# Soft++: An Improved Embedded FPGA Methodology for SoC Designs

Victor Aken'Ova, Guy Lemieux, Resve Saleh
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, B.C., Canada.V6T 1Z4

vaken@ece.ubc.ca, lemieux@ece.ubc.ca, res@ece.ubc.ca

## ABSTRACT

Embedding field-programmable gate array (eFPGA) cores in System-on-Chip (SoC) designs provides an attractive option due to potential cost savings afforded by the ability to make post-fabrication changes. eFPGAs are available in two forms: as hard IP cores and as soft IP or "synthesizable" cores. The hard IP form is limiting because only a small number of fixed-size and fixed-architecture cores are available, none of which may be optimal in terms of area, power or delay for a given SoC application. A purely soft IP form of programmable logic would remove this restriction and enable significant opportunities for architecture customization. Previous research has described a *Soft eFPGA* methodology which uses the ASIC design flow to create small amounts of programmable logic using standard cells. Although highly flexible and easy to use, this approach incurs significant penalties in layout area, power and delay due to its use of generic standard cells. This paper illustrates that it is possible to stay within the ASIC flow and still reduce a significant portion of this overhead by: (1) using a structured layout, (2) adopting a tile-based architecture, and (3) employing architecture-specific tactical standard cells. We call this the *Soft++ eFPGA* methodology. Using this approach, we achieved area and delay improvements averaging 58% and 40%, respectively, over the original Soft eFPGA approach.

## I. INTRODUCTION

The increasing density and complexity of integrated circuit (IC) designs in deep submicron (DSM) technology has led to the emergence of System-on-Chip (SoC) methodologies [1]. The primary driver of this paradigm shift is productivity improvement through Intellectual Property (IP) reuse, where pre-designed and pre-verified IP blocks are used to construct large complex chips. Unfortunately, this has also resulted in significant increases in the cost of ICs due to corresponding increases in engineering and mask costs on the order of tens of millions of dollars [2]. As a result, designers are pursuing software and hardware methods to build programmable SoCs and avoid the extra cost of chip re-spins. For example, a programmable SoC can be used to implement feature enhancements, functionality improvements, standards updates or perhaps to fix design errors that are caught after chip tape-out [3]. Such flexibility can help to amortize chip development costs over several design derivatives.

Embedded FPGAs (eFPGAs) have emerged as a natural hardware solution to this growing challenge because they allow logic functionality to be changed after fabrication. Such cores are generally suitable for small and medium embedded functions such as processor accelerator functions to speed up embedded software, data encryption circuits that may need to be updated from time to time for security, and also I/O interface protocols for data communication [4].

Despite the potential applications and cost benefits of eFPGAs in SoC design, their commercial success has been limited by a number of issues. The biggest issue is the high area, power, and delay overhead that programmable logic architectures generally incur. Although the same issues exist in stand-alone FPGA chips, the implications for eFPGAs are more troublesome because their intended applications are generally in high-performance SoCs with stringent area, speed, and power budgets. There is also an additional source of inefficiency due to the limited offerings in size (logic, interconnect, and embedded memory capacity) and architecture (e.g., logic resources based on lookup tables or product terms) of such cores. Furthermore, vendors typically include excess resources in each core to accommodate a broad range of targeted applications. Unfortunately, this “one-size-fits-all” philosophy introduces the likelihood that resources will be left unused (or under-utilized) by an application. Unused resources effectively increase the area, speed, and power overhead of using eFPGAs in SoCs.

Recent work [5][6][7] has focused on ways to make eFPGA customization relatively inexpensive. Using the *Soft* eFPGA approach described in previous work [6], it is possible to use the ASIC flow and standard cells for ***on-demand*** creation of programmable logic cores tailored to a specific application. Although this new approach makes it possible to significantly cut logic and routing overcapacity, its exclusive use of CMOS standard cells incurs a significant area, power and delay penalty that offsets any benefits. For example, the wide fan-in multiplexers in Soft eFPGAs are implemented using static CMOS logic gates [6][8] which are inefficient for this purpose. Furthermore, the configuration memory within these cores (which accounts for almost half the programmable logic core area [8]) is implemented using standard cell flip-flops rather than more area-efficient SRAM cells.

In this work, we investigate improvements to automatic generation of programmable logic fabrics using the ASIC flow. Our initial work in this area [17][22] demonstrated the viability of the approach. Here, we provide details on the methodology, circuit and layout aspects to support the approach. Specifically, we report the effectiveness of tactical standard cells [9] in the design of eFPGAs. Additionally, we describe a modified ASIC flow approach for the design and implementation of such fabrics. We define our use of tactical cells and a structured (tile-based) layout as the *Soft++ eFPGA* approach. This leads to a much more efficient way of generating programmable fabrics, yet still retains the strengths and familiarity of a traditional ASIC-based design flow.

This paper is organized as follows: Section II describes the hard and soft approaches in further detail. Section III details the architecture, circuit techniques, and ASIC flow that are used to implement Soft++ eFPGAs. Section IV presents key results and comparisons with other approaches including Soft eFPGA. Conclusions are provided in Section V.

## II. BACKGROUND AND RELATED WORK

While eFPGAs can potentially afford designers a tremendous amount of flexibility, the significant overhead associated with this approach makes them unattractive. For example, the area of FPGAs can be 40X higher, critical path delays can be 3X to 4X slower, and power dissipation is roughly 12X higher compared to an equivalent ASIC [11]. Consequently, there has been a recent increase in research [5][6][7][12][13] to improve the quality of *automatically generated* eFPGA layouts. These efforts attempt to use domain-specific knowledge to build customized FPGA architectures and layouts that are optimized for a given application. It is important to note that manual generation of such programmable fabrics is prohibitively time-consuming, making auto-generation essential. In one research effort [13], a new custom CAD flow and toolset were developed to generate FPGA layouts. Our belief is that if the generation process uses existing FPGA and ASIC design flows, rather than new custom flows, eFPGAs would be more accessible to mainstream SoC designers.

There are two main phases in the design and implementation of an embedded FPGA: architecture determination and architecture generation. The first phase, which is not the focus of this work, involves evaluating a wide range of possible eFPGA architectures to meet the area/delay/power requirements when implementing user logic circuits. This second phase involves creating a layout implementation from the architecture parameters determined in the first stage. It is this latter phase that is the focus of this paper.

