

Product-Term-Based Synthesizable Embedded Programmable Logic Cores

Andy Yan and Steven J. E. Wilton, *Senior Member, IEEE*

Abstract—As integrated circuits become increasingly complex, the ability to make post-fabrication changes will become more important and attractive. This capability can be realized by using programmable logic cores. Currently, such cores are available from vendors in the form of “hard” macro layouts. Previous work has suggested an alternative approach: vendors supply a synthesizable version of their programmable logic core and the integrated circuit designer synthesizes the programmable logic fabric using standard cells. Although this technique suffers increased delay, area, and power, the task of integrating such cores is far easier than the task of integrating “hard” cores into an ASIC or system-on-chip (SoC). When implementing a small amount of logic, this ease of use may be more important than the increased overhead. This paper presents a new family of architectures for these “synthesizable” cores; unlike previous architectures, which were based on look-up-tables (LUTs), the new family of architectures is based on a collection of product-term arrays. Compared to LUT-based architectures, the new architectures result in density improvements of 35% and speed improvements of 72% on standard benchmark circuits. The improvement is due to the inherent efficiency of product-term-based designs for small logic circuits. In addition, we describe novel ways of enhancing synthesizable architectures to support sequential logic. We show that directly embedding flip-flops as is done in stand-alone programmable cores will not suffice. Consequently, we present two novel architectures employing our solution and optimize and compare them. Finally, we describe a proof-of-concept layout employing one of our proposed architectures.

Index Terms—Field-programmable gate arrays (FPGAs), programmable logic devices, system-on-chip (SOC) design.

I. INTRODUCTION

RECENT years have seen an impressive improvement in the achievable density of integrated circuits. In order to utilize this excess capacity while maintaining reasonable design and test costs, the system-on-chip (SoC) design methodology has emerged. In this methodology, predesigned and preverified blocks, often called cores, are obtained from internal sources or third parties and are combined onto a single chip. Although this technique partially alleviates some of the complexity, the design and test of a correctly functioning integrated circuit is still difficult.

Manuscript received August 14, 2005; revised February 8, 2006. This work was supported in part by the Altera, Micronet, and the Natural Sciences and Engineering Research Council (NSERC) of Canada. An earlier version of this work appeared in the IEEE International Conference on Field-Programmable Technology, Tokyo, Japan, December 2003, and the Custom Integrated Circuits Conference, Orlando, FL, October 2004.

A. Yan was with the Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC V6T 1Z4, Canada. He is now with Altera Corporation, San Jose, CA 95134 USA (e-mail: ayan@altera.com).

S. J. E. Wilton is with the Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC V6T 1Z4, Canada (e-mail: steview@ece.ubc.ca).

Digital Object Identifier 10.1109/TVLSI.2006.876097

The design costs of manufacturing these chips are also rapidly increasing. A 0.13- μm technology mask set costs approximately \$700 000 USD, a 90-nm set averages about \$1.7 M USD, and a 65-nm set is estimated to cost over \$3.0 M USD [1], [2]. This problem is further aggravated by the fact that ensuring that a design is error-free has become virtually impossible. There will always be some chips that are designed, fabricated, and then deemed unsuitable. This may be due to design errors that are not detected during development and manufacturing or simply due to changes in the design requirements.

A partial solution to this problem is to reduce the number of respins (redesigns due to errors) per design, thereby reducing the costs. This can be accomplished by providing post-fabrication flexibility. There are three main ways to provide flexibility to a SoC. The first is through software. In this method, programmability is achieved by embedding a processor into the SoC. Post-fabrication design changes are accomplished by modifying the software that is executed by the processor. This method is straightforward, but may not provide sufficient performance for some applications. The second method is through the use of programmable registers that configure different modes or options for the desired functionality of the SoC. The advantage of this method is its simplicity, but generally only offers limited flexibility. A third method is through the use of embedded programmable logic cores. A programmable logic core is a flexible logic fabric that can be customized to implement any digital circuit after fabrication [3]–[8]. Before fabrication, the designer embeds a programmable fabric consisting of many uncommitted gates and programmable interconnects between the gates. After fabrication, the designer can then program these gates and the connections between them to implement the desired logic. This method generally offers better performance than software programmability or register configuration. This paper will concentrate on providing post-fabrication flexibility through the use of programmable logic cores.

Post-fabrication flexibility is attractive for a number of reasons. First, it may be possible to postpone some design details until late in the design cycle. Second, as products are upgraded or as standards change, it may be possible to incorporate these changes using the programmable part of the chip, without fabricating an entirely new device. Similarly, it may be possible to fabricate a single chip for an entire family of devices. The characteristics that differentiate each member of the family can be implemented using the programmable logic. Finally, programmable logic provides a mechanism to add debug capability to a fixed-function chip [9]. Several integrated circuits containing programmable logic cores have been described [10]–[14].

Despite these compelling advantages, the use of programmable logic cores has not become mainstream. There are a number of reasons for this. One reason is that designers often find it difficult to identify a subsystem that can be implemented in programmable logic. A second reason is that embedding a core with an unknown function makes timing, power distribution, integration, and verification difficult. Another disadvantage is that embedded programmable logic cores often come in fixed sizes and thus may waste chip area. Lastly, embedded programmable logic cores must address physical connection and placement issues. This is difficult when the regions of fixed and programmable logic are tightly coupled together and/or when there are a number of small programmable pieces distributed over the entire SoC. This extra design complexity limits the use of programmable logic cores to only very advanced VLSI designers.

In [15], an alternative technique is described which addresses the last three concerns by shifting the burden from the SoC designer to mature standard-cell synthesis tools. In this technique, core 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. A “soft core” is one in which the designer obtains a description of the behavior of the core, written in a hardware description language. This description includes flexible resources to implement logic, flexible switches to provide for connections between the logic blocks, and flip-flops to store configuration information. Note that this is distinct from the behavior of the circuit to be implemented in the core, which is determined after fabrication. Here, we are referring to the behavior of the programmable logic fabric itself. Although this technique suffers increased delay, area, and power overhead, the task of integrating such cores is far easier than the task of integrating “hard” cores into a fixed-function chip. For very small amounts of re-programmable logic, this ease of use may be more important than the increased overhead as long as the functional and performance requirements are met.

Since the designer receives only a description of the behavior of the core, he or she must use synthesis tools to map the behavior to gates. These synthesis tools can be the same ones that are used to synthesize the fixed (ASIC) portions of the chip. 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 is connected to it. The use of a “hard core,” however, requires that all of the programmable logic be grouped into a small number of relatively large blocks which can potentially cause floorplanning and timing issues. A third advantage is that the new technique allows users to customize the programmable logic core to precisely support his or her needs. This is because the description of the behavior 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 with implementing programmable logic using a hard core. Thus, for large amounts of circuitry, this technique would not be suitable; a hard programmable logic core should be used. This is a limitation of the soft programmable logic technique; it should only be used if the amount of programmable logic required is small. An envisaged application might be the next state logic in a small state machine that forms part of an intellectual property (IP) core. In this paper, we consider circuits which would require between 25 and 600 three-input lookup-tables (LUTs) to implement in a programmable device.

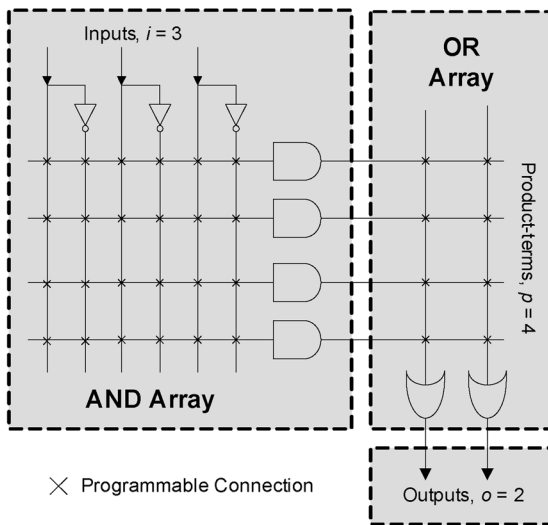
To emphasize this technique further, Fig. 1(a) shows a sample programmable architecture, while Fig. 1(b) shows the corresponding Verilog code. By synthesizing and using standard-cell physical design tools on the code in Fig. 1(b), an implementation of the circuit in Fig. 1(a) can be obtained without creating a custom layout.

In this paper, we present a new family of architectures for a synthesizable embedded programmable logic core. Our core is different from those described in [15] in two ways.

- 1) Unlike the architectures in [15], which are based on LUTs, our new family of architectures is based on a collection of product-term array blocks. It is well known that product-term array blocks can result in density and speed improvements for small circuits [16], [17]; in this paper, we show that the small combinational circuits envisaged for these synthesizable cores are very suitable for product-term-based architectures. Compared to LUT-based architectures, the new architectures result in density improvements of 35% and speed improvements of 72% on standard benchmark circuits.
- 2) A limitation of the architectures proposed in [15] is that they do not support sequential circuits. Supporting sequential circuits in a synthesizable programmable logic core is difficult, since straightforward ways of doing this lead to combinational loops in the fabric; standard synthesis tools have difficulty dealing with circuits containing combinational loops [15]. In this paper, we present and evaluate two ways in which our architecture can be modified to support sequential circuits.

Although these cores make up only a small part of a large SoC, optimizing the area and speed of these cores is important. Reducing the area or delay of the core will allow designers to map more functionality to the programmable logic, leading to a more flexible integrated circuit. As the size of the SoC grows, the number of programmable logic cores that might be embedded increases; thus, the area and speed improvements reported here will be multiplied by the number of programmable logic cores in the design.

