tracks are only accessible through LUT outputs (each vertical track can be driven by one LUT), and can be connected to horizontal tracks using a dedicated multiplexor at each grid point. Note that, except for this multiplexor, no switch block is required. The extension of this architecture to any number of rows and columns is straightforward.

The routing multiplexors in the first column are different from the others. We have performed experiments and shown that the primary inputs are frequently required in many different columns. Thus, we have included routing multiplexors in each row (we will vary the number of these multiplexors in Section 5). For each row there are one or more output select multiplexers to choose a primary output of the circuit. The output multiplexers choose between the outputs of all LUTs located in the last column and any horizontal line located above or below that specific row. The exception to this is that only one routing multiplexer per row from the first column passes a signal to the output select multiplexers.



a) Directional Architecture

b) Close-up of Switch Block

Figure 1: Directional Architecture

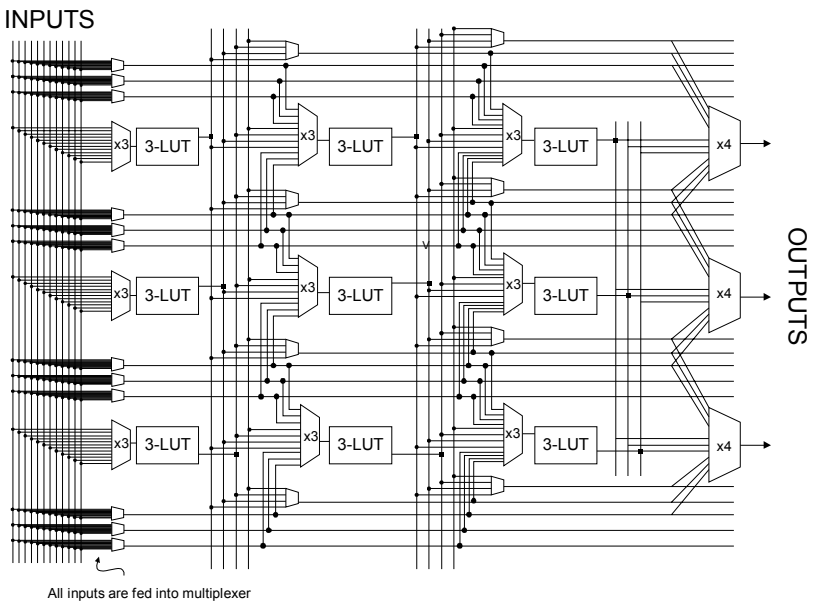


Figure 2: Gradual Architecture

3. CAD ALGORITHMS

Once a programmable logic core has been embedded into a fixed chip, and the chip has been manufactured, the user circuit must be implemented in the programmable logic core. Since our architectures contain novel routing structures, it is likely that existing placement and routing algorithms will not suffice. In this section, we describe placement and routing for the two architectures described in Section 3.0.

It is important to note that we are *not* referring to the standard cell synthesis, placement and routing tools that are used to implement the programmable fabric itself onto the chip. The algorithms in this section are used to implement a user circuit on the programmable fabric after the chip has been fabricated.

4.1 Placement algorithms

a) Directional Architecture

The placement algorithm for the Directional Architecture described in Section 3.1 is based on the original simulated annealing placement algorithm from VPR [7]. The only change was to put a restriction on the placer which stipulates that sources for all blocks must originate from the left of that block. During the annealing, we never allow a move which would result in an illegal placement.

The cost function used in the VPR placement algorithm depends on the delay of potential connections as well as on the Manhattan distance between pins. In a synthesized core, the delay between pins depends on where the individual cells that make up the core are positioned; it may be that blocks adjacent in the conceptual representation of Figure 1 may be positioned far apart in the actual silicon. Nonetheless, we base our placement cost function on the distances and delays in the conceptual representation. We are currently investigating the possibility of "back-annotating" delay and distance information from the implementation of the synthesizable core to get better delay estimation during placement and routing.

b) Gradual Architecture

In the Gradual Architecture, the routing fabric is less flexible than a standard FPGA. Poor placements can easily lead to unroutable implementations. We use a simulated annealing based algorithm, with a unique cost function, as described below.