In the IC industry, *Hard eFPGAs* are available from some vendors in the form of a final hard layout as IP [14][15]. In this case, for practical reasons, the vendors offer a fixed inventory of architectures which are determined to be suitable for a very wide range of possible applications. For example, Actel's Varicore [14] is offered only in sizes of 512, 1024, 2048, and 4096 four-input LUTs. While the vendor focuses effort on producing extremely dense layouts, it is likely that the limited architecture inventory contains significant overcapacity in logic and/or routing for specific applications.

More recently, a new approach for *Soft eFPGA* design was proposed [6][16] where the user can generate their own eFPGA using a high-level parameterized description of the architecture, from which a final layout is generated. In this case, there are a wide range of possible architecture instances and the most suitable one can be selected. This eliminates overcapacity problems, but it is inefficient because the synthesized architecture must be implemented in standard cells.

It is also possible for a *Custom eFPGA* design to be pursued. In this case, similar to Soft eFPGA, the user also generates a tailored eFPGA architecture instance. However, the user must employ additional custom layout effort or specialized layout tools such as [13] to obtain a more dense layout. This requires the expenditure of additional resources, which may or may not be worth the effort and expense.

In this paper, we detail the *Soft++ eFPGA* design flow. This improves upon the Soft eFPGA design flow by selectively employing one strength of the Custom eFPGA flow, namely the use of custom layout in the limited form of tactical standard cells, but avoiding its weakness by avoiding the need for new layout tools or extended layout effort. Although the use of tactical cells to improve the quality of ASIC designs is fairly well-known [9], there have been few attempts to apply these same techniques to programmable logic fabrics. The improved flow also addresses a weakness of the Soft eFPGA flow described [8], which is the unpredictability of timing paths when there is no structure in the eFPGA layout.

## III. SOFT++ eFPGA DESIGN APPROACH

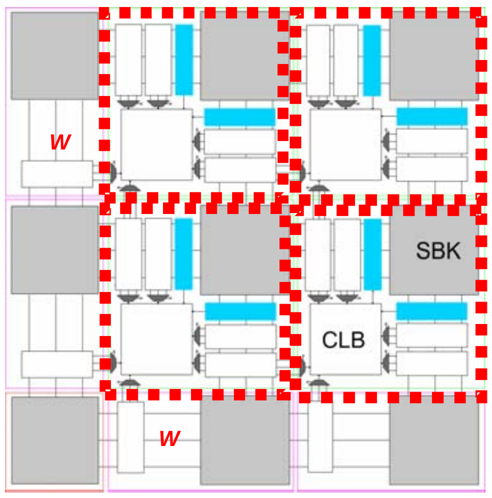
In this work, two key areas were identified to improve Soft eFPGAs. First, circuit level optimizations are used to reduce the area, power and delay overhead that results from the widespread use of ASIC standard cells. Second, a structured layout strategy is used whereby a single eFPGA tile layout is created using the ASIC flow and then replicated to form the fabric. We define the improved approach which embodies both these two features as the *Soft++ eFPGA* approach.

### A. The eFPGA Architecture

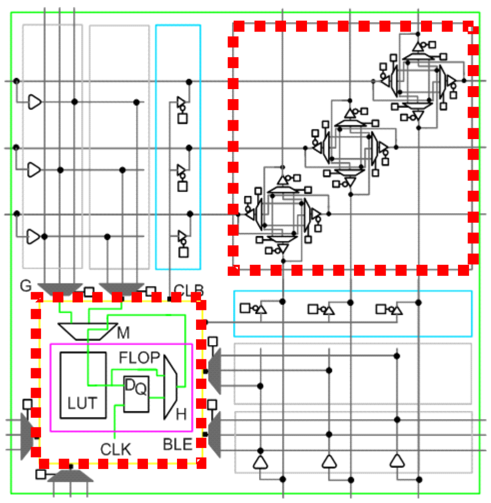
To implement the improvements listed above, it was necessary to select an architecture that is amenable to tactical cells and a structured layout. In prior work [7] [8][16], it was noted that the ASIC tools had some difficulty with the extremely large number of combinational loops that naturally exist in FPGA architectures. To avoid these problems, new *directional* architectures [7][16] were introduced to prevent such loops from ever occurring. In these architectures, signals could only travel in one direction with absolutely no chance for feedback loops. As a result of missing loop-back paths, these architectures were unable to implement sequential circuits. However, these loops are artificial; they only exist if the FPGA contains an *invalid program*. Once a valid programming configuration is loaded, these loops should not exist.

Our recent work [20] has shown that it is possible to overcome the difficulties of using ASIC tools with architectures that contain combinational loops. This allowed us to revert back to the popular island-style FPGA architecture and still use the standard ASIC flow for eFPGA generation. The freedom to implement any architecture in an eFPGA is significant: it allows one to choose the ***most efficient architecture available*** for the intended application or applications without any constraints. For example, it provides the ability to leverage the numerous architectural studies and CAD tools available for island-style FPGAs, to customize and depopulate these architectures as needed, or to invent completely new ones which are highly tuned to the application.

Figure 2(a) shows the island-style architecture [9] as a 2 × 2 array of tiles surrounded by a set of ***W*** vertical and ***W*** horizontal routing tracks. The 4 tiles highlighted in Figure 2(a) are comprised of a configurable logic block (CLB) and a switch block (SBK). Figure 2(b) shows logic and routing details of one such tile. The highlighted rectangular regions in Fig. 2(b) are the CLB and SBK, respectively.



**(a) Portion of eFPGA fabric**



**(b) Single eFPGA tile**

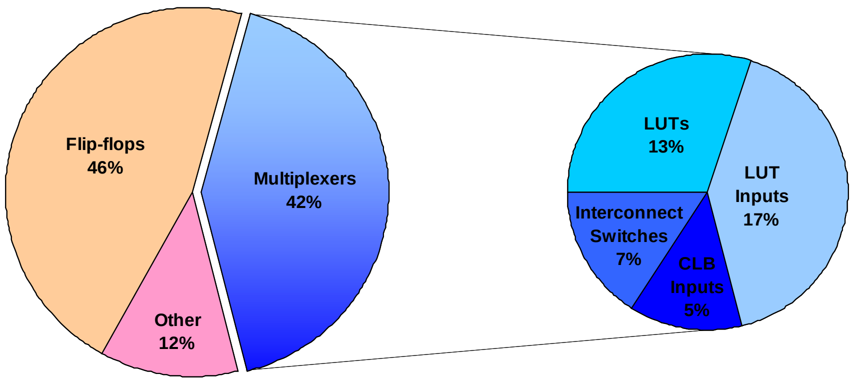
**Figure 2: A 2 × 2 island-style eFPGA**

Each tile also contains one buffer per track, called *track buffers*, to drive signals from the switch block into two adjacent CLBs (above and below, or left and right) via the input multiplexers labeled *G* in Figure 2(b). Within the CLB are basic logic elements (BLEs). The BLEs are mainly comprised of a LUT, a flip-flop and output multiplexer *H*. Multiplexer *M* is used to select which CLB input or BLE output (output of *H*) drives a LUT input. Some key parameters for this architecture are: array size ($\boldsymbol{D} \times \boldsymbol{D}$), LUT input size $\boldsymbol{K}$, the number of LUTs per CLB $\boldsymbol{N}$, and routing channel width $\boldsymbol{W}$.