This paper is organized as follows. Section II describes our new architecture and presents a set of parameters that can be used to describe the architecture. In addition, Section II also describes how our architecture supports sequential circuits. Section III presents experiments aimed at finding good



a)

```

module ptb (inputs, outputs, clk, scan_in, scan_out );
    input  [2:0] inputs;
    output [1:0] outputs;
    input  clk;
    input  scan_in;
    output scan_out;

    // Create the configuration registers for the PTB
    reg [31:0] config_reg;

    wire [5:0] and_inputs;
    wire [4:0] pterms;

    // Connect the configuration registers together
    always@(posedge clk) begin
        config_reg <= {config_reg[30:0], scan_in};
    end

    // Allow probing of the configuration registers
    assign scan_out = config_reg[31];

    // Create the inverted inputs
    assign and_inputs = {
        inputs[2], ~inputs[2],
        inputs[1], ~inputs[1],
        inputs[0], ~inputs[0]};

    // Create a product-term
    assign pterms[0] =
        (and_inputs[0] | ~config_reg[0])
        & (and_inputs[1] | ~config_reg[1])
        & (and_inputs[2] | ~config_reg[2])
        & (and_inputs[3] | ~config_reg[3])
        & (and_inputs[4] | ~config_reg[4])
        & (and_inputs[5] | ~config_reg[5]);

        ⋮

    // Assign an output
    assign outputs[0] =
        (pterm[0] & config_reg[24])
        | (pterm[1] & config_reg[25])
        | (pterm[2] & config_reg[26])
        | (pterm[3] & config_reg[27]);

        ⋮

endmodule

```

b)

Fig. 1. Programmable architecture and corresponding Verilog code. (a) Programmable architecture. (b) Corresponding Verilog code.

values for the architectural parameters identified in Section II. Section IV presents a quantitative comparison of the our architecture with the LUT-based architectures described in [15]. Finally, Section V describes a proof-of-concept implementation of one of our proposed architectures.

II. PRODUCT-TERM-BASED ARCHITECTURE

Here, we describe our family of product-term based architectures. Each member of the family is composed of one or more product-term-based blocks (PTBs) connected using a novel interconnect architecture. We first describe an architecture that supports combinational circuits only and then show how it can be extended to support sequential circuits.

A. Basic PTB Architecture

Our programmable logic core consists of one or more PTBs. Each PTB consists of an AND plane and an OR plane and can be used to implement one or more combinational logic functions. The AND plane is a product term generator; it is used to create products terms that can be fed into the OR plane. The OR plane is used to “sum” the product terms to create the desired Boolean function. The size of a PTB can be characterized by its number of input pins, the number of product terms (which is equal to the number of AND gates in the AND array), and the number of outputs (which is equal to the number of OR gates in the OR array). In this paper, we will refer to the size of a PTB using the

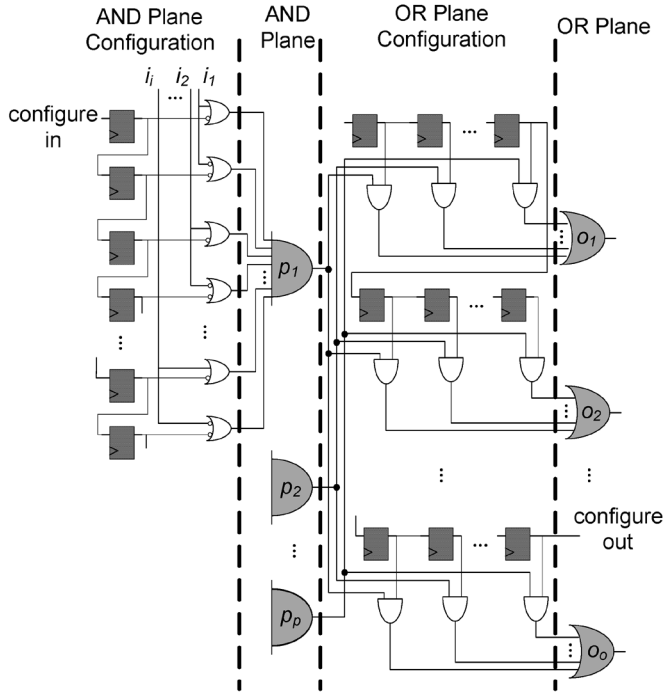


Fig. 2. Schematic of a single PTB.

tuple (i, p, o) , where i is the number of inputs, p is the number of product terms, and o is the number of outputs.

Fig. 2 shows the schematic of the PTB. The programming bits are realized using flip-flops that are daisy-chained together, similar to a scan chain. Although it would be more efficient to use SRAM cells to implement the programming bits, SRAM cells were not considered since we do not have such cells in our standard cell library. To program the PTB, the configuration bit stream is sent into the “configure in” input port. The flip-flop values are used to mask the input signals before driving the AND plane; this mask logic is shown as a set of OR gates, however, ANDs in which the polarity of the input and output signals are switched could also be used. The mask signals control the state of the logic gates that make up the AND and OR plane. A “0” in the flip-flop means the gate is disabled and, conversely, a “1” enables the gates in the plane.

B. Interconnect Architecture

A programmable logic core can be implemented using a single PTB. The behavior of the single PTB can be written in a hardware description language, synthesized, and combined with the rest of the integrated circuit. This works well for small cores, but, as the number of inputs, product terms, and outputs grow, the size of the synthesized fabric will grow. The number of programming bits is proportional to the sum of the product of the number of inputs and product terms, and the product of the number of product terms and outputs, as follows:

$$\text{Number of programming bits} \propto (i * p) + (p * o)$$

where i is the number of inputs, p is the number of product terms, and o is the number of outputs. For large cores, this becomes unwieldy. Because of this, we consider programmable

logic cores with more than one PTBs connected using an interconnect network. Although an interconnect network will undoubtedly impose a delay penalty compared to a programmable core without an interconnect fabric, the delay is small as the depth of the fabric is generally shallow.

Commercial product-term-based devices typically contain a single interconnect switch matrix to connect PTBs to each other and to the I/O pins. This sort of interconnect is not appropriate for our architecture, since it can lead to combinational loops in the unprogrammed fabric (for example, if the output of a PTB is connected to the input of the same PTB). In stand-alone commercial devices, these combinational loops are not a problem, since it is up to the user to configure the fabric in such a way that these combinational loops do not occur. In our case, however, we wish to synthesize the fabric itself using standard synthesis tools. Standard synthesis tools have problems synthesizing circuits with combinational loops. In particular, timing optimization requires the calculation of a well-defined combinational delay; such a well defined delay does not exist in a circuit with combinational loops. Although current synthesis tools can break these loops by arbitrarily removing feedback paths, this often leads to poor results, since there is no way the synthesis tool can know (during synthesis) which feedback paths are actually false paths, and which are important and should be optimized. Thus, we need a fabric without combinational loops.

Fig. 3 shows the routing architecture that we used to connect the PTBs. In this diagram, the select lines for the multiplexers are not shown; they are driven by configuration bits that are configured when the core is configured. We have considered two interconnect strategies. In the first strategy, shown in Fig. 3(a), the PTBs are connected in a rectangular grid, while in the second strategy, shown in Fig. 3(b), the PTBs are connected in a triangular shape (the details of the interconnect will be described below). It is important to note that, since the architectures will be implemented using standard cells and synthesized using standard tools, the shapes of the cores in Fig. 3 may not necessarily match the shapes of the cores when they are laid out. The motivation for using a triangular pattern is that, as shown in [20], logic circuits often have a “triangular” shape. In standard programmable-logic devices, a mismatch between a rectangular architecture and a triangular circuit does not present a problem, since the routing resources are flexible enough that signals can be routed left, right, up, or down, to suit the architecture. This means that, in a standard programmable-logic device, the physical implementation of a circuit need not match the fanout shape of the circuit. In the architectures described in this paper, however, the signal flow is restricted from left to right. Thus, intuitively, we might expect that the triangular architecture will be more efficient at implementing logic circuits than the rectangular architecture. In Section III, we will investigate whether this intuition holds.

Fig. 4 shows a detailed view of our proposed directional routing architecture. Each “Interconnect Switch” labeled in Fig. 3 is implemented using a set of multiplexers as shown in Fig. 4. The inputs of each multiplexer are taken from the outputs of the multiplexers in preceding stages, as well as the primary input pins. The outputs of each multiplexer drive multiplexer inputs in subsequent stages, as well as the primary outputs. The

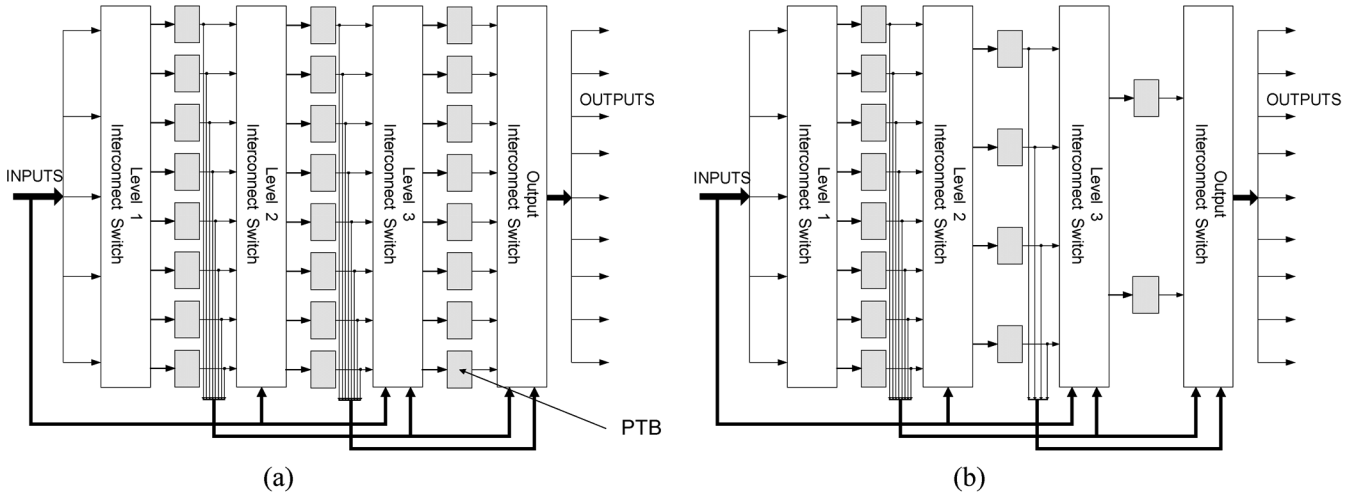


Fig. 3. Routing architectures. (a) Rectangular architecture. (b) Triangular architecture.