Figure 3 shows two examples of a "good" placement on the Gradual architecture. In Figure 3(a), a logic block drives logic blocks in an immediately adjacent column. This net can be routed "for free" since no shared resources are required. Note that the multiplexor used to feed each input pin of a logic block is not a shared resource; there is one multiplexor per input pin. Any number of sinks in the column immediately adjacent to the source can be connected in this way.

For nets which drive logic blocks that are not in the immediately adjacent column, the routing mulitplexors must be used. Since these multiplexors are shared resources, we wish to minimize the number of multiplexors used by each net. In the example of Figure 3(b), a net drives four sinks, but only needs one routing multiplexor, since the sinks are all in two vertically adjacent rows (meaning the track between the two rows can be used to drive all sinks). Again note that the multiplexors used to feed the input pins of each logic block are not shared resources, and thus do not play into the cost of a given placement.

The cost function used in our placement algorithm directly relates to the overuse of routing multiplexors. The cost of a given placement on an C-column, R-row core is:

$$\text{Cost} = \sum_{r=0}^{R}\sum_{c=0}^{C}\left[MAX(0, Occ(c,r) - Cap(c,r) + \gamma)\right]$$

where *Occ(c,r)* is the occupancy of the routing multiplexor (defined below) at location (c,r), *Cap(c,r)* is the capacity of the multiplexor (defined below) at location *(c,r)* and γ is a small constant (experimentally, we have found 0.2 works well). The capacity of all routing multiplexors is 1, except for those in the first column, where the capacity is equal to the number of horizontal lines that can be driven by primary inputs (in Figure 2, this would be 3).

The occupancy of a routing multiplexor is an estimate of how many nets would like to use that routing multiplexor. We can write this as the sum of the estimated demand for that multiplexor by each net:

$$Occ(c,r) = \sum_{n \in Nets} demand(c,r,n)$$

where *demand(c,r,n)* is the estimated demand for the routing multiplexor at (c,r) by net *n*. This number is between 0 and 1; 0 means there is no chance that net *n* will want to use this multiplexor, while 1 means that net *n* will definitely want to use this multiplexor. Consider the net in Figure 4(a). In this case, it is equally likely that the net will use the two indicated multiplexors; therefore, the demand term for this net for each of the two multiplexors is 0.5. In Figure 4(b), it is likely that the net will use