The type of architecture and parameters just described are easily modeled in the VPR toolset developed for FPGA research [10]. The tools accept as input a *user circuit* and an *architecture file*. The user circuit is technology-mapped into $\boldsymbol{K}$-input LUTs, packed into CLBs containing $\boldsymbol{N}$ LUTs, placed on a $\boldsymbol{D} \times \boldsymbol{D}$ array of CLBs, and then routed. The architecture file includes the above parameters, plus additional process-related information used to compute delays. The output of VPR consists of a fully placed and routed solution, the channel width required to complete routing $\boldsymbol{W}$, the estimated layout area of the architecture, the critical-path delay of the user circuit, and a listing of the critical paths. This information can be used to select the best architecture for the user logic circuit.

## B. Suitable Locations for Tactical Cells

A tactical cell is simply a hand-crafted cell to be stored in the standard cell library and used for specialized applications. An area breakdown of a Soft eFPGA is shown in Figure 3. The largest contributors to area are configuration memory (flip-flops) and multiplexers. They comprise 88% of the total area and are not efficiently built using the Soft approach, making them ideal for tactical standard cell substitution. The remaining area of 12%, labeled “Other”, includes tristate buffers, track buffers, and switch block buffers. Figure 3(b) shows that LUTs and their input selectors dominate multiplexing area. It is also expected that optimizing generic standard cell multiplexers will have a large impact on delay, since they appear in significant numbers along most signal paths.



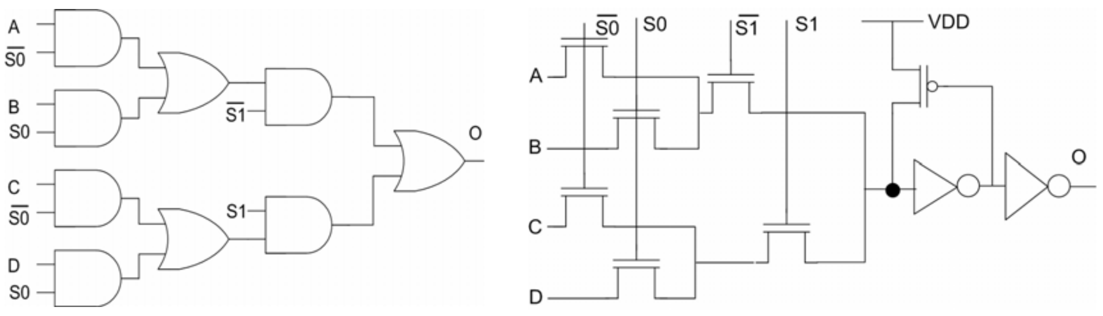
**(a) Overall area distribution** **(b) Multiplexer area distribution**

**Figure 3: Area breakdown of Soft eFPGAs**

## B.1 Details of Tactical Cell Design

Pass transistor logic is the most area-efficient way to implement large multiplexers. For example, Figure 4 shows the implementation of a 4-input multiplexer using (a) standard CMOS gates, and (b) pass transistors. The CMOS gate implementation uses 36 transistors while the other uses 11 transistors. Hence, there is significant opportunity for area reduction. Also, pass transistors should result in lower power and delay because the total switched capacitance is reduced. This is why they are routinely used in stand-alone FPGAs.

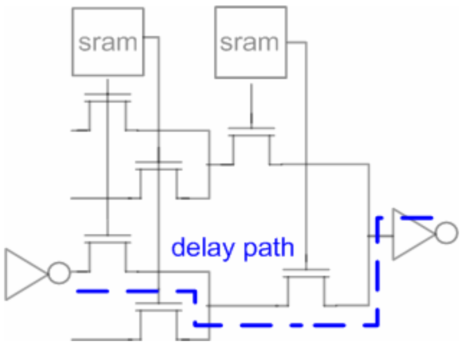
We have designed and implemented pass tree multiplexers of various sizes. The pass transistors of Figure 4(b) were set to the minimum size for 180nm TSMC process since this provided the best area and delay trade-off. The ratio of PMOS to NMOS transistor size in the inverter at the output was set to 1:1 for minimum area. In the cases where repeaters were needed in the pass tree, we determined analytically (and with SPICE) that a PMOS to NMOS ratio of roughly 1.5 gave the best area, speed and power trade-off. We also determined, in the same manner, that repeaters are needed every 4 stages of pass transistors within the multiplexer tree. The use of pass transistors results in a degraded output voltage of Vdd – $V_T$. The reduced noise margins make the multiplexer cell slightly more susceptible to crosstalk or power supply noise, but these can be avoided by careful cell design, *e.g.*, by keeping multiplexer signals in metal 1 and adding shielding, if necessary. The level restoring PMOS transistor in Figure 4(b) is used to ensure a strong high input (Vdd) to the output driver. A weak PMOS device of minimum width and twice the minimum length (to prevent node contention) was used.

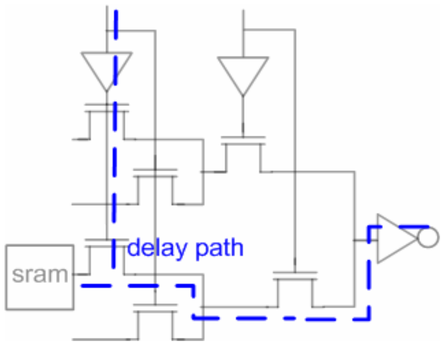


**(a) Static CMOS logic gates** **(b) Pass transistor logic**

**Figure 4: Comparison of 4:1 multiplexer implementations**

Multiplexers are used in two different ways in FPGAs. For the majority of multiplexers (such as *G* shown earlier in Figure 2(b)), the select inputs do not need to switch in functional mode because they are controlled by program bits (SRAMs). These multiplexers, shown in Figure 5(a), do not need large buffers on the select inputs for low delay. However, the select lines of the multiplexer used to implement a LUT *do* switch in functional mode and *are* timing-critical. For this type of multiplexer, shown in Figure 5(b), buffers are needed to ensure that these inputs switch quickly. We used logical effort [19][20] and SPICE simulations to size these buffers appropriately.





