Totem: Domain-Specific Reconfigurable Logic

Scott Hauck[1], Katherine Compton[2], Ken Eguro[1], Mark Holland[1], Shawn Phillips[3], Akshay Sharma[1]

1 University of Washington, Dept. of EE, Seattle, WA, {hauck, eguro, mholland, sharma}@ee.washington.edu

2 University of Wisconsin – Madison, Dept. of ECE, Madison, WI, kati@engr.wisc.edu

3 Annapolis Microsystems, Inc., Annapolis, MD, sphillips@annapmicro.com

Abstract

FPGAs have an important role to play in System-on-a-Chip (SoC) designs by providing programmable hardware resources within complex ASIC designs. However, because the FPGA logic is custom-fabricated with the overall SoC, we have an opportunity to optimize that logic to the SoC's specific needs. This gave rise to the Totem Project, which automates the creation of domain-specific reconfigurable logic. In this paper we present the lessons learned from the Totem Project, including how best to create domain-specific architectures, how to instantiate that logic into silicon, and how to create CAD tools to support these architectures. We also quantify how much improvement these optimizations provide over standard cells and tile-based FPGA logic. Finally, we consider the role of flexibility in domain-specific reconfigurable logic, and present strategies on how best to provide the right amount of flexibility.

Introduction

Reconfigurable logic devices, in the form of FPGAs, PALs, or CPLDs, are a powerful tool in the digital designer's toolbox. With prefabricated yet electrically configurable logic, these devices provide an ideal prototyping medium and a cost-effective solution for low to medium volume systems. Their flexibility and reprogrammability enable bug fixes, functionality upgrades, and even run-time reconfiguration techniques. Also, as their capacity and performance have increased, reconfigurable devices have become capable of supporting entire complex systems in a single device.

With Systems-on-a-Chip (SoCs) integrating multiple disparate functionalities onto a single piece of silicon, it is natural to question the future of reconfigurable logic. Proponents of field-programmable Systems-on-a-Chip argue that the increasing capacity of FPGAs, coupled with the increasing costs of custom fabrication, point to the use of large FPGAs to implement entire systems. However, there will always be applications with performance, power, density, or other requirements that simply cannot be supported in a commodity FPGA. These systems will require custom fabrication, yet can still benefit from reserving some portion of the chip for reconfigurable logic. Although perhaps 90% of the silicon area may be fixed logic, microprocessors, memories, or other standard logic components, inserting reconfigurable logic into the remaining 10% of the chip can provide the benefits of reconfigurability to SoC designers. Bug fixes, in-the-field upgrades, run-time-reconfiguration, and other considerations are even more applicable in SoC than they have been in the System-on-a-Board methodology.

The obvious solution is to directly use existing stand-alone reconfigurable logic structures inside SoC designs. The basic tiles of an FPGA, PAL, or CPLD can be provided as a hard or soft macro to the chip designer, and directly fabricated into the SoC silicon. For example, Xilinx provides some versions of their higher-end FPGA cores to IBM, which has embedded Xilinx reconfigurable blocks in some of its ASIC designs [Xilinx04]. Actel has created a generic FPGA fabric (called an embedded programmable gate array, or EPGA) that can be embedded into an SoC [Actel04]. The size of these VariCore EPGAs in ASIC equivalent gate densities range from 5K to 40K.

Although translating standard commodity reconfigurable logic into IP could be an easy way to provide reconfigurability in SoC, it is not clear that the requirements of SoC systems and those of stand-alone reconfigurable logic are the same. For example, while a commodity chip may need to support a wide variety of applications, an individual SoC might need reconfigurable logic only to support DSP, control logic, or some other style of circuit. Thus, if the reconfigurable logic were optimized to the required domain, a significantly higher-quality system might be possible. There are several companies that are pursuing this option. For example, LSI has created LiquidLogic [LSI04], composed of reconfigurable macro cells designed to be embedded into SoCs, as well as an input-output bank that interfaces with other ASIC circuitry on the SoC. The smallest reconfigurable unit in the architecture is a 4-

bit ALU. Elixent offers their D-Fabrix, which is also based on a sea-of-ALUs [Elixent03]. In the academic world, Wilton's group has developed directional architectures that work well in standard cell flows [Kafafi03, Yan03].

Unfortunately, it is not possible to hand-tune an architecture for every possible situation. First, the number of possible application domains is quite large. Also, a single reconfigurable subsystem may be required to implement circuits from multiple circuit domains, greatly increasing the number of unique architectures needed. Finally, hand-customizing the hardware for each domain or specific SoC would increase design costs dramatically, sabotaging the value of domain-specific reconfigurable cores.

In 1999 our research group asked a simple question – what if we created a system to automatically optimize reconfigurable logic for different SoC designs? Can we significantly improve the quality of these systems, and if so, how?   These observations led to the Totem Project, an effort to automate the creation of domain-specific reconfigurable logic. After 6 years and 6 theses, we can begin to answer these fundamental questions. This paper represents the results of our efforts, focusing on the large-scale implications. In this paper, instead of concentrating on the individual techniques and research approaches, we instead hope to show what, in our view, is crucial in the realm of domain-specific reconfigurable logic.

Before we discuss the research results, we first present the overall Totem flow, breaking the domain-specific FPGA generation task into three major components: Architecture Generation, Layout Generation, and P&R Tool Generation. We then cover our two major testbeds – RaPiD arrays, and CPLDs. Finally, we conclude with an in-depth discussion of the ramifications and lessons of the Totem project. We hope that this paper provides guidance on how best to employ FPGAs in upcoming SoC designs, as well as a base for future research in this important field