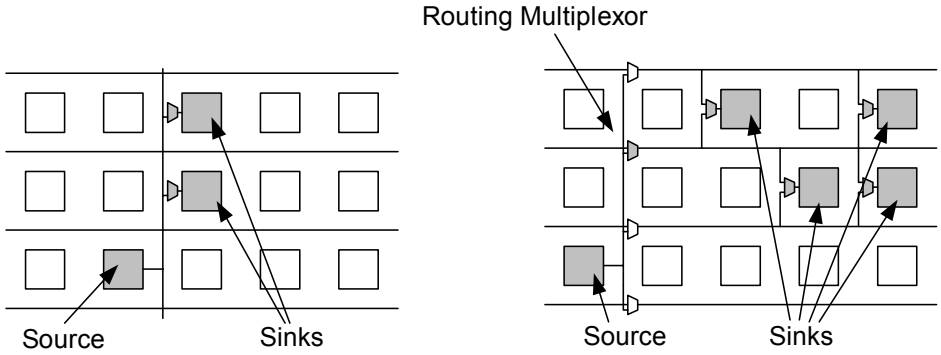


Figure 3: Good Placements on the Gradual Architecture

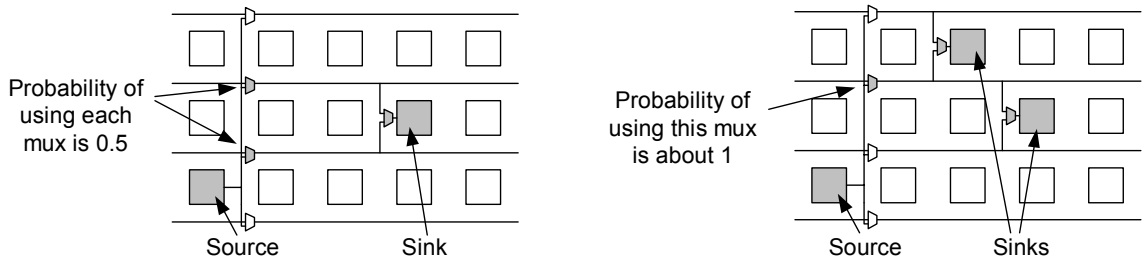


Figure 4: Example Placements on the Gradual Architecture

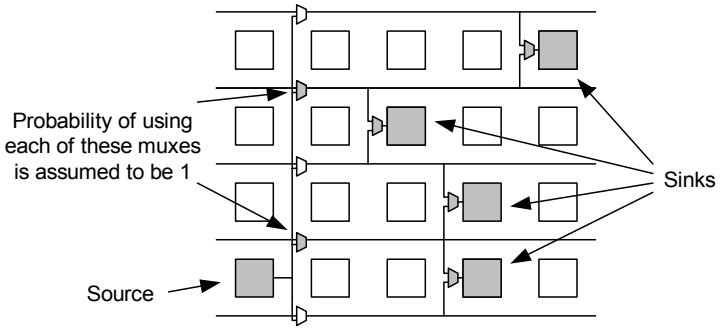


Figure 5: Example Placements on the Gradual Architecture: Sinks in many adjacent rows

Benchmark Circuit	Directional Architecture			Gradual Architecture		
	FPGA Core Size	Tracks per Channel	Cell Area (μm^2)	FPGA Core Size	Input Muxes per row	Cell Area (μm^2)
cc	9x9	4	300 460	8x8	3	263 101
cm138a	5x5	3	80 868	5x5	1	78 319
cm150a	9x9	4	300 460	8x8	3	263 101
cm151a	5x5	3	80 868	4x4	1	43 932
cm152a	4x4	3	53 004	4x4	1	43 932
cm162a	5x5	4	96 854	5x5	2	89 614
cm163a	6x6	5	174 589	5x5	2	89 614
cm42a	5x5	4	96 854	5x5	2	89 614
cm82a	4x4	3	53 004	2x2	1	9 261
cm85a	6x6	4	137 518	6x6	2	128 822
cmb	7x7	3	154 407	7x7	2	184 590
comp	12x12	4	528 332	11x11	2	542 489
con1	4x4	3	53 004	4x4	1	43 932
count	12x12	5	667 344	10x10	4	487 588
cu	8x8	3	199 702	8x8	2	244 676
5xpl	11x11	5	562 305	11x11	2	542 489
i1	8x8	3	199 702	7x7	2	184 590
inc	10x10	5	466 121	10x10	2	424 445
unreg	10x10	4	368 620	9x9	4	388 074
Average			240 737			218 009
Geo. Avg.			175 014			141 954

Table 1: Directional and Gradual Architecture Results

the indicated multiplexor, since a single multiplexor can be used to feed all three sinks, so the demand term for that net is 1. Note that a valid routing could be found that does not use this multiplexor, however, such a route would require two routing multiplexors. During placement, we assume that this will not happen, and thus, set the demand term for all other routing multiplexors for this net to 0. Note that this does not mean the router is constrained to use this routing multiplexor (see Section 4.2).

Finally, Figure 5 shows a net that drives four vertically adjacent rows. In this case, we assume, during placement, that the two indicated routing multiplexors are used with probability 1. Experimentally, we have determined that this leads to better results than if we assign all five routing multiplexors the same value (which would be lower than 1). Again, note that the router is not constrained to actually use the indicated multiplexors.

4.2 Routing algorithms

The negotiated-congestion based routing algorithm from VPR [7] was used for both architectures. For the gradual architecture, the routing task is very easy, since there are only a few potential routes for each net. Nonetheless, the use of a complex router gave us freedom to evaluate different architectures and placement schemes during our architectural investigation.

4. EXPERIMENTAL RESULTS

In this section, we experimentally compare the two architectures described in Section 3. We used 19 small combinational MCNC benchmark circuits. We chose small circuits since these are the type of circuits we expect to be used with our architecture; large circuits would likely be implemented by a hard programmable logic core. For each circuit, we found the minimum-size square core on which the circuit can be placed and routed (in Section 6 we consider shapes other than square). We then created a VHDL description of each core, and synthesized it using Synopsys and a standard 0.18μm CMOS library. The cell area from Synopsys was used for a basis for comparisons (we have completed the physical design of some of the cores using Cadence tools, and have determined that there is a good fidelity between the Synopsys estimates and the final chip area, so we use the Synopsys estimates in this work).