**(a) "Regular" multiplexer** **(b) LUT multiplexer**

**Figure 5: Critical delay paths for the different multiplexing circuits in an eFPGA**

To reduce flip-flop area, we use standard 6-transistor SRAM cells for eFPGA program memory. In FPGAs, SRAMs are unidirectional in that distinct "ports" are used for reading and writing to a cell. This is shown in Figure 6, where the input couples through a pass transistor to store a new bit value via the write port. The complement value is stored at *bitb* and serves as input to the feedback inverter named *charge keeper*. The *sense inverter* is larger since it is the input driver in the LUT function table.

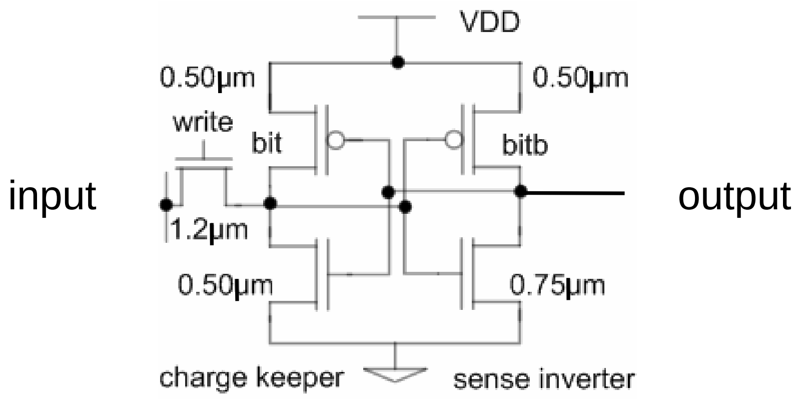
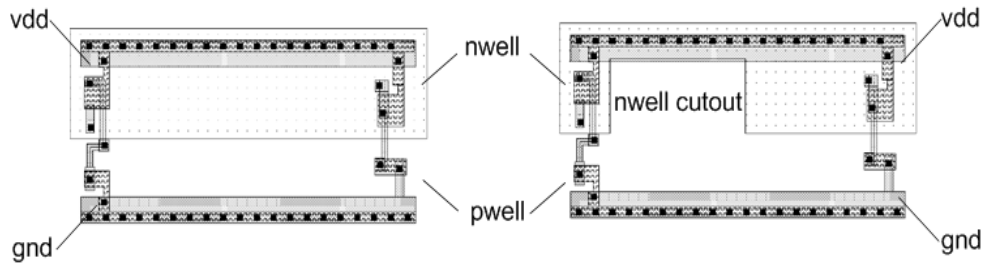


**Figure 6: SRAM cell transistor sizing for embedded FPGA program memory**

### B.2 Details of Circuit Layout

Multiplexers built from NMOS pass transistor logic require very few PMOS transistors. However, in typical standard cell layouts, NMOS transistors are constrained to the lower half of the cell and PMOS transistors to the top half. Since the PMOS area would be mostly empty, we create larger p-well regions by reducing the n-well regions in the upper half of the layout template. This is illustrated in Figure 7.



**(a) Before n-well cutout** **(b) After n-well cutout**

**Figure 7: Standard cell layout structure before and after n-well cutout region**

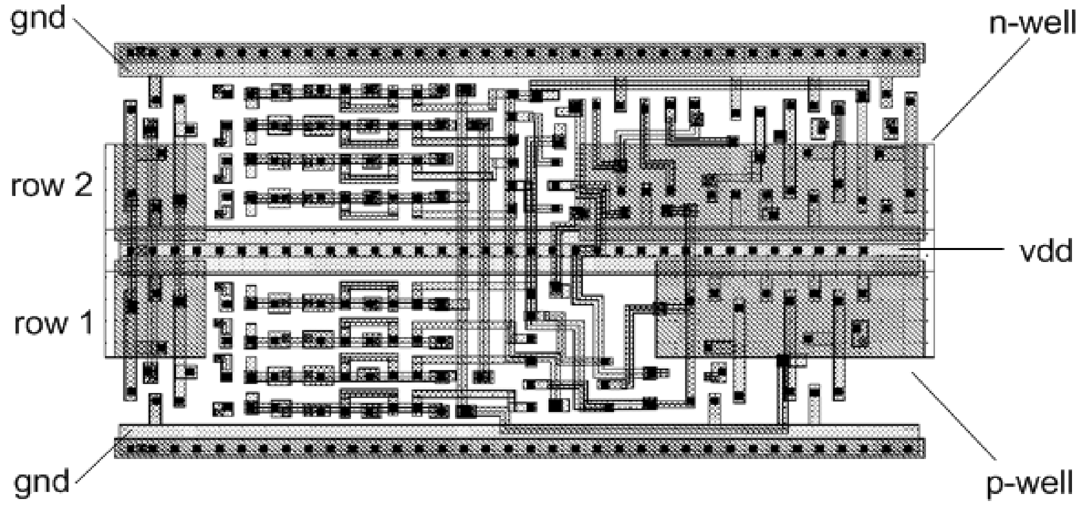
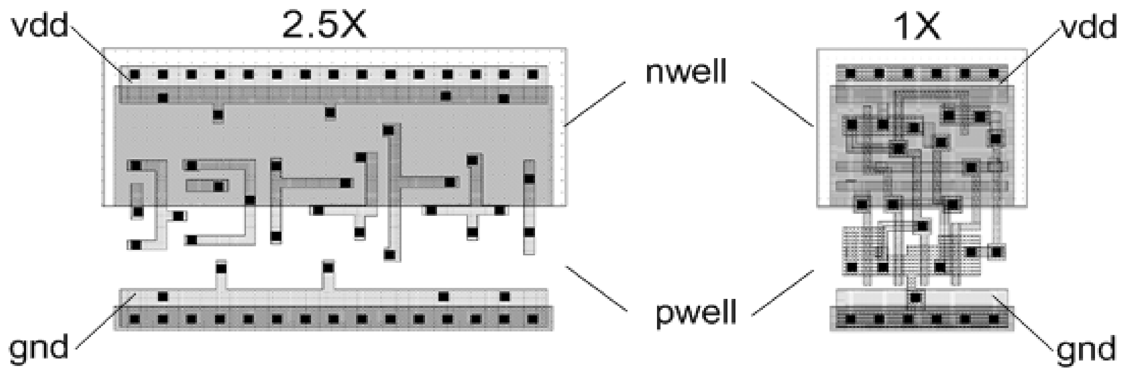


**Figure 8: Double height standard cell layout of a 32:1 multiplexer (2 metal layers)**

The resulting area cut-out from the n-well is added to the p-well area, making room for many more NMOS transistors. A small n-well region is still retained for buffer implementation and level-restoring transistors. In addition, the n-well region is left along the edges as a guard-band to ensure that layout design rules are satisfied with neighboring standard cells [20]. An analysis showed that this n-well cutout approach saves an additional 25% in area for multiplexers with 16 or fewer inputs. For larger multiplexers, we obtained comparable savings by combining this approach with double-height cells, as shown in Figure 8.