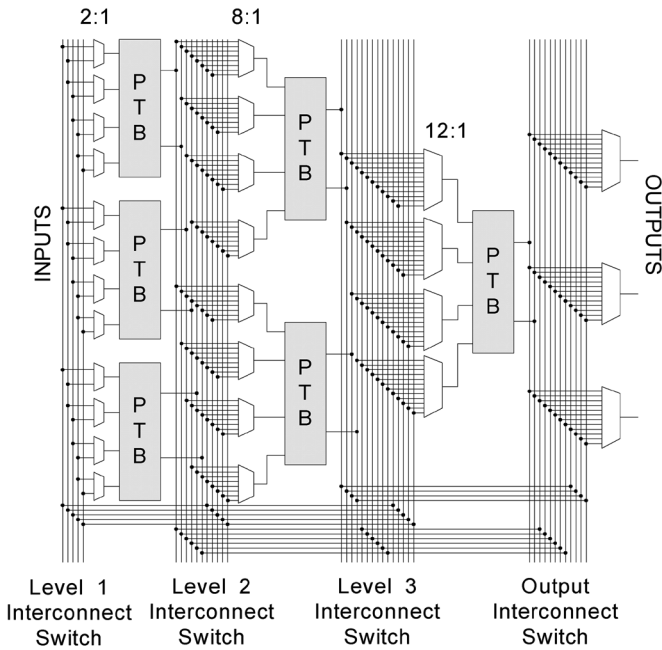


Fig. 4. Details of routing architecture.

select lines of each multiplexer are controlled by configuration bits; the state of these configuration bits would be set when the programmable logic core is configured (after fabrication). The result is a directional architecture, in which any PTB output can drive any other PTB located to the right of itself, but can not drive any of the PTBs located to the left of itself. The multiplexers are standard combinational multiplexers; the signals are not registered between PTB levels. We are able to reduce the size of the multiplexers, by taking advantage of the fact that all input connections to a PTB are equivalent. For a circuit with n primary inputs, to be mapped into a (i, p, o) PTB block, the size of a multiplexer at level l is computed as

$$\text{Mux Size}_l = n + \sum_{s=1}^{s=l-1} C_s \cdot o - (i - 1)$$

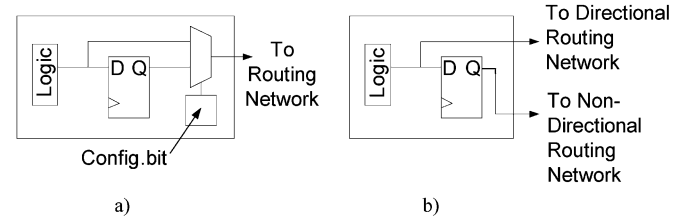


Fig. 5. Adding registers to logic blocks. (a) FPGA-like way. (b) Dual-network architecture.

where C_s denotes the number of PTBs at level s . The size of a routing multiplexer is calculated by summing the number of primary inputs with the number of signal outputs from the PTBs in the preceding levels, and subtracting by the number of inputs in a PTB.

The “fully connected” switch fabric may not seem to be very area-efficient, especially since the multiplexers grow in size with the depth of the core. Because of this, [15] suggested a sparsely populated routing fabric used in their LUT-based architecture. However, in our case, the number of PTB blocks and the depth of the fabric are comparatively small, meaning that the size of the multiplexers do not become unwieldy. An advantage of a fully connected architecture is that the placement task becomes simple and the routing task becomes trivial.

C. Sequential Architectures

Standard FPGA architectures support sequential logic through the use of one or more flip-flops embedded in each logic block, as shown in Fig. 5(a). Each logic block output can be programmably registered or unregistered by changing the configuration bit controlling the select lines of the multiplexer. The logic block can be registered by selecting the registered path or can be unregistered by selecting the combinational path. The problem with this approach for a synthesizable programmable-logic architecture lies in the interconnect. Applying a directional interconnect (meaning that signals can flow from left to right only) to an architecture containing flip-flops as in

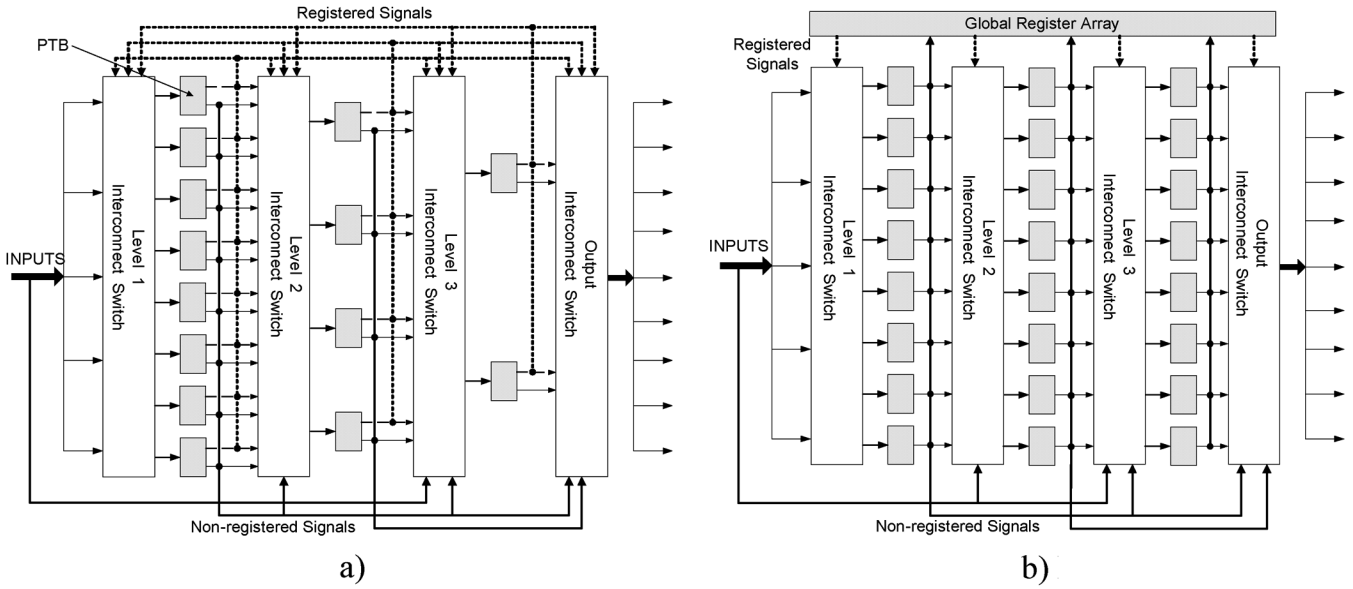


Fig. 6. Architectures that support sequential circuits. (a) Dual-network architecture. (b) Decoupled architecture.

Fig. 5(a) will not suffice, since there will be no way to implement registered feedback loops; such loops are an important part of most sequential circuits such as state machines (where the state variables must be used as inputs to the next state logic). However, simply adding feedback signals to the architecture to support these registered feedback loops will re-introduce combinational loops in the unprogrammed fabric (through the combinational output of the logic block). The same problem was found in [15]; rather than presenting a solution, that work limited the fabric to combinational circuits.

Our approach is to provide both the registered output and combinational output signals outside of the PTB and connect the registered outputs and unregistered outputs to the other PTBs using two separate routing networks, as shown in Fig. 5(b). The unregistered network is “directional,” as in [15], while the registered network can connect an output to a PTB in any level, as shown in Fig. 6(a). In this way, the unprogrammed fabric contains loops, but each loop contains at least one flip-flop, meaning that the synthesis tools will be able to process the circuit effectively. In Section III, we will refer to this architecture as the dual-network architecture.

Unlike the approach in stand-alone FPGAs, this technique requires two routing networks, which significantly increases the size of the fabric. The area overhead is dependent on the number of logic block outputs and the depth of the programmable core. This method of providing sequential circuit support can also be applied in [15]. However, circuits implemented using [15] generally requires more logic blocks with a deeper logic depth, and consequently results in a higher area overhead. The additional area increase for dual-network architectures, however, can be reduced by only including flip-flops for some logic block outputs. In Section II-D, we will present experiments to determine how many logic blocks should contain registered outputs.

An alternative sequential architecture can be motivated by noticing that depopulating the flip-flops as described above reduces the flexibility of the architecture. One way to combat this

is to replace the flip-flops within the logic blocks with a global array of registers, as shown in Fig. 6(b). All logic block outputs connect to a number of “general” registers through an array of multiplexers. The outputs of the registers then feed back into the logic block inputs. By not associating any particular register output to any specific logic block output, utilization of the registers can be greatly increased. This added flexibility, however, comes at a cost of increase area for signal routing. In the next section, we will determine whether this alternative architecture (which we refer to as the decoupled architecture) leads to an overall improvement in density and determine how many global registers are required.

The decoupled scheme could also be used with the architecture from [15], however, in that case, the multiplexers that feed signals to the global register array would be quite large. An extension of this technique to LUT-based fabrics is an interesting area of future work, but will not be discussed here.

D. Architectural Parameters

The previous subsection described our new family of product-term-based architectures. There are two classes of parameters we use to describe a specific fabric within our architectural family: high-level parameters and low-level parameters. Consider a VLSI designer who wishes to employ one of our cores. The designer would have a rough idea of how much logic should fit in the core (and, hence, the size of the core in terms of number of LUTs or PTBs) as well as the number of inputs and outputs of the core. The designer can also provide information on the type of circuits, combinational or sequential, to be implemented on the fabric. The designer would use these quantities, which we refer to as *high-level parameters*, to choose a specific core from a library or a core generator. The high-level parameters are shown in Table I.

Low-level parameters, on the other hand, would not normally be specified by the VLSI designer. These parameters describe the details of the core layout. The designer of the core library

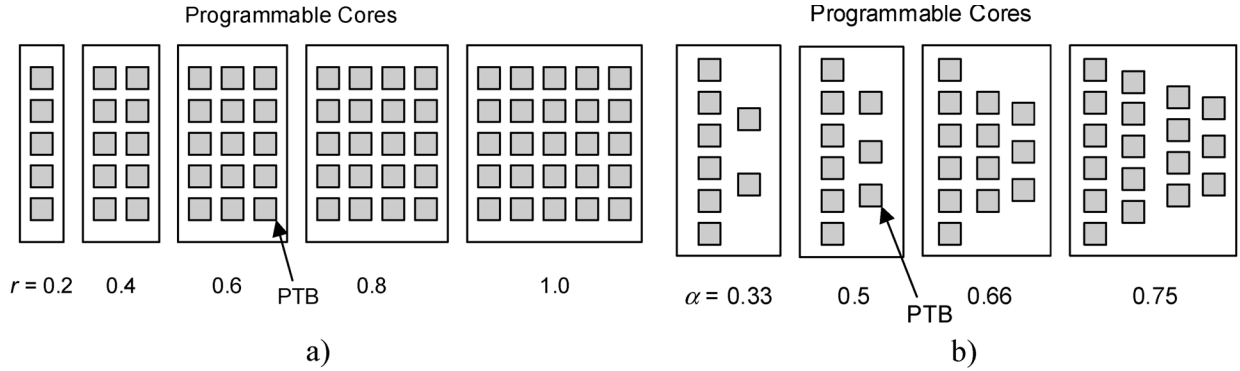


Fig. 7. Illustration of parameters r and α . (a) Parameter r in a rectangular core. (b) Parameter α in a triangular core.