Totem - Creation of Domain-specific Reconfigurable Logic

The goal of the Totem Project is to provide a complete automatic path for the creation of custom reconfigurable hardware, targeted for use in Systems-on-a-Chip (SoCs). There are three primary components of the project. The first is the high-level architecture generation, which determines the resource requirements and how those resources should be arranged. It creates the description of both the logic blocks and the programmable interconnect. The second component is the VLSI layout generator, which takes a description of the architecture from the high-level architecture generator and translates it into actual transistors and layout masks. Its primary goal is to provide the most efficient implementation in terms of area, power, and/or performance. The final module is the place and route tool generator, which creates the CAD suite for the architecture. The resulting tools handle the mapping of user designs onto the programmable substrate created by the architecture and layout generators.
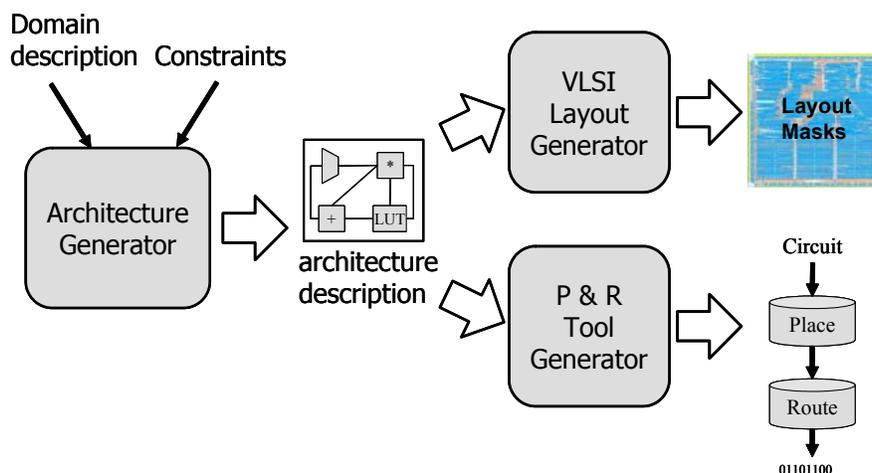

Figure 1. Totem system flow.

Totem Target Technologies

Although the generic flow from Figure 1 is applicable to the creation of almost any style of reconfigurable logic, different types of logic will require different algorithms for each of these steps. So far in the Totem project we have focused on two different architectural styles: RaPiD, a coarse-grained architecture developed for signal processing applications, and CPLDs, supporting arbitrary logic in their generic PAL/PLA blocks and crossbar interconnects. We discuss these testbeds and the techniques specific to each in the next sections.

RaPiD

The disparity between the coarse-grained nature of many computations (such as those needed for DSP), and the fine-grained nature of traditional FPGAs, leads to inefficiencies in hardware implementations. The RaPiD system [Ebeling96, Cronquist99] addresses this problem by using a very coarse-grained structure. This style of architecture has specialized computational elements such as ALUs, RAMs, and multipliers, each operating on full words of data. The components are arranged along a one-dimensional axis, and connected by word-width routing tracks. The architecture is heavily pipelined to provide very fast computations on streams of data. While the routing flexibility is somewhat lower than standard FPGAs, the routing architecture complexity is also lower, reducing routing area as well as simplifying the routing process.
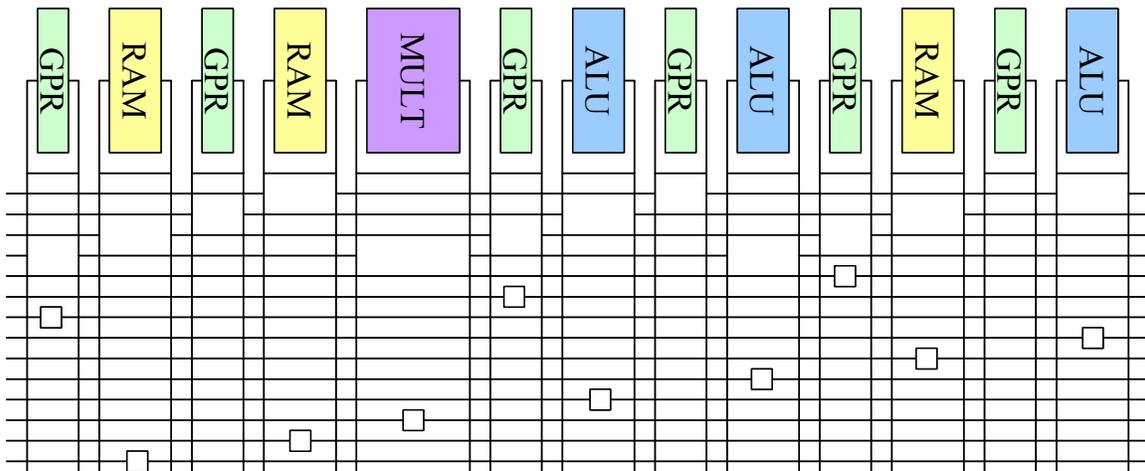

Figure 2. The RaPiD datapath.

RaPiD logic units are grouped into a basic cell, as shown in Figure 2; multiple copies can be abutted to create larger arrays. The logic units within the cells operate on full words of data, and include 16-bit ALUs, 16x16 multipliers, 16-bit wide RAM units, and 16-bit registers. Each component contains a multiplexer on each of its inputs that chooses between the signals of each routing track. Each component also has a demultiplexer on each of the outputs that allows the unit to directly output to any of the routing tracks. Inputs are on the left side of a logic unit, while the outputs are on the right side of the unit (designated by a single vertical line for each).

The routing architecture itself is a one-dimensional segmented design, where each track is composed of as many wires as the word width of the architecture (16-bits in the original implementation). Full words of data are therefore communicated between the computational units of the architecture. There are 14 routing tracks, plus one additional routing track that only contains "feedback" wires. These feedback wires are only permitted to route an output of a unit back to one or more of the inputs of the same unit. Additionally, a word-sized "zero" is also provided as a possible input to each multiplexer. The top five routing tracks are local routing tracks, including the special feedback track. These tracks contain short wires for fast short-distance communication. The bottom ten tracks provide longer distance routing. The small squares on these routing tracks are bus connectors, which allow the wire segments to be optionally connected to form longer wires. Additionally, the bus connectors provide optional pipeline registers to mitigate the delay added through the use of longer wires and routing switches.

RaPiD-style architectures are an interesting testbed for the Totem approach for multiple reasons. First, RaPiD is already a domain-specific architecture, optimized to signal processing applications. Thus, this allows us to see how much customizing an architecture to a specific user's needs improves over a manually domain-optimized architecture. Also, the 1D nature of the interconnect simplifies some of the steps in Totem, since 2D versions of problems such as layout generation are more difficult to solve than 1D versions. Finally, colleagues at U.W. had full-custom layouts of the architecture, a mapping flow, and benchmark designs to serve as a comparison for manual design and layout techniques [Ebeling96, Cronquist99].