5.1 Directional Architecture vs. Gradual Architecture

The first four columns of Table 1 show the results for the Directional Architecture. For each benchmark circuit, we varied both the core size and the number of tracks in each channel, and chose the configuration which resulted in the minimum area; the chosen size and channel width is shown in Columns Two and Three of the table. For each configuration, we then synthesized

Half as many I/O connections	9.67 %
Default	18.9 %
Twice as many I/O connections	2.33 %
Margin	9.21 %
Conclusion	Sensitive

Percent Difference, baseline algorithms	18.9 %
Percent Difference, fast algorithms	15.5 %
Margin	3.4 %
Conclusion	Slightly Sensitive

Table 2: Sensitivity Results

the architecture using Synopsys; the fourth column in the table shows the cell area required to implement the core.

The final three columns show the results for the Gradual Architecture. In this case, we varied both the core size and the number of input multiplexors per row, and chose the configuration which resulted in the lowest area. These numbers are reported in Columns Five and Six of the table, and the synthesized cell area from Synopsys is shown in the final column. As the table shows, the geometric average of the area required to implement the circuits on the Gradual Architecture is 18.9% less than that required to implement the same circuits using the Directional Architecture.

5.2 Soft vs. Hard Programmable Logic Cores

As mentioned in Section 2, the primary disadvantage of using a "soft" programmable logic core is the reduced density, speed, and increased power consumption. In this subsection, we estimate the density of a soft core compared to a hard core (we have not yet compared the two in terms of speed or power).

The most accurate way to compare the area required by soft and hard programmable logic cores would be to lay out (by hand) a hard core, and compare its area with the numbers in Table 1. This is a time-consuming task. Instead, we estimate the size of a hard core using a detailed transistor-count model, following the methodology described in [8]. We focus on a 4x4 Gradual Architecture with three input multiplexors per row. By estimating the number of Minimum Transistor Equivalents (MTE's) required to implement the circuit, and converting this to area in our 0.18µm technology, we estimate the layout of such a core to require 12 868µm^2. A soft core was generated using these same parameters, and the size (after synthesis using Synopsis and physical design using Cadence) was 81 092µm^2. Thus, the synthesized core is approximately 6.4x less dense than the hard core.

This number is significant. Clearly, for large programmable logic cores, our approach would not be suitable. However, if only small amounts of programmable logic are required, this density penalty may be acceptable. In addition, the use of a hard-core will usually require the selection of a core from a library. Since it is unlikely that a library would contain all sizes and shapes of cores, in most cases, a designer would end up choosing a larger core than is required. Using a soft core, the designer can create a core of any size. Thus, the penalty may not be as bad as the above number suggests.

We have also compared our sizes to commercial FPGA layouts using publicly available information from Chipworks. These comparisons yielded little insight, however, since the commercial devices contain far more tracks per channel, and contain additional elements such as flip-flops.

5.3 Sensitivity of Results

As described in [9], it is critical to analyze results for their sensitivity to experimental assumptions. Table 2 shows two of our sensitivity results for the data in Table 1. The first part of the table shows how the conclusions change if we alter the number of input/output connections per grid. In the experiments in Section 5.1, it was assumed that an nxn Directional Architecture has 2n input/output connections along each of the four edges of the core, and that an nxn Gradual Architecture has 4n input/output connections along the left and right edges of the core. We tried two other input/output ratios, and gathered the results in Table 2. Although the Gradual Architecture always gave better density than the Directional architecture, the margin by which the Gradual was better varied. According to the methodology in [9], we classify this experiment as sensitive to the input/output ratio, even though the conclusion was the same in all cases.

The second part of the table shows how a less aggressive placement schedule (fewer moves per temperature and larger temperature drops during the annealing) and routing schedule (fewer routing attempts) affects the conclusions. In this case, the margin was smaller, meaning the experiment was only slightly sensitive to the choice of algorithm.