TABLE I
HIGH-LEVEL ARCHITECTURAL PARAMETERS

Parameter	Symbol
Number of Primary Input Pins	PI
Number of Primary Output Pins	PO
Number of PTBs (or LUTs) in the core	n
Sequential or Combinational circuit only	s/c

TABLE II
LOW-LEVEL ARCHITECTURAL PARAMETERS

Parameter	Symbol
Inputs per PTB	i
Product-Terms per PTB	p
Outputs per PTB	o
Rectangular Core: Ratio of levels to number of PTBs in first level	r
Triangular Core: Ratio of PTBs in neighbouring levels	α
Sequential Fabric, Dual-Network Architecture: Number of registered PTBs in each core level	v
Sequential Fabric, Decoupled Architecture: Ratio of global registers to number of PTBs	d

itself (as opposed to the VLSI designer who uses the library) would like to use optimum values for these parameters in the design of each core in the library. The low-level parameters for our synthesizable PTB-based cores are listed in Table II. The first three parameters (i, p, o) characterize the size of the PTB logic blocks. The next two parameters (r, α) detail the interconnect between the PTBs; r determines the depth of the core for rectangular fabrics and α determines the “drop-off” rate of PTBs for triangular fabrics. More precisely, the parameter r is defined as the ratio between the number of logic levels to the number of PTBs in each level, and α indicates the number of PTBs in a given level of a triangular core, relative to the number of PTBs in the preceding level. These parameters are shown graphically in Fig. 7.

The parameters v and d are for the dual-network and decoupled architecture, respectively, and both are used to describe how many flip-flops should be included in the core. Section III will describe each parameter in more detail and seek their optimum values.

III. LOW-LEVEL PARAMETER OPTIMIZATION

Here, we seek optimum values of the low-level parameters listed in Table II. We do not attempt to find optimum values for the high-level parameters in Table I, since these are parameters that would usually be determined by the VLSI designer depending on the applications expected to be mapped to the programmable logic core.

Rather than investigate every possible combination of the parameters in Table II, we sweep each parameter individually, holding the other parameters constant. We start by optimizing the number of product-terms in each PTB, followed by the number of inputs and outputs for each PTB. We then use these values to find the best value of r and α and use these values to compare the rectangular and triangular interconnect patterns. Finally, we consider cores to support sequential circuits by sweeping the number of registers that should be included, and by comparing the dual-network and decoupled architectures. Note that, by optimizing one parameter at a time, it is possible that we will not uncover the true optimum combination of parameters. However, we have chosen the order of parameters such that we optimize those parameters which we expect to have a larger impact on the overall efficiency first. Thus, we expect we end up choosing a good set of parameters, even if it is not optimum.

For most experiments, we present both area and delay results. In general, in cost-sensitive applications, area is the primary optimization criteria, while in high-speed operations, the delay through the programmable core is more important. Thus, we optimize for both using an area*delay metric.

A. Number of Product Terms Per PTB

Intuitively, the larger the number of product terms p available in each PTB, the fewer PTBs are needed to implement a circuit. On the other hand, the size of both the AND and OR plane is roughly proportional to the number of product terms. Thus, we would expect that there is an optimum number of product terms that should be included in each PTB.

To find the optimum value of p , we used an experimental approach, in which we swept p from six to 21 terms. The number of inputs and outputs of each PTB was fixed at $i = 12$ and $o = 3$, respectively; this is in line with previous results [16]. For each potential value of p , we mapped 46 combinational benchmark circuits to the architecture using PLMap [21]. The benchmark

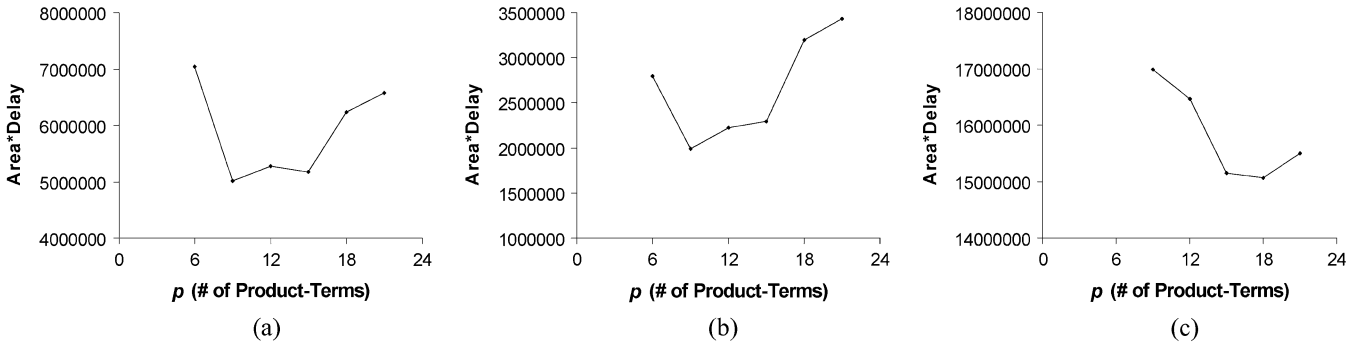


Fig. 8. Area \times delay as a function of the number of product terms per PTB. (a) All benchmark circuits. (b) Small benchmark circuits. (c) Large benchmark circuits.

circuits contained between 10 and 300 equivalent 4-LUTs; we have chosen to use small circuits because they are representative of the circuits that will typically be implemented in a synthesizable programmable logic core. We then use custom place and route tools [22] to determine the smallest triangular core with $\alpha = 0.5$ (experiments have shown that this is a good choice) into which the circuit would fit. The placement tool is a greedy-based algorithm that ensures the placement of the PTBs satisfy the unidirectional signal flow of the architecture. Since the interconnect fabric of the architecture is “fully connected,” the design of the routing tool is simple; the router merely assigns nets to specific multiplexers and configures the select lines to connect the signals properly. More details on the placement and routing tools can be found in [22].

Next, a synthesized behavioral description of the core was created using Synopsys Design Compiler with 0.18- μm technology wire load models from TSMC and standard cell libraries from Virtual Silicon. Even though we are using commercial wire load models, they are still only approximations of the actual loads encountered in a real layout; however, we have determined that there is a good fidelity between synthesized results and physical chip implementation [15]. Also note that, although the efficiency can be improved somewhat by specially constructed synthesis constraints [24], we do not use any such constraints here. The programmable fabric was synthesized as a single unit instead of breaking up into smaller components to take advantage of synthesis optimizations. Experimental results, however, have shown that top-down synthesis versus bottom up synthesis of the programmable core results in similar area and delay values. Area and delay numbers were then gathered to evaluate the architecture.

Notice that we are synthesizing the programmable logic core on its own, without any surrounding logic (the remainder of the ASIC). It is possible that if the programmable logic core and the rest of the ASIC is synthesized together, and the physical design is obtained by placing and routing all cells together, cells that make up the programmable logic fabric may be interspersed with cells that make up other parts of the ASIC. In that case, the wiring delays may deviate (increase) slightly from those obtained here. On the other hand, new synthesis optimizations may appear when the programmable logic and ASIC logic is combined. However, experimental results (such as the proof-of-concept implementation described later in this paper) have shown that the impact is small.

Fig. 8(a) shows the geometric average of the area \times delay product over all benchmark circuits as a function of p . The best value for p is 9. However, we have observed that small circuits tend to prefer a smaller p , while larger circuits tend to prefer a larger p . We repeated our experiments, but partitioned the benchmark circuits into two sets—one set for small circuits (less than 50 equivalent 4-LUTs) and the other set for larger circuits. The results are shown in Fig. 8(b) and (c). We see that small circuits prefer $p = 9$, with $p = 12$ or 15, resulting in a relative difference of 12%–13%. Breaking down the graph into area and delay components (not shown), $p = 12$ or 15 results in a 23% degradation in area compared to nine product terms. For larger circuits, we see that $p = 18$ or 15 provides the best area and delay results. When $p = 9$, we get an 11% increase in area \times delay product. As a result, we propose that cores aimed at small circuits use PTBs with $p = 9$ and cores aimed at larger circuits use PTBs with $p = 18$. This combination results in a 5% improvement in area–delay product than would be obtained by just setting p to 9 for all cores.

B. Number of Inputs Per PTB

We next find the best choice for the number of inputs per PTB. The number of product terms per PTB was set to $p = 9$ or 18, as described in the previous section, and the other parameters were the same as in the product term experiments. The number of PTB inputs i was swept from 8 to 15 inputs, and the area and delay numbers were recorded as before and plotted in Fig. 9. Fig. 9(b) shows that delay decreases as more inputs are added to the PTB. This is because increasing the number of inputs allows more logic to be packed into a single PTB. By increasing the capacity of the PTB, logic depths of mapped circuits gets shorter and, consequently, the delay decreases. In terms of area, Fig. 9(a) shows a minimum at $i = 10$. There are two competing trends in the area graph. If the number of inputs per PTB is too low, then more PTBs are required to map user circuits. However, if there are too many inputs per PTB, then some of the inputs may not be used. Overall, Fig. 9(c) shows that $i = 10$ results in the best area \times delay product. Even for applications in which speed is a primary concern, $i = 10$ gives good results.