In the Totem-RaPiD system we optimize both the interconnect and the logic [Compton01, Compton03, Compton06]. For the logic blocks, we can adjust the mix of different types of logic units [Eguro03, Eguro05a], and their placement within the overall array. Also, we have considered the introduction of a completely new set of basic logic elements optimized for private-key encryption [Eguro02]. In terms of interconnect, the optimizations determine the number and length of the short and long wires in the RaPiD interconnect, as well as the number and location of pipelining registers [Compton02, Compton03a].
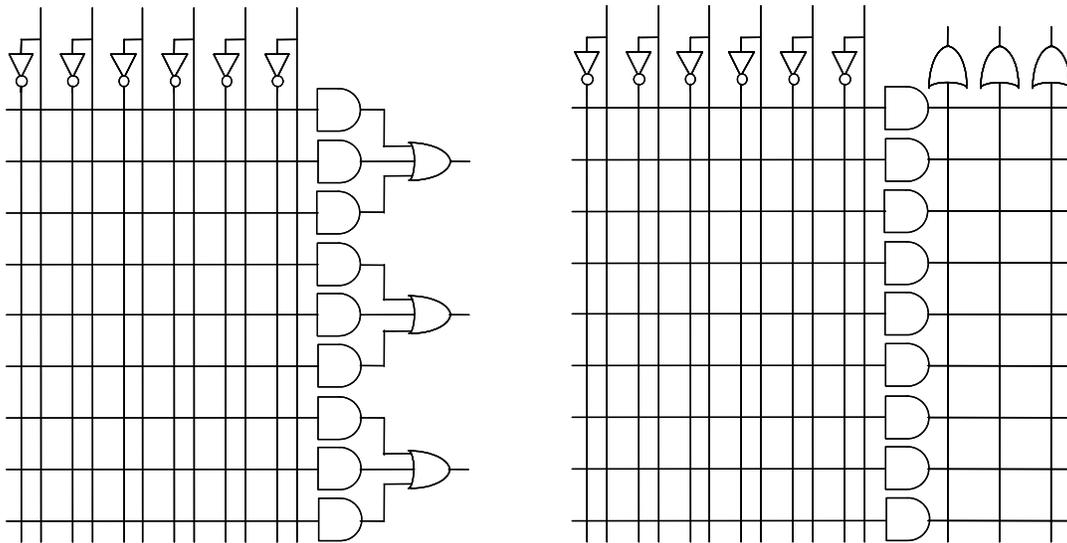

Figure 3. PAL (left) and PLA (right) structures [Biehl93].

CPLD

In SoC designs we expect to dedicate a small amount of the chip area (perhaps 10%) to reconfigurable logic. For board-level designs, the best reprogrammable implementation strategy for small logic functions is PALs, PLAs, and CPLDs. A Programmable Array Logic (PAL) is a device optimized for implementing two-level sum-of-products logic equations (Figure 3 left). The device consists of a set of arbitrary product terms (the AND gates) leading to fixed summation terms (the OR gates) that produce the chip's outputs. The product terms are logic structures that can be programmed to implement an AND of any combination of its inputs.

One limitation of a PAL is that product terms cannot be shared between outputs. Thus, if two outputs both require the product term $\overline{A}\,\overline{B}C$, they would each need to generate the function with their own product terms. A different form of PLD, called a Programmable Logic Array (PLA), allows product terms to be shared between output functions (Figure 3 right). In a PLA, the AND array of product terms (the AND plane) leads to a similar OR array (the OR plane). PLAs are characterized by their number of inputs, product terms, and outputs, shown numerically as (in-pt-out). While PLAs have more flexibility than PALs since the connections between the AND and OR gates are programmable, this flexibility results in lower performance. The performance degradation is primarily due to the fact that in a PLA a signal must travel through two programmable connections (one in the AND plane, one in the OR plane), while in a PAL the signal goes through only one programmable connection.
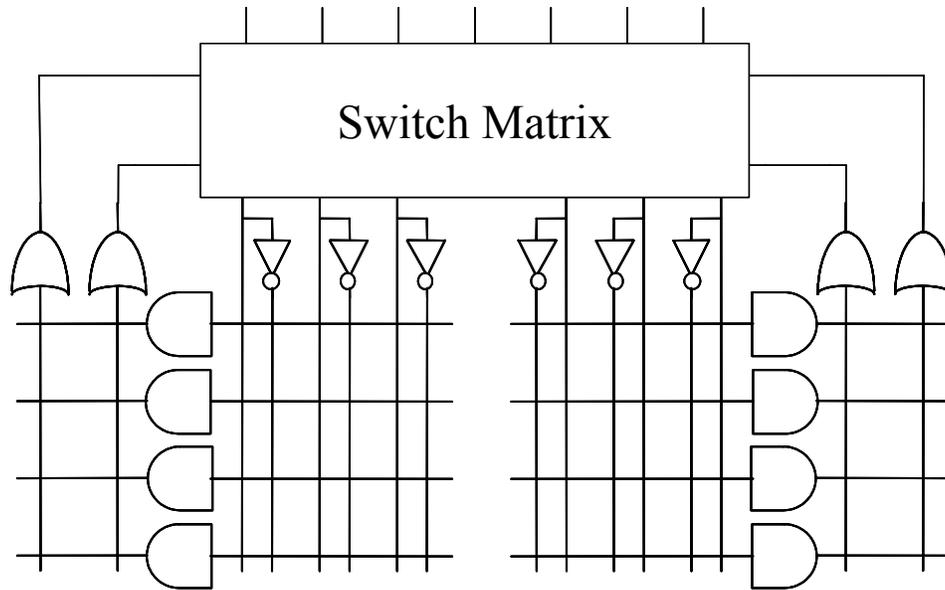
Figure 4. CPLD structure. The switch matrix provides configurable connectivity between the inputs, logic arrays, and any feedbacks from the outputs [Biehl93].

While the previous structures are fine for small PLDs, they may not scale to larger arrays. Primarily, as the PLD gets larger, the number of connections to a given product term also grows larger, slowing down the signal propagation and increasing the required area. To combat this problem, Complex Programmable Logic Devices (CPLDs) break the system up into several smaller product term arrays, connected together with a switch matrix (Figure 4). Thus, the CPLD can be viewed as an interconnection network connecting together a collection of smaller PLDs. This network can either be a full crossbar, where each output can talk to any input, or a sparse crossbar, which reduces the connection flexibility in order to improve area, power, and performance. In this way, the CPLD can have much larger capacity than a single PLD, while keeping propagation delays low.