The layout template of a flip-flop and a detailed SRAM layout are shown in Figure 9. An area improvement factor of 2.5X was obtained for this case. For our SRAM layouts, we did not employ any special SRAM-specific design rules to further reduce area, but that is certainly possible. For compatibility, we also strictly followed our commercial library rules regarding pin placement, and the width of power and ground rails, etc.



**(a) Flip-flop (routing only)** **(b) SRAM (Detailed)**

**Figure 9: Standard cell flip-flop compared to a tactical cell SRAM**

Table I compares the layouts of our tactical standard cells with functionally equivalent structures created from the standard cell library. As mentioned above, the layout of a flip-flop from the library is 2.5X larger than our tactical SRAM cell. An area savings of 3.5X to 7.6X was also achieved for pass-transistor multiplexers. In this table, the LUT area includes SRAM bits and the multiplexer together, for an overall area improvement close to 4X. Delay improvements will be presented later in the context of benchmark circuits.

**Table I: Layout area and delay savings with tactical cells vs. standard cells**

| Cell | Total Standard Cell Area ($\mu m^2$) | Tactical Cell Area ($\mu m^2$) | Area Improvement Factor |
|---|---|---|---|
| SRAM bit | 61 | 24 | 2.5 |
| 8:1 MUX | 267 | 77 | 3.5 |
| 16:1 MUX | 899 | 146 | 6.1 |
| 32:1 MUX | 2230 | 293 | 7.6 |
| 4-LUT | 1880 | 530 | 3.5 |
| 5-LUT | 4180 | 1060 | 3.9 |

## C. Structured Layout Approach

In previous work [7][16], it was assumed that we could forego the benefits of a structured eFPGA design approach for greater architectural and layout flexibility. Consequently, a tile-based structure was not imposed on the specially-developed directional architectures for Soft eFPGA. As a result, a mismatch existed between the *unstructured physical layout* obtained from the ASIC back-end tools and the *structured architectural model* assumed by the FPGA CAD tools. Because the physical layout dictates the actual RC characteristics of eFPGA resources, it is important for the FPGA CAD tools to accurately model this internally during placement and routing. Moreover, if assumptions made by FPGA CAD tools about the RC characteristics of one resource relative to another do not correspond to the actual layout, the tools may inadvertently make suboptimal decisions during resource allocation. This could increase the path delays and impact timing negatively.

The second issue concerns ASIC CAD tool runtime. With our previous unstructured Soft eFPGA methodology, we found that the runtime of ASIC tools increased significantly as the size of the fabric grew. This happens because eFPGAs (especially those implemented with generic standard cells) contain a significantly larger number of nets and logic gates compared to a typical

ASIC of the same size. Further, an unprogrammed eFPGA contains an extremely large number of uncommitted timing paths (with loops!) that the tools must consider. This unduly stresses the tools and causes them to run out of memory rather quickly at certain points in the ASIC flow. For example, using a 32-bit version of the CAD tools and a compute server with 5 GB of memory, we could only carry out logic synthesis on circuits having a logic capacity of only 400 equivalent ASIC gates using the Soft eFPGA approach.

We resolved these issues in the Soft++ eFPGA design flow, as shown in Figure 10, by leveraging the regularity and structure that accompanies the island-style architecture. There are two phases in the flow: the first phase, shown in Figure 10(a), generates the *architectural parameters* for the fabric while the second phase, in Figure 10(b), generates the *physical layout* of the fabric. Architecture parameter generation process starts with the RTL description of the application circuit(s) to be implemented and uses the VPR flow to estimate area and delay for various ***N, K, W*** and ***D*** of the fabric. VPR's internal area and delay calculations are known for their good fidelity at ranking these parameter settings [10]. These parameters are passed to the template of the RTL architectural description of the eFPGA tile. The next step is to carry out logic synthesis on a single tile, which can easily be handled by the tools. To size the gates in each tile, the timing goals are iteratively tightened in a loop until the fastest results are obtained. The tile netlist is now ready for the second phase.

In the second phase, the top level netlist is constructed using a $\boldsymbol{D} \times \boldsymbol{D}$ array of such tiles. This array is placed as a structured layout by the floorplanner, followed by the standard steps of clock tree routing and power routing. When the final layout is produced for each tile, care is taken at the tile boundaries so that the array can be constructed by abutting one tile to another, both horizontally and vertically (see Figure 2(a)). The last step is to generate the GDS-II layout for the entire array.

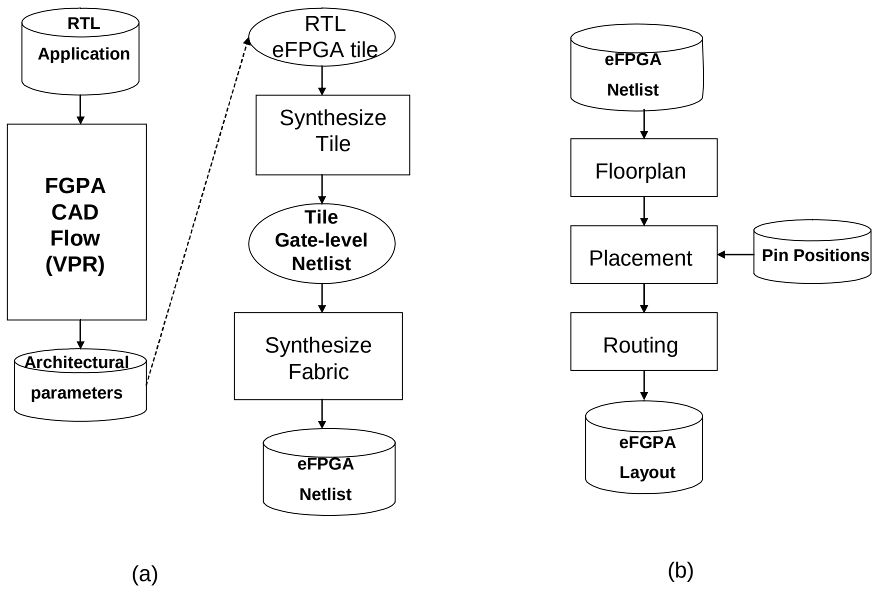
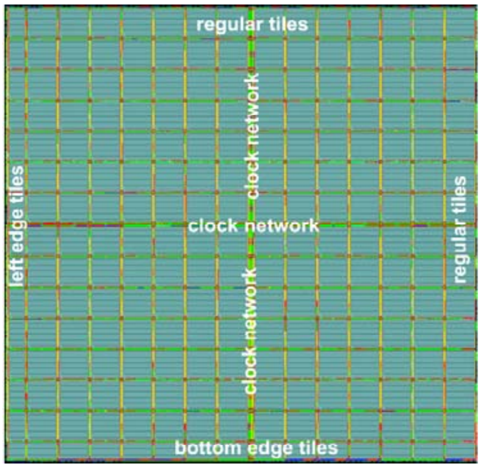