C. Number of Outputs Per PTB

The number of outputs o was swept from 1 to 5, while fixing the PTB block parameter i to 10, p to 9 or 18, and assuming a triangular interconnect structure with $\alpha = 0.5$. As shown in

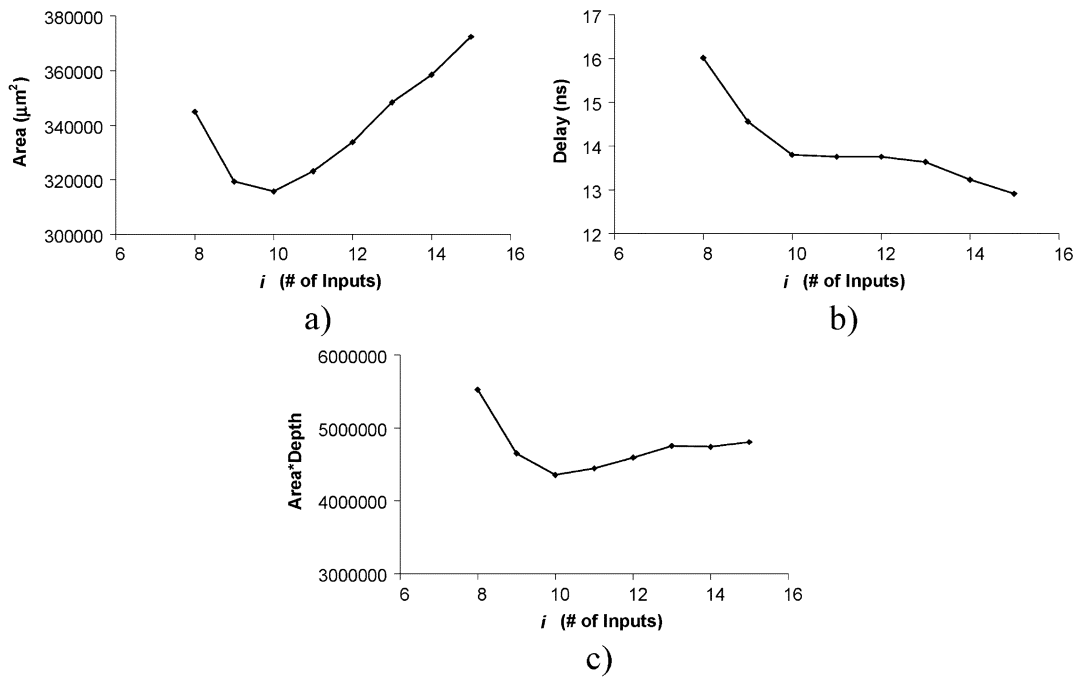


Fig. 9. Impact of the number of input pins in each PTB. (a) Area. (b) Delay. (c) Area \times delay.

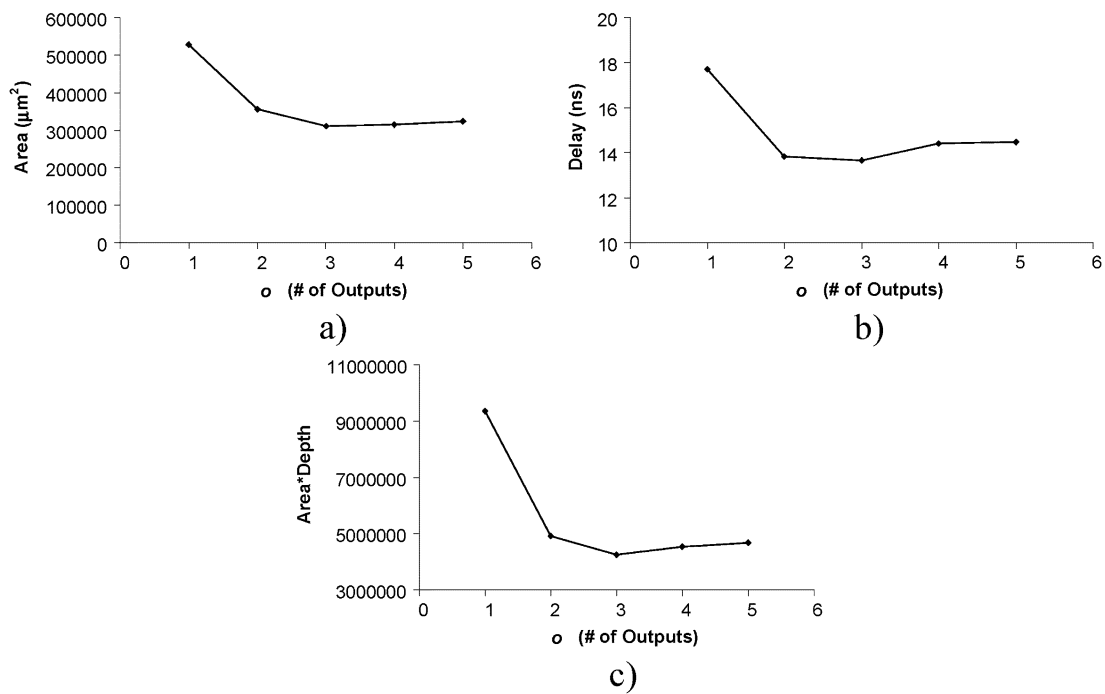


Fig. 10. Impact of the number of output pins in each PTB. (a) Area. (b) Delay. (c) Area \times delay.

Fig. 10(a), there are two competing trends for area. If the number of outputs is too small, more PTBs are required to implement a circuit (this also affects the speed). On the other hand, if the number of outputs is too large, they cannot be used effectively, leading to a waste of area. These two factors lead to a minimum area at $o = 3$. Delay increases as o is increased beyond three, because increasing the number of outputs increases the fanout of the AND gates in the PTB to drive the additional OR gates. Overall, $o = 3$ results in the best area \times delay product; the next best point $o = 4$ results in a 7% area \times delay degradation. The

result $o = 3$ also is the best choice for speed-critical applications.

D. Value of r for Rectangular Architectures

To find the optimum value of r for rectangular cores, we used a similar procedure. The parameter r is defined as the ratio between the number of logic levels to the number of PTBs in each level. PTBs of either ($i = 10, p = 9, o = 3$) or ($i = 10, p = 18, o = 3$) were used (depending on the core size,

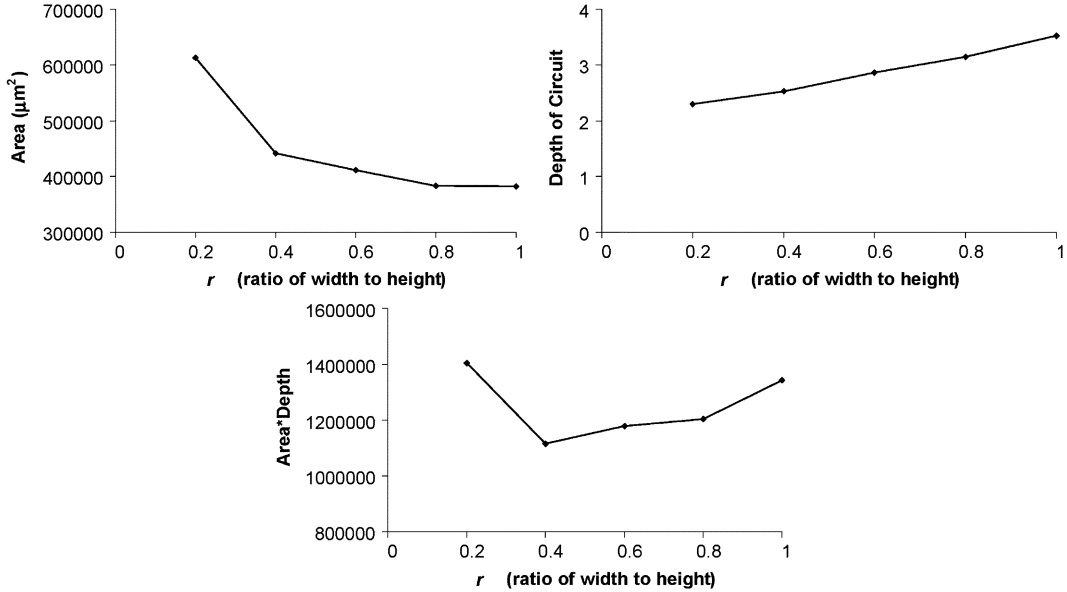


Fig. 11. Area, depth, area \times depth results for rectangular cores as a function of r .

as described above). Fig. 11 shows the impact of this parameter on area, circuit depth, and area \times depth averaged over our benchmark circuits (again, geometric average is used). We have used depth instead of estimated delay in these circuits; since we are using a fixed-size PTB, these quantities are well correlated. As the graph shows, as r increases, the number of levels in the core increases, leading to a longer delay. On the other hand, the area decreases as r increases. A shallow core (small r) places more constraints on the placement of logic, since more stringent precedence relationships must be obeyed (recall that each PTB can drive PTBs in subsequent levels only). Overall, a value of $r = 0.4$ gives the best area \times depth result. For applications in which speed is the primary criteria, a lower value of r would be appropriate.

E. Value of α for Rectangular Architectures

As described in Section II, α indicates the number of PTBs in a given level of a triangular core, relative to the number of PTBs in the preceding level. Given a triangular core that has y PTBs in the first level, there are $n_i = \alpha^{i-1}y$ (rounded to nearest integer) PTBs in the i th level. The number of levels in a triangular core depends on α indirectly. All triangular cores have a minimum of two levels. For $y > 3$, there are x levels, where x is the smallest value for which n_x is less than or equal to 3.

We swept α from 0.33 to 0.8, and for each benchmark circuit, constructed the smallest triangular core (with the specified value of α) in which the circuit would fit. We then synthesized the core and measured area and depth as before. As shown in Fig. 12, a value of $\alpha = 0.5$ is a good choice. We see that both delay and area increases for α larger than 0.5. A large α implies a larger depth and more PTBs, thereby increasing both delay and area. When α is less than 0.5, mapping depth decreases, but area increases. This is because there are not enough levels to map the benchmark circuits for small α , thus the number of PTBs in the first level must increase to provide the required mapping depth.

F. Comparison of Triangular and Rectangular Cores

Comparing the results from Figs. 11 and 12, we can see that the best triangular core leads to 23% smaller area but a 5.2% larger depth (and hence delay) than a rectangular core, on average. The best area \times delay product is 19% lower in a triangular core than in a rectangular core. Thus, we use triangular cores with $\alpha = 0.5$ in the remainder of this paper.

G. Dual-Network Architecture Optimization

Sections III-A–III-F contain results for cores that support combinational circuits only. In the remainder of this section, we study cores that can implement sequential logic as well.