5. NON-RECTANGULAR FABRIC

The grid of logic blocks in standard FPGAs is square or rectangular. From [10], however, logic circuits often have a "triangular" shape. In standard FPGAs, this is not a problem, since the routing resources are flexible enough that signals can be routed left, right, up or down, as shown in Figure 6(a). This means that in a standard FPGA, the physical implementation of a circuit need not match the shape of the circuit. In the architectures described in this paper, however, the signal flow is restricted from left to right. As shown in Figure 6(b), this can lead to unused logic blocks if the circuit does not have a naturally square shape.

We can alleviate this problem somewhat by creating a programmable logic core that is not square. We have observed that in many implementations, several logic blocks in the rightmost columns remain unused. We can take advantage of this by removing logic blocks from the last few columns, as shown in Figure 6(c). We quantify the number of logic blocks removed using the parameter *c*, where *c* is defined as the proportion of the logic blocks in the top row that have been removed. In Figure 6(c), *c* is 2/3. In all cases, we remove blocks in a "triangular"

fashion; if we remove *m* blocks from column *i*, we remove *m-1* blocks from column *i-1*. A value of 0 for *c* indicates a rectangular core; a value of 1 indicates a triangular core. Note that a non-zero value of *c* does not imply a non-rectangular layout. The diagram in Figure 6(c) is a conceptual representation; the core will be synthesized into gates, and the gates will be placed into rows regardless of the shape of the conceptual representation.

Intuitively, as *c* is increased, the area of the implementation will go down. If c is decreased too much, however, the area will rise, since a larger grid will be needed. This can be seen in Figure 7. Figure 7(a) shows how the implementation area depends on c for each circuit implemented on the Gradual Architecture (one line per circuit). Because we had problems synthesizing large triangular cores using our synthesis tools, results are only shown for 11 of the 19 benchmark circuits. The geometric average over these 11 circuits is shown in Figure 7(b).

Although each individual circuit in Figure 7(a) shows the expected trends, the results in Figure 7(b) indicate that the gain obtained using a non-zero value of *c* is small. From Figure 7(a), the "breakpoint" (the point at which a larger grid is needed) is not the same for each circuit. Thus, the average results show that only a modest improvement can be achieved. Overall, the value of *c* that gave the lowest area was 0.6, which resulted in a 11.1% lower area than a square core, averaged over all circuits.

6. CONCLUSIONS

In this paper, we have presented two new architectures for synthesizable programmable logic cores. Synthesizable programmable logic cores are different than the programmable cores currently available from vendors in that the cores are obtained as a HDL description, and synthesized using standard synthesis tools. The use of these cores has significant area overhead; we have estimated an overhead of 6.4x compared to using "hard" programmable logic cores. Yet, for small logic

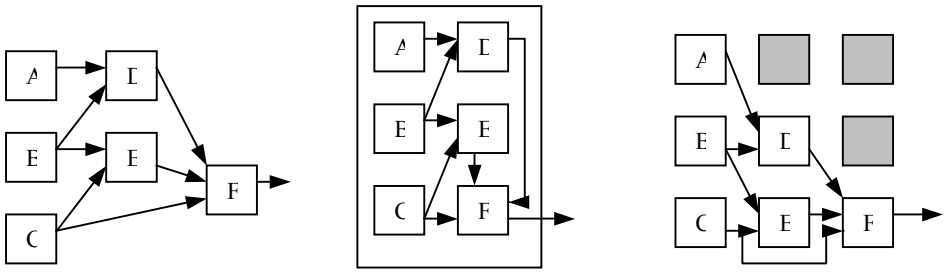
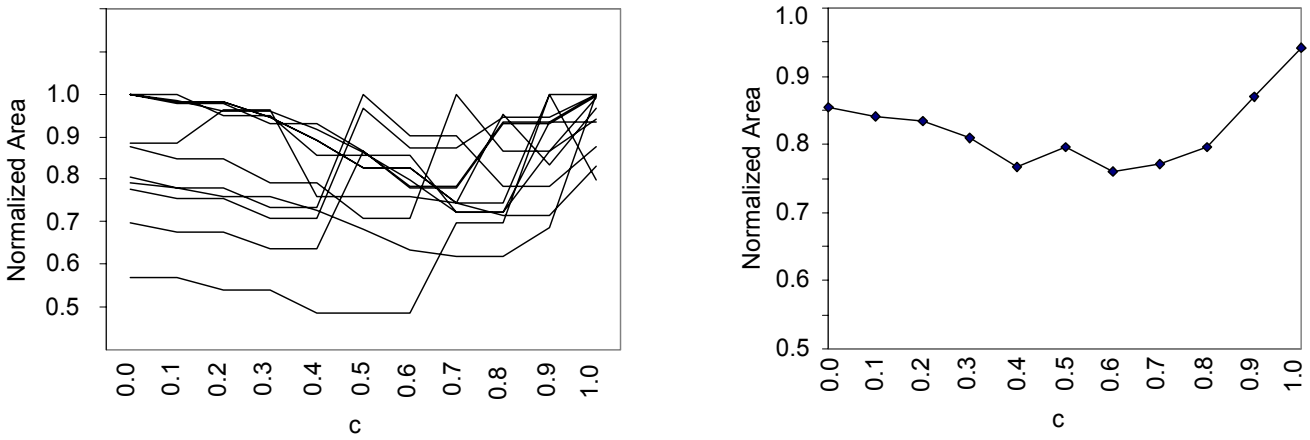