**Figure 10: A Soft++ eFPGA flow based on a structured physical design strategy**

The result of layout synthesis of a 14 × 14 array using the Soft++ flow is shown in Figure 11. Each tile in the array has an identical layout, except for the edge tiles. This is in stark contrast to the original Soft flow where one tile can be scattered over a region of the layout. One such fragmented tile is shown in Figure 12 in the shaded area (light). Clearly the timing of such a tile is difficult to predict in advance and may lead to longer delays. In fact, we observed an average critical-path delay savings of 7% across a subset of our benchmark circuits “programmed” into the fabric due to the structured layout strategy of Figure 11 over the unstructured layout of Figure 12. In some benchmarks, the location of the critical path was the same in both fabrics, implying the delay improvement was entirely due to physical layout differences.



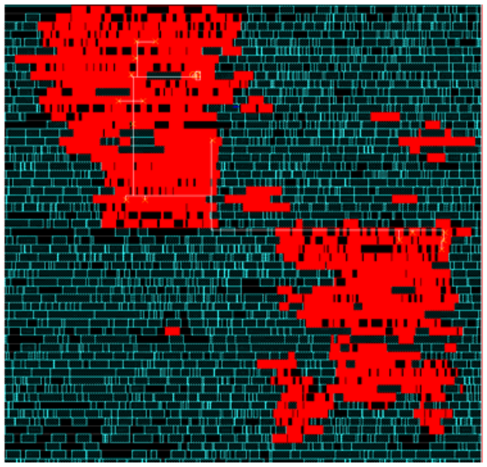


**Figure 11 : Structured layout (14x14)** **Figure 12 : Unstructured layout (1 tile)**

Compared to the flat synthesis approach used in the Soft flow, our “bottom-up” hierarchical approach also increases the capacity of fabrics that can be synthesized by at least an order of magnitude (from 400 equivalent ASIC gates to over 4000). This is possible because synthesis is divided into more manageable portions, which ultimately cuts the demand on memory and run-time. The structured layout approach used in the Soft++ flow also reduced CAD runtime significantly for physical design. For example, the layout generation for the 14 × 14 array shown in Figure 11 took roughly 3 hours. The layout of the same circuit required approximately 6X longer (17.5 hours) to complete using the original Soft flow. Runtime results were measured on a Sun Blade 1000 containing a 900MHz UltraSPARC-III processor and 5GB of memory.

## IV. EXPERIMENTAL RESULTS

In this section, the area and delay improvements from the use of structure and tactical cells is presented across several benchmark logic circuits. We determined a suitable eFPGA architecture for each one in VPR and processed the architecture through the Soft++ eFPGA design flow. For the eFPGA designs, the 180nm (TSMC) node was used to perform comparisons, but the results were also validated using STMicroelectronics 90nm technology [22].

In total, we used 17 benchmark circuits. First, to compare against the Soft eFPGA and custom flows, we used the 9 MCNC [18] benchmarks shown in Table I. Seven of these nine circuits range in size from 56 to 200 4-LUTs while the two largest ones, ex5p and apex4, are comprised of 1112 and 1340 4-LUTs, respectively. Second, to compare against a Hard eFPGA library, we added several more circuits that were slightly larger, including two encryption circuits (Rijndael and Twofish) from [12], five circuits from OpenCores.org [21], and FHK, a proprietary circuit from a Bluetooth core [1].

## A. Soft++ versus Soft and Custom eFPGAs

To evaluate the impact of our tactical cells on area, we first implemented an island-style fabric using the Soft eFPGA flow and generic standard cells, and monitored the total layout area. The area contribution of each cell and/or cell group was calculated. Next, we replaced these cells with their tactical standard cell equivalents and then recalculated the eFPGA area based on these substitutions. This produced the area required to implement the same circuit using the Soft++ eFPGA flow.

A per-circuit breakdown of results is given in Figure 13. The results in this figure are normalized relative to the area of a *Custom eFPGA* that implements the same architecture. The custom areas were obtained from VPR, which produces area estimates in units of number of minimum-sized transistor area equivalents. We obtained a physical area by multiplying this number with the size of a minimum-size transistor in our target technology; this produces an extremely optimistic layout area estimate for Custom eFPGAs since it assumes transistors are placed as densely as possible with no “layout overhead”. The Soft++ flow provides an average area reduction of 58% (2.4X improvement), although it is still roughly 2.8X more expensive than the custom layout approach.