In our investigations into Totem-CPLD, we have considered SRAM-programmable PALs, PLAs, and CPLDs [Holland05]. For the PALs and PLAs, we have created techniques for automatically removing portions of the programmable structure unneeded for a given set of applications [Holland04]. At the CPLD level, we consider adjustments to the base PLAs in the structure (by altering their number of inputs, product terms, and/or outputs), the number of PLAs, and the richness of the interconnect structure [Holland05a].

Implications of the Totem Project

Based on our efforts in the Totem project, we believe we now have a good understanding of many issues in the generation of domain-specific reconfigurable logic. In the sections that follow, we will discuss what we have learned, and the resulting implications for potential users of reconfigurable logic in SoC. We will also point out what we feel are the right areas for future research. We have chosen to structure this discussion as a set of questions and answers, concentrating on the results and implications of the Totem efforts. For those readers interested in the exact algorithms and other details for the approaches, we have also included references to specific papers, theses, and technical reports that expand on those topics. Overall, this section begins by considering domain-specific logic as a whole, then progresses through issues specific to architecture generation, layout generation, and tool generation.

1. What is a domain?

In the world of commodity IC's, the concept of a "domain" is fairly straightforward. A domain is a set of related computations that have similar features—similarities that allow a single commodity device or approach to efficiently support these computations. A domain can be somewhat broad, such as incorporating all of signal processing. In

this domain, most applications are high-bandwidth and compute-intensive in nature, and generally focus on word-width computations. However, since the domain is fairly large, a variety of resources and a flexible routing structure must still be provided. Alternatively, a domain can be more specialized, where all computations within a domain are extremely similar, or are a subset of a larger domain. To continue the previous example, a smaller domain within DSP could include only FIR filters, allowing a designer to use hardware even more optimized than something created for DSP in general.

Although initially we viewed a "domain" as discussed above, after several experiments we discovered that this was an inaccurate, or at least incomplete, view. The driving force of the domain definition is not always circuit similarity, but rather the target use of the SoC. A "domain" is dictated by the designer of the SoC, and a domain-specific subsystem is a subsystem that best supports those circuits the SoC designer plans to map to that subsystem. For one SoC, the "domain" may be "encryption protocols", if that will be the only use of the reconfigurable logic. Another SoC, perhaps one intended for use in a mobile videophone, might use the reconfigurable logic to implement the Rijndahl encryption algorithm AND a video compression engine AND a software defined radio. The truth is that "domain", as it applies to customized reconfigurable cores, is whatever the SoC designer wants and needs it to be. For restricted domains, the circuits will have many common features and the reconfigurable logic can be highly optimized, yet limited in what it can support. For more generic domains, the circuits can have few common features, and the resulting reconfigurable logic must be more flexible and thus less highly optimized.

2. *How much better is domain-specific than domain-generic?*

There are several alternatives to domain-specific logic. Circuits can be implemented directly in ASIC logic via a standard cell flow, avoiding the overhead of programmability. Or, if reconfiguration is desired, standard FPGA tiles can be included on the SoC, avoiding the need to customize the logic to a specific domain. Thus, domain-specific implementations are only useful if they offer a significant quality improvement over standard FPGA tiles, while retaining the ability to be reconfigured to handle multiple circuits.

Table 1   Area comparison of domain-specific reconfigurable logic with other implementation technologies [Compton03]. Results are area in mm^2. "fail" represents cases where the domain fails to map to the technology, and these cases are ignored for the means at the bottom.

	FPGA	RaPiD	Domain-Specific (AMO RaPiD)	Standard Cells	cASIC
Radar Processing	19.719	4.996	2.877	4.101	1.520
OFDM	59.157	fail	20.800	9.168	4.574
Digital Camera	23.006	fail	8.768	7.268	2.475
Speech Processing	78.877	79.937	37.635	26.523	13.010
Image Processing	19.719	fail	3.681	6.843	1.833
FIR	26.292	6.661	3.630	2.846	2.004
Matrix Multiplication	19.719	3.331	2.347	1.785	1.264
Sorters	26.292	4.996	3.476	1.541	1.487
Geo. Mean (AMO-normalized)	4.76	1.69	1.00	0.81	0.40
Geo. Mean (Standard Cell norm)	5.90	2.20	1.24	1.00	0.50

Our efforts on Totem-RaPiD provide insight into the area impacts of domain-specific architectures (note that we do not include delay because the limitations of current pipelining routers make the delay results somewhat suspect). As shown in Table 1, we compare the area costs of multiple domains (each consisting of many different circuits) on FPGAs, RaPiDs, and standard cells, as well as two domain-specific techniques. AMO (Add Max Once, named for the interconnect generation algorithm), is a technique for architecture generation that yields very flexible, yet area-efficient, RaPiD-style domain-specific architectures. Configurable ASIC (cASIC) produces inflexible structures reconfigurable only within the provided circuit set, akin to a custom datapath.

The first things to consider are the FPGA, RaPiD, and standard cell columns. In the standard cell process, we use very tight standard cell layouts and fixed interconnect wires. As such, it provides a very dense implementation of circuits. FPGAs, represented in this study by the Xilinx Virtex-II, have a much larger area because both their logic and routing must be programmable, though they can reuse area between different circuit mappings via

reconfiguration. In our tests the FPGA solution was 5.9x larger than standard cells, representing the large costs of complete flexibility. RaPiD represents a compromise of sorts between standard cells and FPGAs. By restricting the interconnect and logic to support specific styles of circuits, yet retaining some programmability, the areas should move closer to those of standard cells. Indeed, for those benchmarks that successfully map to RaPiD, the implementations are only 2.2x larger than standard cells, and 2.68x smaller than FPGAs. However, the table also demonstrates that this approach comes at a cost – many of the domains considered simply do not fit onto RaPiD because of RaPiD's limited interconnect. Any fixed architecture runs the risk of not meeting the requirements of a given designer's circuits.

The column "Domain-Specific" demonstrates the advantages of optimizing the reconfigurable logic for the actual targeted domain specified by the SoC designer. These implementations are significantly smaller than a fixed RaPiD, even though we use similar structures and interconnects, because we tailor the exact mix to the designer's actual circuits. In fact, we also are able to accommodate domains that RaPiD does not – our tools add logic and interconnect resources to fit the members of the domain. Using the Totem generator, we achieve a 4.76x smaller implementation than that of FPGA-based tiles, and a 1.69x smaller than even a fixed domain-specific architecture such as RaPiD, while still ensuring that it can actually handle all of the circuits the user requires.