Figure 6: Implementing a circuit on a triangular core



a) One trace per benchmark circuit

b) Geometric Average over Benchmark Circuits

Figure 7: Area as a function of *c* for Gradual Architecture

circuits, these “soft” cores have a number of advantages: they are easy to integrated with fixed logic, cores of any size and shape can be created, and upgrading to a new technology does not require a new hand-layout. One of the primary applications we envisage for these cores is the implementation of next state logic/output logic for state machines.

Our architectures are different than traditional FPGAs in that they only support combinational circuits, and are “directional”, in that signal only flow in one direction through the fabric. In addition, the interconnect pattern is less flexible and the routing resources less plentiful. We have performed experiments to show that small combinational circuits can be implemented on these cores efficiently.

Better synthesis results could be obtained by “tweaking” the standard-cell library to include cells specifically optimized to implement our programmable logic fabric (as was done in [5]). We have not considered this in this paper, since our goal was to create architectures that can be implemented using the standard synthesis tools, cell libraries, and design flows that integrated circuit designers are already familiar with. Nonetheless, if this design technique was to become mainstream, specially-designed standard cells could be created. We have also not considered the power and speed implications of our cores. We suspect that to obtain good speed and power results, some sort of “back-annotation” of detailed routing information is required, so that the tools that understand that logic cells adjacent in the conceptual representations may not actually be physically adjacent on silicon. We are currently investigating these issues.

ACKNOWLEDGEMENTS

Funding was provided by Micronet, Altera, and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] Actel Corp, “VariCore Embedded Programmable Gate Array Core (EPGA) 0.18µm Family”, Datasheet, December 2001.

[2] Leopard Logic Inc, “HyperBlox FP Embedded FPGA Cores”, Product Brief, 2002.

[3] M2000, Inc, “M2000 FLEXEOStm Configurable IP Core”, http://www.m2000.fr.

[4] eASIC, “eASIC 0.13µm Core”, http://www.easic.com/products/easicore013.html.

[5] S. Phillips, S. Hauck, “Automatic Layout of Domain-Specific Reconfigurable Subsystems for Systems-on-a-Chip”, *ACM International Symposium on Field-Programmable Gate Arrays*, Feb. 2002, pp. 165-176.

[6] R. Osann, S. Eltoukhy, S. Mukund, L. Smith, “Programmable Logic Array Embedded in Mask-Programmed ASIC”, World Intellectual Property Org. Patent #WO 01/63766 A2, Feb. 2001.

[7] V. Betz and J. Rose. VPR: A New Packing, Placement, and Routing Tool for FPGA Research. In Proceedings, *International Workshop on Field Programmable Logic and Applications*, Sept. 1997.

[8] V. Betz, J. Rose, and A. Marquardt. Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, 1999.

[9] A. Yan, R. Cheng, S.J.E. Wilton, ``On the Sensitivity of FPGA Architectural Conclusions to the Experimental Assumptions, Tools, and Techniques", in the *ACM International Symposium on Field-Programmable Gate Arrays*, Feb. 2002, pp. 147-156.

[10] M. Hutton, J. Rose, J. Grossman, and D. Corneil, "Characterization and Parameterized Generation of Synthetic Combinational Benchmark Circuits," in IEEE Trans. on CAD, Vol. 17, No. 10, October 1998, pp. 985-996.