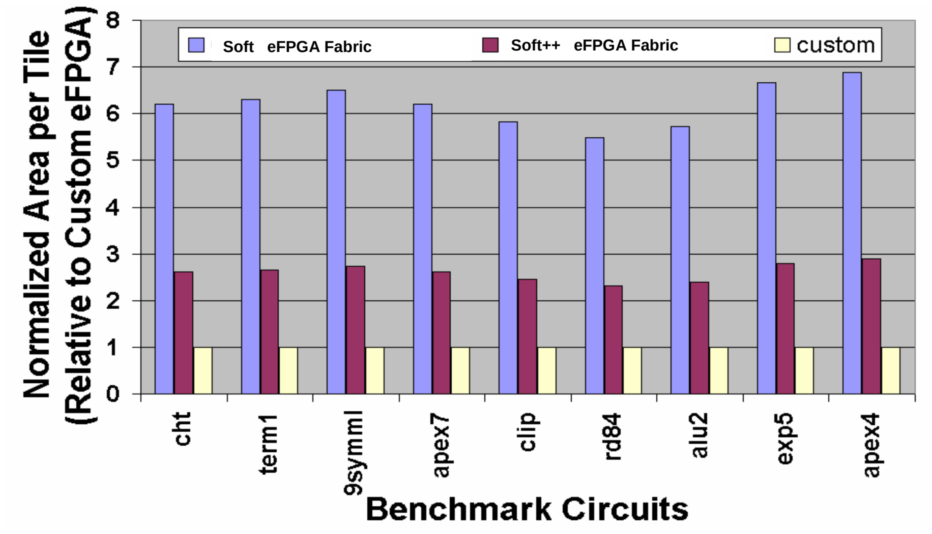


**Figure 13: Area comparison of Soft, Soft++ and Custom eFPGAs**

There are ways to further close the gap between Custom and Soft++. For example, the SRAM cell used in our Soft++ flow uses standard logic design rules and requires a layout area of 24µm$^2$. Aggressive SRAM cell layout would produce single-cell sizes between 9.7µm$^2$ [19] and 5.7µm$^2$ [9], or 2.5X to 4X smaller than ours. Therefore, if we assume a 2X improvement to our SRAM cell layouts and carefully compute area, the Soft++ eFPGA improves to 1.6X larger than full-custom layouts. Furthermore, our experience has suggested the VPR area estimates are somewhat optimistic so Soft++ may be competitive with custom if a detailed comparison is carried out.

We should also note that enormous manual layout efforts are required to produce a Custom or Hard eFPGA, making them less attractive. On the other hand, our proposed approach makes it possible to use standard ASIC tools to automate most of the flow while producing reasonably efficient results. In Section IV.B, we will present results showing that Soft++ fabrics can be more efficient than Hard eFPGAs due to a limited selection of available Hard eFPGA cores.

**T**o measure critical path delay, netlists for Soft and Soft++ eFPGAs were "programmed" using bitstreams computed from VPR. This programming information was applied in PrimeTime using timing path exceptions. Detailed analysis has shown that the critical path locations reported by PrimeTime after programming are often identical to VPR timing reports [20]. Furthermore, the critical path delays were also very similar, despite VPR assuming a custom eFPGA layout.

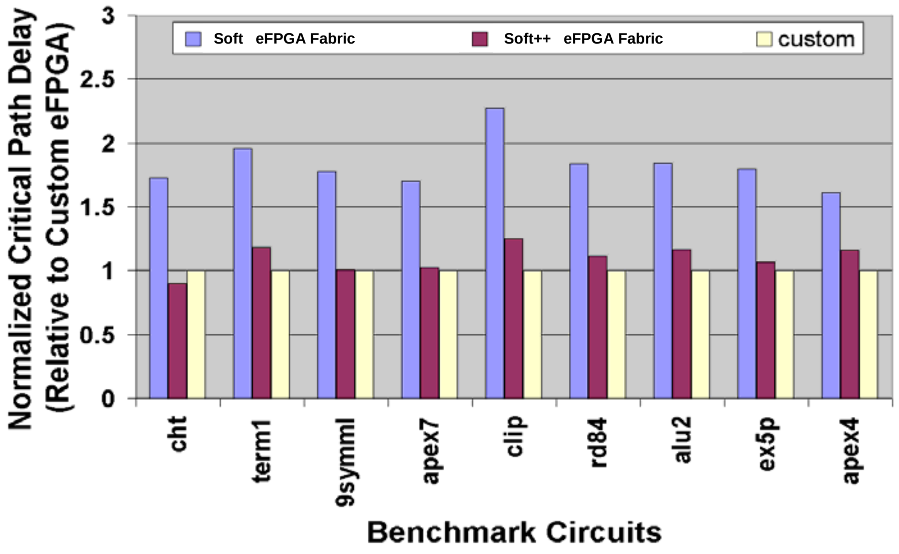


**Figure 14: Delay comparison of Soft, Soft++ and full-custom eFPGAs**

Figure 14 provides a comparison of the three approaches. It shows an average delay reduction of 40% due to the use of tactical standard cells alone. Combined with the savings from structured layout, this result places Soft++ eFPGA delays within 7% of full-custom. This is a significant result as it demonstrates that the ASIC flow can be used to generate programmable fabrics that are competitive with custom designed fabrics.

## B. Soft++ versus Hard eFPGAs

While the area penalty of Soft++ fabrics is still troublesome, we now demonstrate that they can be more area efficient than Hard eFPGAs depending on the application and the Hard eFPGA library. We use a representative Hard eFPGA library shown in Table II, which is based on data in [14] and [15]. The first column provides a label that will be used to reference the corresponding hard IP core. The remaining columns list the attributes of each core.

**Table II: Available Hard eFPGA library in 180nm technology**

| eFPGA Device | Fabric Parameters N, K, Size | Equivalent ASIC Gates | Equivalent FPGA 4-LUTs | Available I/O |
|---|---|---|---|---|
| V18L2X1 | 4, 4, 2 x 1 | 5,000 | 512 | 448 |
| V18L2X2 | 4, 4, 2 x 2 | 10,000 | 1024 | 640 |
| V18L4X1 | 4, 4, 4 x 1 | 10,000 | 1024 | 704 |
| V18L4X2 | 4, 4, 4 x 2 | 20,000 | 2048 | 896 |
| V18L4X4 | 4, 4, 4 X 4 | 40,000 | 4096 | 1280 |

For the selected benchmarks to follow, we first used the VPR flow to obtain the number of CLBs needed to implement each circuit in an island-style eFPGA with ***N***=4 (cluster size) and ***K***=4 (LUT size). Based on the number of CLBs required by VPR to implement each application circuit (usually less than the total number of CLBs), we selected the ***smallest*** hard eFPGA in Table II that would accommodate each circuit by itself.

In the Soft++ approach, it is possible to determine *a priori* the best eFPGA values for parameters ***N***, ***K*** and ***D*** for a target application. Using these parameters and the Soft++ ASIC flow in Figure 10, we generated customized eFPGA fabrics using our tactical standard cells. This fabric size was compared against the size selected for the hard core implementation. The results are shown in Table III.

The first column lists each of the benchmark circuits. The second column shows that the best set of architectural parameters (using the area-delay product as a metric) varies depending on the application. The next column provides the area of the Soft++ fabric that fits the benchmark. Next, the smallest *hard* core that accommodates the same benchmark is provided using the labels from Table II. In the last column, we compute the area ratio of the generated Soft++ eFPGA fabric and the most suitable Hard eFPGA core.