When compared to the standard cell implementations, our domain-specific implementations pay merely a 1.24x area increase for adding programmability to their structures, instead of the 2.20x of RaPiD or the 5.90x of FPGAs. We are able to approach the efficiency of a standard-cell flow partially by restricting the flexibility to what is needed by the actual domain, and also by using well-crafted layouts of the basic elements (similar to a macro-cell approach).

Earlier we stated that cASICs are very limited in their configurability. In fact, they can only be configured between the specific circuits used to generate the hardware. Since they use our macrocells, and have direct point-to-point routing with little or no programmability, they represent an implementation close in quality to a full-custom layout. This style of circuit can be useful for cases where the exact circuits are known, and no additional flexibility is required. In this case, a cASIC can be used in place of a series of standard-cell circuits to reduce the area required. However, they fail in one of the main goals of reconfigurable subsystems for SoC – the ability to handle new circuits that are similar to, but not exactly the same, as those the SoC designer initially provided.

Table 2. Area-delay product comparison of domain-specific CPLDs with domain-generic CPLDs [Holland05, Holland05a]. Results are normalized to the domain-specific results. The logic unit sizes are given in (inputs – product terms – outputs) format at the top of the columns.

	Xilinx (36-48-16)	El Gamal (10-12-4)	PLAmap (10-20-5)	Domain-Specific	Comb (9-80-4)	Seq (18-42-8)	FP (8-18-2)	Arith (7-14-2)	Encrypt (13-46-4)
Combinational	13.88	13.63	5.27	1.00	1.00	9.46	5.37	4.26	4.04
Sequential	1.84	2.47	1.96	1.00	3.46	1.00	3.81	6.04	2.05
Floating Point	29.43	13.15	7.37	1.00	2.93	15.76	1.00	1.02	3.26
Arithmetic	40.32	41.48	14.95	1.00	7.11	33.49	1.82	1.00	6.34
Encryption	7.52	4.16	2.85	1.00	1.40	3.42	1.09	1.19	1.00
Geo. Mean	11.79	9.48	5.04	1.00	2.51	7.02	2.10	1.99	2.80

Domain-specific logic is also beneficial for CPLD-style architectures. Table 2 shows the area-delay product of mapping a set of domains to both domain-specific and domain-generic architectures. Several baselines are provided for comparison. "Xilinx" represents an architecture similar to the commercial Xilinx CoolRunner devices; "El Gamal" is a design from a 1991 academic analysis of PLA sizings in reprogrammable architectures by Kouloheris and El Gamal [Kouloheris91]; "PLAmap" is a manually-designed structure based on our own initial analysis of running several LGSynth93 circuits through the CPLD technology mapper (PLAmap [Chen01]) used in our research. The results indicate that converting from a commodity, standardized architecture to a domain-specific one represents an improvement of 5.04x to 11.79x in area-delay product—a huge improvement.

One might contend that the positive results do not reflect the high quality of our solutions, but instead a poor choice of comparative architectures. We therefore conducted an experiment specifically to test the benefit of domain-specific architectures within the CPLD framework, attempting to control for other design considerations. Circuits from one domain were implemented on architectures generated from different domains. The rightmost five columns

of Table 2 show the results of this test. The columns represent different generated domain-specific architectures. For example, mapping the Combinational domain of circuits to an architecture created for the Sequential domain results in an area-delay product 9.46x larger than if those circuits were mapped to the architecture generated for their own domain. These results show that even controlling for the way architectures are designed, results are at least 2x worse when the architecture is incorrectly optimized, and thus a mismatch in architecture to domain has a significant cost.

Overall we see a significant benefit from domain-specific logic. The benefits found so far are 4.8x in area compared to traditional FPGA tiles for Totem-RaPiD, and 5x to 11.8x in area-delay product as compared to commodity CPLDs for Totem-CPLD.

3. How do you automatically create the architecture for a domain-specific FPGA?

Architecture generation for a domain-specific FPGA is somewhat akin to high-level synthesis for an ASIC design; the tools must determine what resource mix and interconnect structure will best support the desired functionality. In an ASIC flow we are generally seeking the best datapath to support a single, predefined computation. However, in domain-specific FPGA generation, we must instead create a datapath capable of supporting multiple circuits, but only a representative sample of the circuits may be available at the time of architecture creation. Thus we must not only optimize based on the demands of the individual circuits, but also anticipate future designs.

In the Totem Project we have developed architecture generation techniques for four different situations:

1. *Creation of a complete RaPiD-style datapath for a domain from scratch, including logic block selection and placement, as well as interconnect generation.* This was solved by placement and routing of multiple user designs simultaneously within a simulated annealing framework. The resulting resource costs were directly modeled in the annealing cost function [Compton01, Compton02, Compton03, Compton06].

2. *Taking a predefined RaPiD datapath and identifying which resources can be eliminated for a given domain.* This was solved by first placing and routing circuits from the domain onto the existing datapath, and then iterating though a "reduction" process. Subsequent placements and routings involved penalties on unused or underused resources in the architecture, thus pushing individual circuit mappings to enhance commonality of resource usage. At the end, any resources unused by all input circuits can be eliminated. This is our "subtractive" system [Phillips04, Phillips04a].

3. *Generating a PAL/PLA substrate for a given domain, including eliminating individual programmable switchpoints in the AND and OR plane.* This was solved via a simulated annealing framework performing simultaneous placement of multiple user designs. The cost function was augmented to penalize resources that are used by only a few benchmarks, and thus seek to increase the number of switchpoints that could be eliminated [Holland04, Holland05].

4. *Creating a complete CPLD from scratch, with an emphasis on selection of the best PLA to use as a logic block (selection of inputs – product terms – outputs).* This was solved by making independent mapping calls to the PLAmap technology mapper [Chen01] and an area and delay estimator for the resulting CPLD. We then evaluate each candidate architecture's quality by aggregating the results across the full set of provided user designs [Holland05, Holland05a].

Although the individual techniques vary between different architecture styles and implementation strategies, we have found that the overall flow is fairly similar. To create an efficient architecture for multiple benchmarks, first start with an algorithm that can create an architecture for a single benchmark. Then augment the algorithm with a mechanism to aggregate the costs across multiple benchmarks, and use this to influence the mapping of each benchmark in the domain.