We first consider the dual-network architecture illustrated in Fig. 6(a). In this architecture, flip-flops are associated with PTBs. As described in Section II, we can reduce the area by only including flip-flops in some PTBs. By examining benchmark circuits mapped on an architecture with registers in every PTB, we observed that the utilization of registers in lower levels [which is near the left side of Fig. 6(a)] was significantly lower than the utilization of registers in higher levels [which is near the right side of Fig. 6(a)]. Intuitively, this makes sense; combinational values may need to be computed using several levels of PTBs before they are registered. Thus, rather than depopulating the flip-flops uniformly, we assume the number of PTBs with registered outputs in level i is equal to $r_i = n_i * v^{(l-i+1)}$, where n_i is the number of PTBs in level i , l is the number of levels, and v is a constant that we optimize in this section.

To find an optimum value for v , we mapped 47 MCNC sequential benchmark circuits, each containing between 14 and 296 equivalent 4-LUTs, to PTBs using PLAMap [21]. For each value of v , custom tools from [22] were used to find the smallest architecture onto which each circuit could be mapped. The fabric was then synthesized as before, and the area and depth of this architecture were measured. Again, we have used

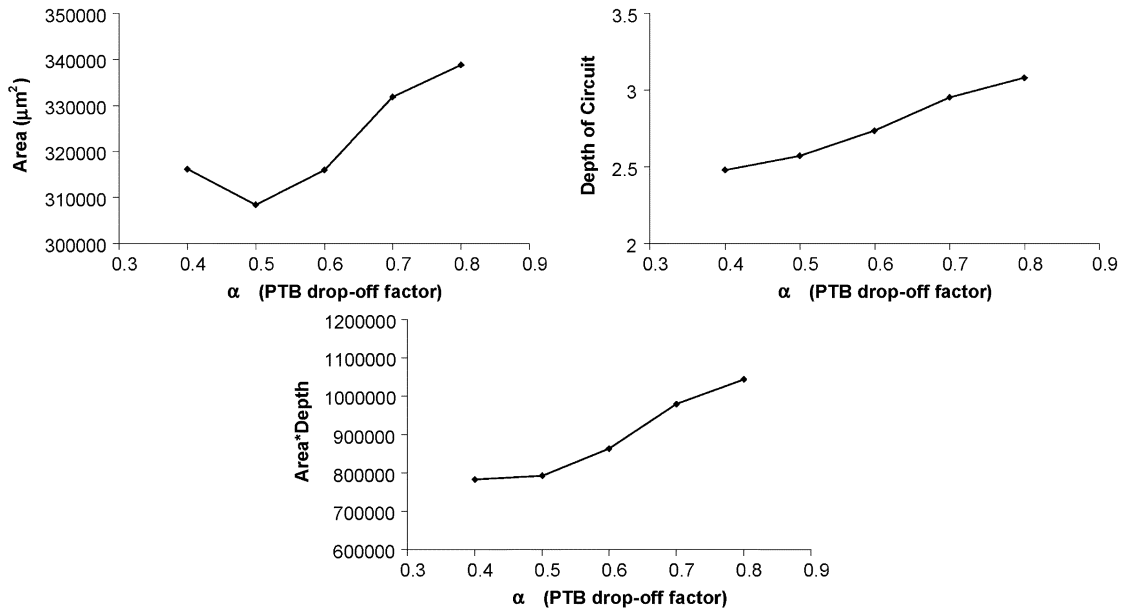


Fig. 12. Area, depth, area \times depth results for triangular cores as a function of α .

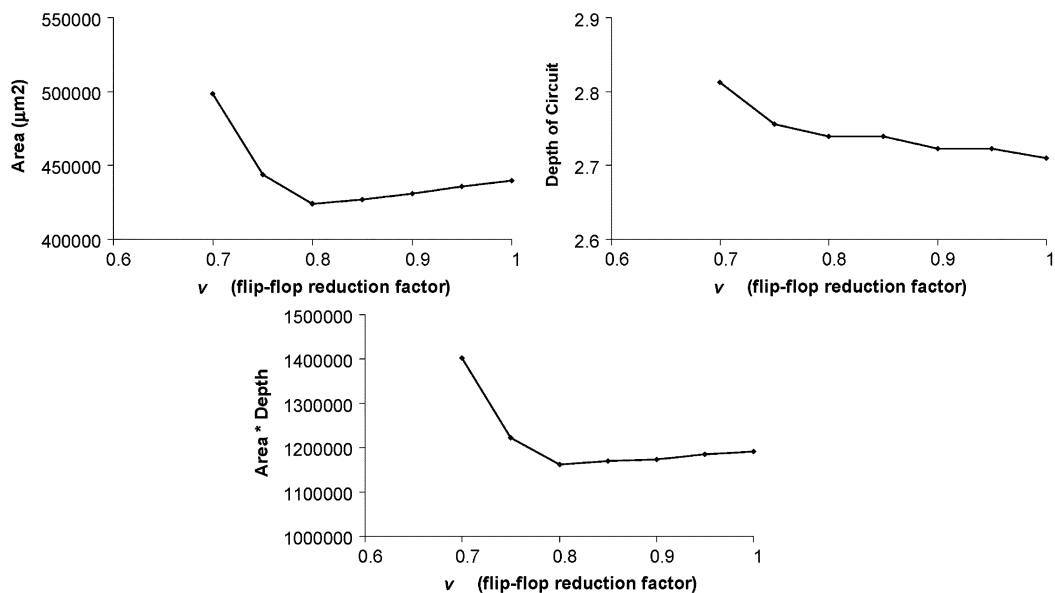


Fig. 13. Area, depth, area \times depth results for the dual-network architecture as a function of v .

depth instead of estimated delay because we found that these quantities were well correlated.

Intuitively, as v increases, the number of flip-flops and the size of the nondirectional network increases, so we would expect the area to rise. On the other hand, if v is too small, it will be more difficult to find a legal mapping for circuits with many sequential elements, and thus we must compensate by choosing a larger core. As shown in Fig. 13, these two competing trends cause a minimum. Overall, the area \times depth is minimum at $v = 0.8$.

H. Decoupled Architecture Optimization

As described in Section II-C, the decoupled architecture contains a global array of registers that can be shared among all

PTBs. In this section, we find the best value of d , which is the number of global registers, as a fraction of the total number of PTB output signals. Thus, if there are n PTBs, each with o outputs, there are $r = n*(d*o)$ flip-flops in the register file. Intuitively, if d is too low, there may not be enough flip-flops to implement a given benchmark circuits, and thus the size of the fabric must be increased to compensate. On the other hand, if d is too large, some flip-flops will remain unused. As shown in Fig. 14, a comprise of $d = 0.5$ provides the best area \times depth tradeoff.

I. Dual-Network and Decoupled Architecture Comparison

Comparing the minimums of the two curves in Figs. 13 and 14, it is clear that a well-optimized decoupled architecture is more efficient than a well-optimized dual-network

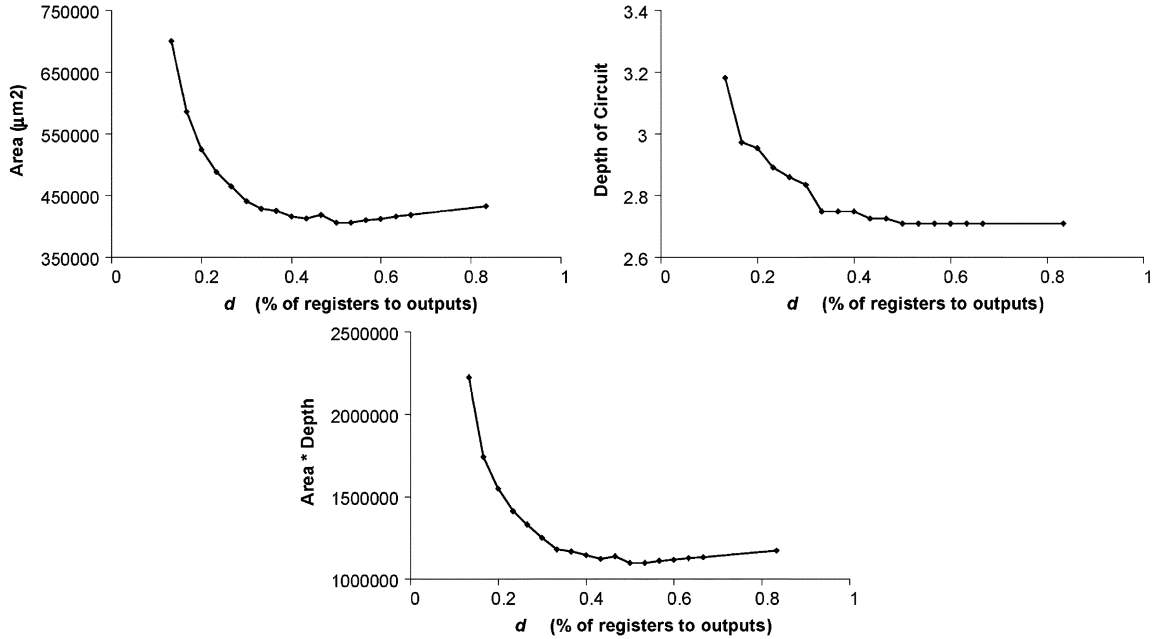


Fig. 14. Area, depth, area \times depth results for the decoupled architecture as a function of v .

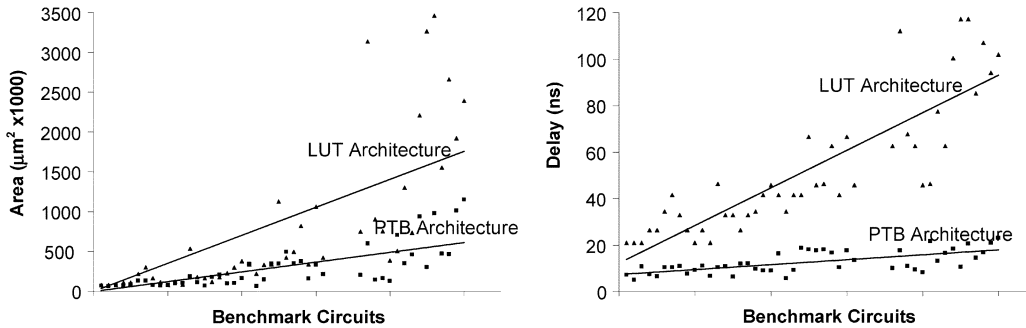


Fig. 15. Comparison to an LUT-based architecture as a function of benchmark circuit size.

architecture. The area \times delay product of the best decoupled architecture is roughly 8% lower than that of the best dual-network architecture.