**Table III: Area comparison of our Soft++ approach versus a Hard eFPGA library**

| Application Circuits | "Best" Soft++ N, K, DxD | Soft++ 180nm Core Area ($um^2$) | "Best" 180nm Hard eFPGA fromTable II | Soft++ Area / Hard Area |
|---|---|---|---|---|
| FHK | 5, 5, 6x6 | 1.74E+06 | V18L2X1 | 0.3 |
| Cordic CLA | 10, 4, 7x7 | 4.48E+06 | V18L2X1 | 0.7 |
| $I^2C$ master | 8, 5, 7x7 | 4.34E+06 | V18L2X1 | 0.7 |
| Cordic RCA | 4,4,12x12 | 4.64E+06 | V18L2X2 | 0.4 |
| UART | 7, 5,14x14 | 1.73E+07 | V18L4X2 | 0.8 |
| SPI | 6, 4,18x18 | 2.37E+07 | V18L4X2 | 1.1 |
| Twofish | 2, 5, 30x30 | 2.58E+07 | N/A | 0.4 |
| Rijndael | 2, 5, 33x33 | 2.05E+07 | V18L4X4 | 0.7 |

Based on this analysis, the Soft++ approach outperforms the Hard eFPGA approach on all but one circuit, namely, the Serial Peripheral Interface (SPI). This is an important result since it provides evidence that Soft++ cores can indeed be as efficient as hard cores and often even more efficient, if the application and the limited hard core selection are taken into account. Note that there was no allowance given in the Soft++ results for further increases in size of the application circuits. In practice, a margin of safety would be added to the fabric for future growth. On the other hand, none of the Hard eFPGAs in the library were large enough to accommodate the "twofish" encryption circuit, even though it contains far fewer than 40,000 equivalent ASIC gates (the capacity of the largest Hard eFPGA core reported in Table II). In this case, a larger core is needed from the vendor. However, our approach was able to handle this benchmark quite easily.

## V. CONCLUSIONS

To summarize, the improved Soft++ approach uses a structured layout, tactical cells and an island-style architecture to provide significant improvements over the original Soft approach. An average area improvement of 2.4X and average delay improvement of 1.6X were obtained on a set of benchmark circuits (using the same architecture). The Soft++ approach described here makes practical the synthesis of eFPGAs that are at least an order of magnitude larger in capacity than the original Soft eFPGA approach. These Soft++ eFPGAs were shown to be within a factor of 3 in layout area to a Custom eFPGA, and had comparable delays. Furthermore, the final Soft++ fabrics were shown to be competitive with the Hard eFPGA cores available in industry in terms of area.  While it may be possible to make these fabrics even more efficient with a custom CAD flow, we believe that it is important to maintain consistency with standard ASIC flows in use today. Since the Soft++ approach relies heavily on existing FGPA and ASIC design flows, it makes the use of programmable logic much more attractive for future SoC designs, especially as designs scale below 90nm CMOS technology.

## ACKNOWLEDGEMENTS

Funding was provided by PMC-Sierra and Natural Sciences and Engineering Research Council of Canada (Grant No. STPGP 257684).  The Canadian Microelectronics Corporation provided access to Bluetooth cores from Thales. We thank Pedram Sameni for tactical cell library layout and Steve Wilton for valuable discussions.

## BIBLIOGRAPHY

[1] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu and A. Ivanov, “System-on-Chip: Reuse and Integration”, Proceedings of IEEE, vol. 94, no. 6, June 2006, pp. 1050–1069.

[2] International Technology Roadmap for Semiconductors, http://public.itrs.net

[3] S. Wilton, R. Saleh “Programmable Logic IP Cores in SoC Design: Opportunities and Challenges”, IEEE Custom Integrated Circuits Conference, San Diego, CA., May 2001, pp. 63-66.

[4] A. Cappelli et. al “XiSystem: a XiRISC-Based SoC with a Reconfigurable IO Module”, IEEE International Solid States Circuit Conference Digest of Technical papers, Feb. 2005, pp. 196–197.

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

[6] J. Wu, V. Aken'Ova, S.J.E. Wilton, R. Saleh, ``SoC Implementation Issues for Synthesizable Embedded Programmable Logic Cores", in the IEEE Custom Integrated Circuits Conference, San Jose, CA., Sept. 2003, pp. 45–48.

[7] A. Yan, S. J. E. Wilton, "Product-Term Based Synthesizable Embedded Programmable Logic Cores", IEEE Transactions on VLSI, vol. 14, no. 5, May 2006, pp. 474–488.

[8] J. Wu “Implementation Considerations for Soft Programmable Logic Cores” Master’s (MASc) Thesis, October 2004.

[9] D. Chinnery, K. Keutzer, *Closing the Gap between ASIC and Custom,* Kluwer Academic Publishers, 2002.

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

[11] I. Kuon, J. Rose, “Measuring the Gap between FPGAs and ASICs”, in the ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, February 2006, pp. 21–30.

[12] M. Holland, S. Hauck “Automatic Creation of Domain-Specific Reconfigurable CPLDs for SoC”, International Conference on Field-Programmable Logic and Applications, August 2005.

[13] I. Kuon, A. Egier, J. Rose, “Design, Layout and Verification of an FPGA using Automated Tools”, in the ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, Feb. 2005, pp. 215–226.

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

[15] P. Zuchowski et. al “A Hybrid ASIC and FPGA Architecture” IEEE International Conference on CAD, Nov. 2002, pp. 187–194.

[16] N. Kafafi, K. Bozman, S.J.E. Wilton, “Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores”, ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, pp. 1–9, Feb 2003.

[17] V. Aken’Ova, G. Lemieux, R. Saleh, “An Improved Soft eFPGA Design and Implementation Strategy”, IEEE Custom Integrated Circuits Conference, San Jose, Sept. 2005, pp. 179–182.

[18] S. Yang, “Logic Synthesis and Optimization Benchmarks, Version 3.0”, Technical Report, Microelectronic Center of North Carolina, 1991.

[19] D. A. Hodges, H. G. Jackson and R. Saleh, *Analysis and Design of Digital Integrated Circuits*, Third Edition, McGraw-Hill, 2003.

[20] V. Aken’Ova “Bridging the Gap between Soft and Hard eFPGA Design”, Master’s (MASc) Thesis, March 2005.

[21] Opencores.org, http://www.opencores.org/browse.cgi/by_category.

[22] V. Aken’Ova, R. Saleh, “A Soft++ eFPGA Physical Design Approach with Case Studies in 180nm and 90nm”, IEEE International Symposium on VLSI, Germany, March 2006.