Creation of an architecture for a single benchmark is relatively straightforward, though it varies by technology. For example, the proper PAL or PLA to use for a circuit can easily be determined after logic synthesis and minimization. In a CPLD the task is somewhat more complex since there are multiple variables to consider (inputs, product terms,

and outputs of the basic PLAs, plus number of PLA blocks and sparsity of the crossbar), but simple search techniques involving multiple technology mapping calls can be performed [Holland05, Holland05a]. For a complete RaPiD datapath, we must perform placement of the individual logic units, binding of computations from the user designs to the physical units, and interconnect generation for the interconnect muxes, demuxes and wires [Compton01, Compton02, Compton03, Compton06]. These techniques are similar to standard high-level synthesis techniques [DeMicheli94]. However, as we will discuss in question 4 below, it is important that all created elements be in a relatively regular and structured pattern, or else the resulting architecture will not support circuits other than the original benchmarks.

Once a tool is available to generate an architecture for a single user design, handling multiple designs simply requires aggregating information from architecture generation for each of those designs, and using that information to guide subsequent steps in architecture generation. For example, one simple mechanism is to augment the base tool such that it performs single-benchmark architecture generation on all of the benchmarks simultaneously. Then, as a decision is made on any part of the architecture generation, the impact on all benchmarks can be measured, and the proper decision made for the overall domain.

It may not always be possible to generate an architecture for multiple user designs simultaneously. Sometimes a needed tool is available only as an executable, preventing modifications to the tool's execution; PLAmap is an example of this (scenario #4). In other situations, it may be difficult to change an algorithm to directly support multiple mappings; this was an issue in the place & route tools for RaPiD (scenario #2). In both cases, this was solved by a meta-generator that aggregated information from multiple runs of the single-circuit generator. For example, to find out the best CPLD for multiple circuits, we can simply run PLAmap on each candidate architecture individually and aggregate the results. For subtractive (scenario #2), we ran the placement and routing flow for each user design separately. Then, we identified those resources that were used by relatively few user designs, since they were the most likely candidates for elimination if the design mappings were coordinated. These resources were penalized, and the placement and routing flow re-run. Through multiple iterations of this flow, the mappings of individual circuits were coordinated across the entire domain.

4. How do you optimize for a domain, yet still keep flexibility to support new circuits?

The simplest way to generate a new domain-specific architecture is to add exactly the resources required by a given user design (Figure 5). Logic units of exactly the right operations can be placed where-ever needed, and point-to-point wires put in place to hook together just those resources that must be connected for that design. In this way one can get a reconfigurable structure very highly optimized to the circuits provided by the user; this is in fact the premise behind our cASIC approach (introduced in question 2 above).

Unfortunately, an architecture so highly optimized to only a few circuits has very little chance of supporting any other circuits. Even slight modifications or bug fixes on the circuits used to actually generate the architecture are unlikely to fit. We refer to this as the "cASIC trap". While a cASIC structure may be sufficient for some uses, and (as shown above) can achieve higher densities than non-configurable standard cell designs, it removes most of the benefits of reconfigurability.
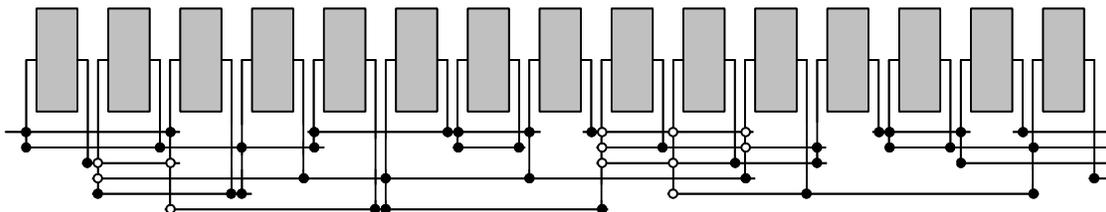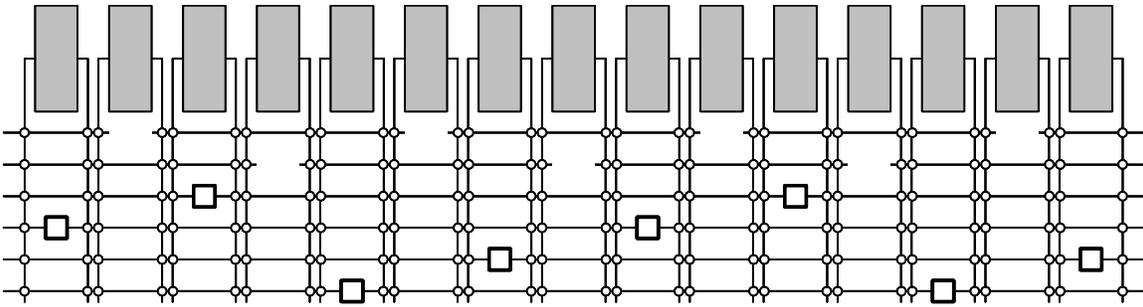

Figure 5. Example of a cASIC interconnect. Empty circles are programmable connections, filled circles are fixed connections.

Figure 6. Example of a flexible interconnect.

Instead of developing architecture generation algorithms that put down individual resources, we've found it much more effective for the algorithms to instead select resources from a set of regular, flexible patterns. For example, when developing a PLA as the basic logic unit for a CPLD, the tool is not allowed to select individual programmable switches to add or delete; the resulting structure is unlikely to support other circuits efficiently. Instead, the system can choose the number of inputs, outputs, and product terms for these basic elements. In this way the logic elements are optimized to the needs of circuits from that domain, yet are likely to be reusable for other circuits. Similarly, when creating the interconnect structure of a RaPiD device, our cASIC approach runs independent wires from a given source to a given destination, and only has muxes and demuxes for resources that must actually be connected in the input benchmarks (Figure 5). Our flexible algorithms instead deal in complete tracks, with segmented wires of a uniform length and muxes and demuxes at every logic unit they traverse (Figure 6). Thus, the flexible tools can choose the mix of short and long tracks, and even choose the length of individual tracks, but the resulting interconnect is still a flexible set of resources that can support many different types of designs.