IV. COMPARISON TO LUT-BASED FABRIC

In this section, we compare the area and delay efficiency of our synthesizable core with that of the LUT-based architecture in [15]. Since the architecture in [15] supports only combinational circuits, we restrict our comparison to combinational circuits. Based on the results in the previous section, we used a triangular core with $\alpha = 0.5$ and PTB input $i = 12$, product terms $p = 9$ or 18, and output $o = 3$.

The results are shown in Table III. Most circuits saw significant area decreases compared to the LUT-based fabric. The improvement is primarily due to the fact that, although the product-term blocks are larger than the blocks in the LUT-based architecture, more circuitry can be implemented within each block. This means that less area is required to implement the circuitry needed to route the signals between the logic blocks. It is also because product-term based structures are inherently more suitable for implementing small logic circuits. A few of the benchmark circuits, however, resulted in area increases for the PTB-

based architecture compared with the LUT-based fabric. This is primarily due to technology mapping the logic to PTBs. Recall that a PTB contains many product terms (and also OR gates) and it is up to the technology mapping algorithm to be as efficient as possible in logic packing. Often full utilization of the product terms (and OR gates) in the PTBs is difficult to achieve. Because of this, certain benchmark circuits may not perform very well when mapped to PTBs. Overall, the PTB-based architecture is 35% smaller and 72% faster than the LUT-based architecture. In the architecture of [15], most of the area and delay was due to large routing multiplexers. In our architecture, since we have larger, and hence fewer, product-term blocks, the routing fabric is simpler and faster. This is more pronounced for larger circuits. Fig. 15 shows this graphically. In these graphs, the benchmark circuits are arranged from smallest to largest along the horizontal axis. For each benchmark circuit, the size and delay of the circuit constructed on the LUT-based fabric of [15] and the size and delay of the circuit constructed on our new PTB fabric are both plotted. For larger benchmark circuits (those towards the right of each graph), the improvement in area and delay is more significant than for the smaller benchmark circuits (those towards the left of each graph).

TABLE III
COMPARISON WITH AN LUT-BASED ARCHITECTURE

	LUT architecture	Product-term architecture	Improvement	LUT architecture	Product-term architecture	Improvement
cm138a	79 450	68 009	14.4 %	20.9	7.1	66.1 %
cm42a	79 670	64 228	19.4 %	20.9	4.9	76.5 %
cm163a	80 365	80 926	-0.7 %	20.9	10.8	48.4 %
cm85a	109 995	72 071	34.5 %	26.3	7.3	72.4 %
rd53	107 263	85 012	20.7 %	26.3	6.4	75.6 %
cm150a	216 144	134 449	37.8 %	34.4	10.3	70.0 %
mux	302 770	134 449	55.6 %	41.5	10.3	75.1 %
cmb	163 693	80 938	50.6 %	32.8	10.9	66.7 %
x2	114 248	70 815	38.0 %	26.3	7.7	70.6 %
cm162a	80 023	78 730	1.6 %	20.9	9.3	55.8 %
pm1	122 989	101 335	17.6 %	26.3	11.1	57.6 %
decod	85 597	103 051	-20.4 %	20.9	6.7	68.2 %
cc	537 513	187 250	65.2 %	46.3	10.4	77.5 %
i1	179 248	116 297	35.1 %	32.8	10.6	67.5 %
misex1	163 588	69 920	57.3 %	32.8	6.3	80.7 %
pcle	120 500	180 362	-49.7 %	26.3	11.9	54.8 %
sct	180 436	206 614	-14.5 %	32.8	12.2	62.9 %
cu	228 576	100 168	56.2 %	34.4	9.8	71.6 %
sqrt8	300 204	105 303	64.9 %	41.5	9.0	78.4 %
sqrt8ml	377 658	164 543	56.4 %	45.7	9.0	80.4 %
lal	340 068	340 425	-0.1 %	41.5	16.3	60.6 %
squar5	218 648	66 302	69.7 %	34.4	5.6	83.9 %
ldd	331 367	147 931	55.4 %	41.5	9.2	77.9 %
pc1er8	336 547	346 406	-2.9 %	41.5	18.8	54.7 %
c8	1 124 829	346 504	69.2 %	66.5	18.1	72.8 %
count	421 721	498 829	-18.3 %	45.7	17.8	61.1 %
comp	496 800	353 452	28.9 %	46.3	18.1	60.9 %
b9	817 867	375 121	54.1 %	62.6	16.7	73.3 %
unreg	339 868	160 713	52.7 %	41.5	10.5	74.8 %
frg1	1 062 808	334 953	68.5 %	66.5	17.7	73.4 %
misex2	420 493	215 676	48.7 %	45.7	13.5	70.4 %
f51m	749 837	201 085	73.2 %	62.6	10.1	83.8 %
cht	3 139 111	601 530	80.8 %	112.2	17.7	84.2 %
b12	902 504	143 963	84.0 %	67.7	10.9	84.0 %
5xp1	756 187	166 027	78.0 %	62.6	9.4	85.0 %
inc	388 765	130 689	66.4 %	45.7	8.3	82.0 %
vg2	507 168	707 288	-39.5 %	46.3	21.6	53.3 %
ttt2	1 300 072	354 500	72.7 %	77.4	13.1	83.0 %
rd73	732 367	459 375	37.3 %	62.6	16.4	73.7 %
term1	2 210 396	938 399	57.5 %	100.4	18.5	81.6 %
9symml	3 266 584	304 526	90.7 %	117.3	10.6	91.0 %
apex7	3 460 155	977 454	71.8 %	117.3	20.7	82.4 %
bw	1 556 027	470 369	69.8 %	85.3	14.4	83.1 %
clip	2 662 875	464 458	82.6 %	107.1	16.9	84.2 %
rd84	1 923 606	1 013 640	47.3 %	94.2	21.0	77.8 %
alu2	2 394 192	1 149 793	52.0 %	102.0	23.0	77.4 %
Average	771 539	292 910		52.3	12.5	
Geomean	411 950	203 951	35 %	45.8	11.6	72 %

The results in this section focus on small- and medium-sized programmable logic cores. As described in the Introduction, those are the types of cores for which we expect the soft methodology to be appropriate. For large amounts of programmable logic, the architecture in [24] may be a better choice. In that architecture, an island-style architecture with a more traditional horizontal and vertical channel routing architecture is employed. By carefully controlling the synthesis results and by synthesizing islands of logic and tiling these islands together, the architecture and approach in [24] will likely be more appropriate for large programmable-logic fabrics. We do not draw a direct comparison between their results and

our results in this paper, simply because we are targeting only small- and medium-sized circuits.

V. PROOF-OF-CONCEPT IMPLEMENTATION

As a proof of concept, we laid out a programmable-logic core using the dual-network architecture with $v = 1.0$ and used it to implement a parallel network interface module from [23], as described in [15]. This module acts as a bridge between a test access mechanism (TAM) circuit [23] and an IP core under test. In the work described in [23], the TAM is a communication network that transfers test data to/from internal IP blocks on the chip in the form of packets. The module we selected allows the

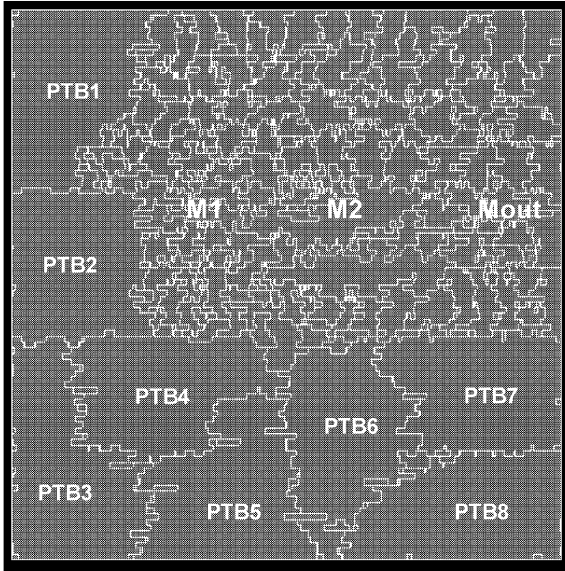


Fig. 16. Floorplan of our proof-of-concept chip.

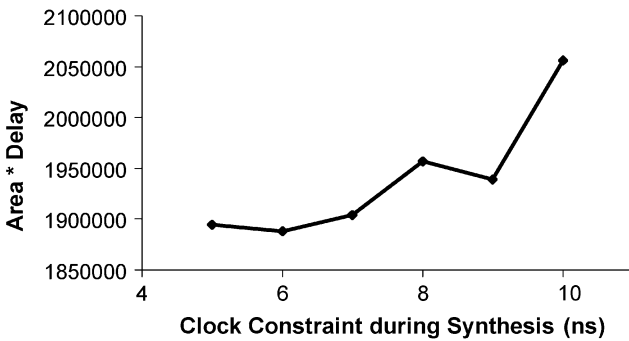


Fig. 17. Area \times delay results for various clock constraints during synthesis.

TAM and the IP core to run at different frequencies, resulting in higher overall TAM throughput. Since such a circuit may become an essential component of future SoCs, it is a suitable circuit in which to prove our concept. More details on the circuit can be found in [15]. Unlike [15], in which only the combinational next-state logic was implemented, in our architecture, we can implement the entire state machine, including the flip-flops.

Fig. 16 shows the floorplan of our chip. The placement of the logic and interconnect blocks closely match the conceptual view in Fig. 3. The PTBs labeled “PTB1-5” and “PTB6-8” are PTBs belonging to the first and second levels, respectively, and the multiplexers labeled “M1, M2, and Mout,” are interconnect blocks in the first, second, and last levels, respectively.