Note that there is a benefit to flexible interconnect generation strategies beyond supporting many wiring patterns. In a cASIC approach it is hard to add extra "spare" resources to the structure to support future larger designs. Since there are no regular patterns in the structures, one cannot request "more of the same". However, if a domain requires 10 tracks of length 16 wires with bus connectors, then it is easy to determine how to add 20% more interconnect resources. This issue will be considered in more depth in question 6 below.

Creation of flexible structures does require methods for determining how to effectively space out these resources. For example, when we have 10 tracks filled with length 16 wires, the staggering of breaks in the channel can have an impact on the interconnect quality. However, in a regular channel, these types of problems are separable, and can be investigated theoretically. In this specific case, we have developed an abstract model and both optimal (in restricted cases) and heuristic algorithms to best place breaks in a segmented interconnect [Compton03, Compton03a].

5. Do "flexible" architectures actually achieve their goal, and how do you measure flexibility?

One relatively unique requirement of a domain-specific FPGA is that it be flexible: flexible in order to handle new circuits not part of the initial specification, and flexible to support bug fixes and functionality upgrades. The architecture generation and layout generation tools must have flexibility to automatically create FPGAs for many different types of needs. Unfortunately, since flexibility is not a traditional requirement on ASIC systems, there is no standard way to quantify flexibility.

If we had a large enough set of circuits from a user's domain, testing flexibility would be simple. We would provide the tools with a small sample of circuits, generate an architecture, and then see how many of the domain members fit onto that architecture. However, we rarely have enough circuits for this, particularly since we want to understand whether a domain-specific FPGA will work for our *next* project, and the next project is invariably larger and more complex than what has been done before.

Our solution is synthetic circuit generation [Compton03, Compton04]. Techniques exist to take real circuits and automatically generate similar new circuits [Darnauer96, Hutton98, Wilton01, Hutton02]. Essentially, they profile the input circuit for fundamental properties such as logic mix, fanout, logic depth, etc., and use graph construction techniques to create new circuits with similar, though not identical, properties. These generated circuits can then be used as the large set of example circuits to evaluate a domain-specific FPGA system. Note that synthetic circuits do raise one danger: since the circuits are synthetic, and only mimic those properties that the synthetic circuit generation tool actually measures, it is possible that some unmeasured but critical feature of real circuits may be lost. The solution is to generate architectures with the synthetic circuits during flexibility analysis, and to measure with the real designs, so that we always determine what proportion of the *real* circuits can be supported by a domain-specific FPGA.

This approach of using synthetic circuits to generate the FPGA, and real circuits to evaluate them, does provide an additional opportunity. We can manipulate the settings for the synthetic benchmark creator to check the sensitivity of the domain-specific FPGA generators to different parameter mismatches. For example, the SoC designer may be concerned that future designs may have less locality, represented by a higher Rent exponent in their designs, and the domain-specific FPGA may be sensitive to this. To test this dependence, the synthetic circuit generator can be fed benchmark statistics with an artificially low Rent exponent. If architectures generated from these low Rent exponent circuits can support the real user designs (with the correct Rent exponent), this gives confidence that the domain-specific FPGAs can tolerate these alterations.

We have applied this flexibility measurement within our Totem-RaPiD effort [Compton03, Compton04] to evaluate our AMO RaPiD domain-specific FPGAs, which are the most flexible architectures we generate. For single-circuit examples, where we create a single synthetic benchmark based on one circuit, then map the real design to the resulting domain-specific architecture, we have a 93.8% success rate. When this is boosted to a full domain of circuits, with 5 synthetic benchmarks to control architecture generation, 99.7% of the real designs can be handled.

One might question whether this high success rate is due to real flexibility, or just an artifact of the testing methodology. To answer this, consider our GH (Greedy Histogram) technique, which was developed at the same time as AMO [Compton02, Compton03]. AMO evenly spread the logic units throughout the architecture, so that each region had the same proportion of the ALUs, multipliers, etc; GH was allowed to adjust the logic unit placement to better match the input benchmarks. AMO used tracks with only powers-of-2 lengths, and smoothly spaced segment breaks; GH could pick arbitrary track lengths and segment breaks based on the demands of the benchmarks. Thus, GH was a somewhat more cASIC-style version of AMO, using less regular interconnects and logic block placements to more closely match a given domain. For single-circuit tests only 18.5% of circuits could be accommodated by GH-produced architectures, and for full domain testing 91.6% of circuits can be supported. Given that the GH approach only achieves approximately a 1% area improvement over AMO, it is clear that the more regular architecture construction approach is superior. It is also clear that more regular structures provide much higher flexibility.

Note that one might consider a 99.7% success rate insufficient, since if a computation cannot be mapped to the device it is useless. However, these experiments are done without considering resynthesis. In a real domain-specific FPGA, just like a normal commodity FPGA, we can expect some initial circuit designs to fail to map. The user must then restructure the design somewhat, perhaps by applying time-multiplexing, to allow it to map to the device. Alternatively, as discussed in question 6 below, we can add resources to boost the range of circuits that can be supported on the domain-specific FPGA.

6. What is the best way to spend extra resources to ensure future designs will fit?

During the generation of a domain-specific FPGA, our goal is to create an architecture that best fits the user's domain, as represented by a set of example circuits. Architecture generation then creates as efficient an architecture as possible for that domain. However, we can expect that the user will want to map other designs to this substrate, and these new designs will likely require more logic, routing, or other resources. Thus, a common request from the SoC designer would be to add some slack to the architecture, spending extra silicon area to maximize the likelihood that future designs will also fit this substrate. However, the best way to use these extra resources is unclear.

The answer for most standard methods for including FPGAs into SoC is "more of the same". The FPGA IP is provided as a fixed tile, comprising both logic and routing, and the designer can simply add extra tiles in order to fill up the area to be devoted to reconfigurable logic. As such, we are assuming that the logic block complexity, ratio of interconnect to logic blocks, and connectivity pattern of the interconnect should not be varied as we include larger and larger arrays, and hope to support even wider ranges of circuits from a domain. It is not obvious that this is the best approach. For example, for most domains as circuits get larger their interconnect demands change, and emphasize more long-distance communication. Thus, when the SoC designer has extra resources to spend, it is unclear how to best spend them.