The speed and area required by our core depends on the timing constraints supplied to the synthesis tool. Fig. 17 shows an area \times delay graph for several different timing constraints. The implementation with the lowest area \times delay product had a delay of 6 ns, and an area of 348 000 μm^2 in a 0.18- μm TSMC process. Table IV compares these measurements with those obtained from the LUT-based fabric in [15]. As the table shows, our architecture is 12% smaller and 40% faster than the LUT-based core. The difference is primarily due to the use

TABLE IV
AREA AND SPEED OF OUR PROOF-OF-CONCEPT CHIP

	Product-term fabric	LUT-based fabric [15]
Area (μm^2)	238,000	396,000
Delay (ns)	5.5	10

of a product-term-based architecture rather than a LUT-based architecture.

VI. CONCLUSION

In this paper, we have presented a family of product-term-based architectures for synthesizable programmable logic cores. Synthesizable programmable-logic cores are different than the programmable cores that are currently available from vendors in that they are obtained as a HDL description and synthesized using standard synthesis tools. The use of these cores has significant area overhead. However, for small logic circuits, these “soft” cores have a number of advantages: they are easy to integrate with fixed logic, we can create cores of any size and shape, and they are easy to migrate to a new technology.

Because of the significant overhead, the optimization of the architecture of these cores is important. We have identified several architectural parameters, and have experimentally deduced a good (but not necessarily optimum) set of parameters that leads to efficient implementation of these cores. Overall, we found that the best product-term block with 10 inputs, three outputs, and either nine or 18 product terms (depending on the size of the core) is a good choice. Furthermore, we have shown that these product-term blocks should be connected in a triangular fashion, with each level containing half the number of PTBs as the preceding level. Finally, rather than associating flip-flops with specific logic blocks, as is done in a stand-alone FPGA, it is better to provide a set of general-purpose registers, along with accessing circuitry so that any of the logic blocks can connect to the register file.

Compared with a previously published LUT-based architecture for synthesizable logic cores, we found our architecture to be 35% faster and 72% more dense on standard benchmark circuits. This is significant; the overhead in a programmable-logic core (both speed and area) is large, and reducing this overhead is important. The improvement is primarily due to the fact that, although the product-term blocks are larger than the blocks in the LUT-based architecture, more circuitry can be implemented within each block. This means that less area is required to implement the circuitry needed to route the signals between the logic blocks. It is also because product-term-based structures are inherently more suitable for implementing small logic circuits.

The primary criteria for deciding between a “soft” core and “hard” core is the expected size of the logic to be implemented in the core. If implemented in three input LUTs, the circuits in Table III would require between 25 and 600 LUTs, and require up to 1 mm^2 of chip area (in a 0.18- μm process). For most integrated circuits, it would not be reasonable to implement circuits that are larger than this using a soft core. For circuits smaller than 600 three-input LUTs, however, the soft-core approach may provide a simple way to add flexibility without the need for a specialized CAD flow.

An interesting area of future work is to investigate heterogeneous architectures, in which the parameters of the PTB vary from one level of the structure to another. In a tile-based FPGA, this is usually not an option, since FPGA vendors want to fabricate a single tile and replicate. In our flow, however, since the fabrics are synthesized, it would be reasonable to create PTBs of different sizes in the same fabric.

Better synthesis results could be obtained by adding specialized cells to the standard-cell library to implement our programmable-logic fabric [24]. Also, when synthesis library cells become mature enough to support automatic inference of configuration memory of our programmable fabric, larger sized circuits can be implemented with less area penalty. In that case, however, the challenge of distributing the outputs of each of these memory cells throughout the fabric will become a challenge. 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 are already familiar to integrated circuit designers. If this design technique is to become mainstream, specialized standard cells should be created.

REFERENCES

- [1] M. LaPedus, "IC makers brace for \$3 million mask at 65 nm," *EETimes*, Sept. 2003 [Online]. Available: <http://www.eetimes.com/showArticle.jhtml?articleID=18309458>
- [2] D. Lammers, "Chip industry growing, but experts wonder for how long," *EETimes*, Dec. 2003 [Online]. Available: <http://www.eetimes.com/showArticle.jhtml?articleID=18310473>
- [3] J. Greenbaum, "Reconfigurable logic in SOC systems," in *Proc. Custom Integrated Circuits Conf.*, May 2002, pp. 5–8.
- [4] S. J. E. Wilton and R. Saleh, "Programmable logic IP cores in SoC design: Opportunities and challenges," in *Proc. Custom Integrated Circuits Conf.*, May 2001, pp. 63–66.
- [5] "VariCore Embedded Programmable Gate Array Core (EPGA) 0.18 μm Family" Actel Corp., Mountain View, CA, Dec. 2001.
- [6] "HyperBlox FP Embedded FPGA Cores" Leopard Logic Inc., Cupertino, CA, 2002.
- [7] *M2000 FLEXEOStm Configurable IP Core*, [Online]. Available: <http://www.m2000.fr>, M2000, Inc.
- [8] *eASIC 0.13 μm Core*, [Online]. Available: <http://www.easic.com>, eASIC
- [9] B. R. Quinton and S. J. E. Wilton, "Post-Silicon debug using programmable logic cores," in *Proc. Int. Conf. Field-Programmable Technol.*, Singapore, Dec. 2005, pp. 241–248.
- [10] M. Borgatti, F. Lertora, B. Foret, and L. Cali, "A reconfigurable system featuring dynamically extensible embedded microprocessor, FPGA, and customisable I/O," *IEEE J. Solid-State Circuits*, vol. 38, no. 3, pp. 521–529, Mar. 2003.
- [11] T. Vaida, "PLC advanced technology demonstrator testchip," in *Proc. Custom Integrated Circuits Conf.*, May 2001, pp. 67–70.
- [12] P.S. Zuchowski, C.B. Reynolds, R.J. Grupp, S.G. Davis, B. Cremen, and B. Troxel, "A hybrid ASIC and FPGA architecture," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2002, p. 187, 194.
- [13] T. Vaida, "Reprogrammable processing capabilities of embedded FPGA blocks," in *Proc. IEEE Int. ASIC/SOC Conf.*, Sep. 2001, pp. 180–184.
- [14] F. Lien, J. Feng, E. Huang, C. Sun, T. Liu, N. Liao, and D. Hightower, "A hardware/software solution for embeddable FPGA," in *Proc. Custom Integrated Circuits Conf.*, May 2001, pp. 71–74.
- [15] S. J. E. Wilton, N. Kafafi, J. Wu, K. Bozman, V. Aken'Ova, and R. Saleh, "Design considerations for soft embedded programmable logic cores," *IEEE J. Solid-State Circuits*, vol. 40, no. 2, pp. 485–497, Feb. 2005.
- [16] J.L. Kouloheris and A. E. Gamal, "FPGA performance vs. cell granularity," in *Proc. Custom Integrated Circuits Conf.*, May 1991, pp. 6.2.1–6.2.4.
- [17] A. Kaviani and S. Brown, "The hybrid field programmable architecture," *IEEE Design and Test*, vol. 16, no. 2, pp. 74–83, Apr.–Jun. 1999.
- [18] A. Yan and S.J.E. Wilton, "Product term embedded synthesizable logic cores," in *Proc. IEEE Int. Conf. Field-Programmable Technol.*, Tokyo, Japan, Dec. 2003, pp. 162–169.
- [19] A. Yan and S. J. E. Wilton, "Sequential synthesizable embedded programmable logic cores for system-on-chip," in *Proc. Custom Integrated Circuits Conf.*, Oct. 2004, pp. 435–438.
- [20] M. Hutton, J. Rose, J. Grossman, and D. Corneil, "Characterization and parameterized generation of synthetic combinational benchmark circuits," *IEEE Trans. Comput.-Aided Design*, vol. 17, no. 10, pp. 985–996, Oct. 1998.
- [21] D. Chen, J. Cong, M. Ercegovac, and Z. Huang, "Performance-driven mapping for CPLD architectures," in *Proc. ACM Int. Symp. Field-Programmable Gate Arrays*, Feb. 2001, pp. 39–47.
- [22] A. Yan, "Product-Term Based Synthesizable Embedded Programmable Logic Cores," M.A.Sc. thesis, Dept. Electr. Comput. Eng., Univ. British Columbia, Vancouver, BC, Canada, Jan. 2005.
- [23] M. Nahvi and A. Ivanov, "A packet switching communication-based test access mechanism for system chips," in *Proc. IEEE Eur. Test Workshop*, 2001, pp. 81–86.
- [24] V. Aken'ova, G. Lemieux, and R. Saleh, "An improved soft eFPGA design and implementation strategy," in *Proc. Custom Integrated Circuits Conf.*, San Jose, CA, Sep. 2005, pp. 179–182.



Andy Yan received the B.A.Sc. and M.A.Sc. degrees in electrical and computer engineering from the University of British Columbia, Vancouver, BC, Canada, in 2002 and 2005, respectively.

In the summer of 2004, he was with Intel Corporation, Santa Clara, CA, where he was involved with establishing design and experimental methodologies for embedded field-programmable gate array (FPGA) architectures. He is currently with Altera Corporation, San Jose, CA, as a Software Engineer. His present responsibilities at Altera involve the

development of FPGA models, netlists, and timing files for static timing analysis tools. His areas of interests include programmable logic architectures, ASIC and FPGA design, and CAD algorithms.

Mr. Yan was a recipient of the Best Paper Award at the 2003 International Conference on Field-Programmable Technology for his paper on synthesizable programmable architectures.



Steven J. E. Wilton (S'86-M'97-SM'03) received the M.A.Sc. and Ph.D. degrees in electrical and computer engineering from the University of Toronto, Toronto, ON, Canada, in 1992 and 1997, respectively.

In 1997, he joined the Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC, Canada, where he is now an Associate Professor. During 2003 and 2004, he was a Visiting Professor with the Department of Computing, Imperial College, London, U.K., and with the Interuniversity MicroElectronics Center (IMEC), Leuven, Belgium. He has also served as a consultant for Cypress Semiconductor and Altera Corporation. His research focuses on the architecture of field-programmable gate arrays (FPGAs) and the CAD tools that target these devices.

Dr. Wilton was the Program Chair for the ACM International Symposium on Field-Programmable Gate Arrays and the program co-chair for the International Conference on Field Programmable Logic and Applications in 2005. He was the recipient of the Best Paper Awards at the International Conference on Field-Programmable Technology in 2003 and 2005, and at the International Conference on Field-Programmable Logic and Applications in 2001 and 2004. In 1998, he won the Douglas Colton Medal for Research Excellence for his research into FPGA memory architectures.