In our Totem-CPLD effort we have been able to explore this question. Our goal was to determine the most area-efficient strategy to add spare resources to allow future designs to fit. To create tests, we took each domain from our benchmark suite, removed one or more circuits from the domain, and determined whether the architecture generated for the reduced domain changed from the original. We discovered 49 cases where the base architecture was sensitive to a specific circuit, and used these for further testing. We measured, for a given resource addition strategy, how large an area increase was required to get a given circuit to fit.
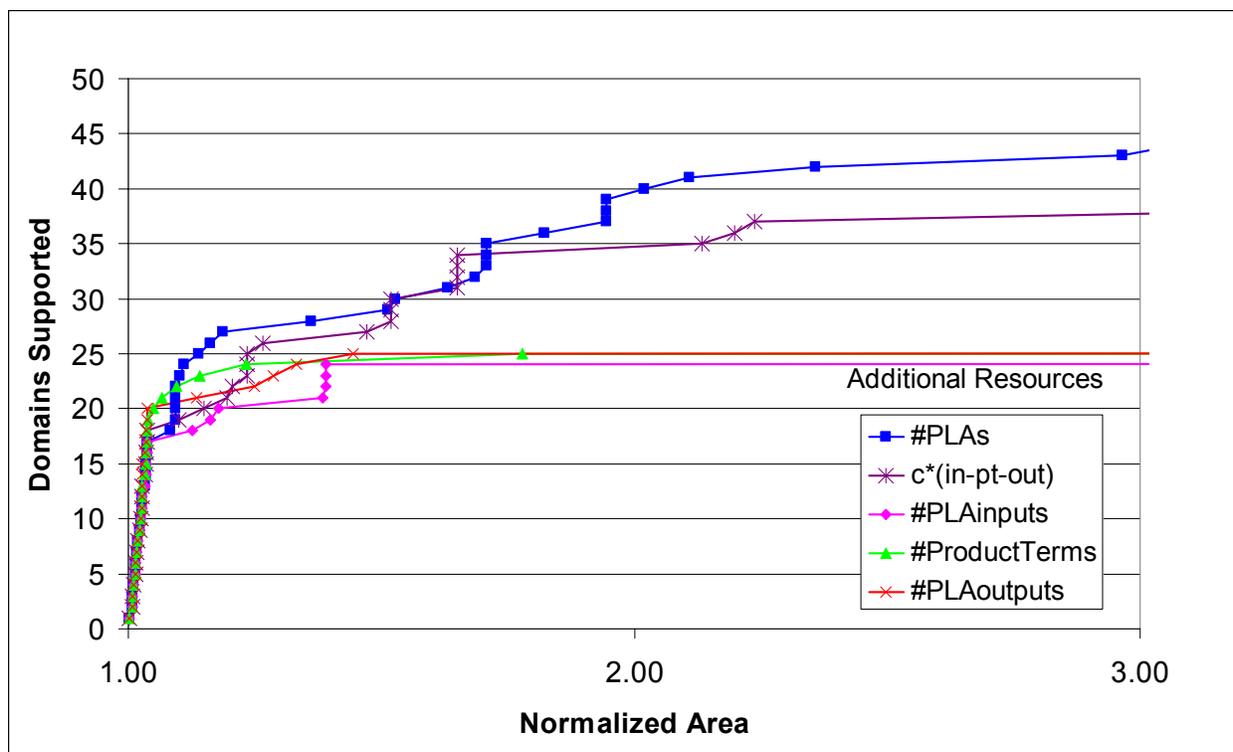

Figure 7. Area increase vs. additional domain circuits supported, given different strategies for using extra resources [Holland05].

The results of this investigation are shown in Figure 7. The horizontal axis is area increase, and the vertical is the number of troublesome mappings that work at the given area increase. Note that the curve is monotonically increasing, since once we have enough resources to fit a given design, it will still fit as we add even more resources. Only sensitive scenarios are plotted here – out of the single-circuit removal scenarios tested, 62% of them created the same architecture when that circuit was removed. The scenarios tested were:

1. #PLA: Increase the number of PLAs in the CPLD. Since the CPLD's routing structure is a sparse crossbar, the interconnect resources scale up proportionally with this change.
2. c*(in-pt-out): Increase the size of the PLAs, in terms of inputs, product terms, and outputs, by a multiplicative factor. Thus we might consider logic blocks with twice as many inputs, outputs, and product terms, and therefore can handle a significantly larger portion of the computation.

3. #PLAinputs: Increase the number of inputs to the PLAs.
4. #productTerms: Increase the number of product terms in the PLAs, thus allowing more complex functions to be implemented.
5. #PLAoutputs: Increase the number of outputs from the PLAs.

There are a couple of striking features in the graph in Figure 7. First, some strategies are doomed to failure, since they will never be able to support some computations. For example, when we just add product terms to the PLAs, we will never be able to support functions with wider fan-ins than the base set of circuits, since the number of inputs and number of PLAs doesn't change. This is shown by the #productTerms curve in the graph reaching a plateau. Similar reasoning explains why the #PLAinputs and #PLAoutputs curves also eventually stop supporting new circuits.

Increasing the number of logic units – #PLA – or the capacity of each logic unit – c*(in-pt-out) – will each eventually support any design, once enough capacity is available. However, it is clear that simply adding more PLA blocks is superior than increasing the logic block size, since the #PLAs curve generally dominates the c*(in-pt-out) curve. Hybrid techniques, which scale both the logic block capacity and number of logic blocks simultaneously, also were not as efficient as just increasing the number of logic blocks [Holland05]. Thus, the correct answer does seem to be "more of the same" – we can get significant benefits in the quality of implementation by optimizing logic block and interconnect structures to a domain (question 2 above), but to add capacity in the most efficient manner you simply add more of those same resources.

Note that this analysis held true for Totem-RaPiD as well [Compton03, Compton04] – if we used a good-quality architecture generator such as AMO, a new design from the domain would almost always fit if the architecture had enough of each type of logic unit (recall that RaPiD has multiple functional units, including ALUs, Multipliers, and RAM, and thus you'd need enough of each class). If you do not have enough resources, a strategy such as time-multiplexing or other resynthesis would be required to support the computation.

There was one dimension other than number of logic blocks that was found to be important in Totem-CPLD: the sparseness of the crossbars (or hence the flexibility of the interconnect). Our architecture generators reduce the number of switches in the crossbar until the base circuits just barely mapped to the array, thus saving area and delay in the crossbar. However, this may not be the best answer for supporting future circuits.
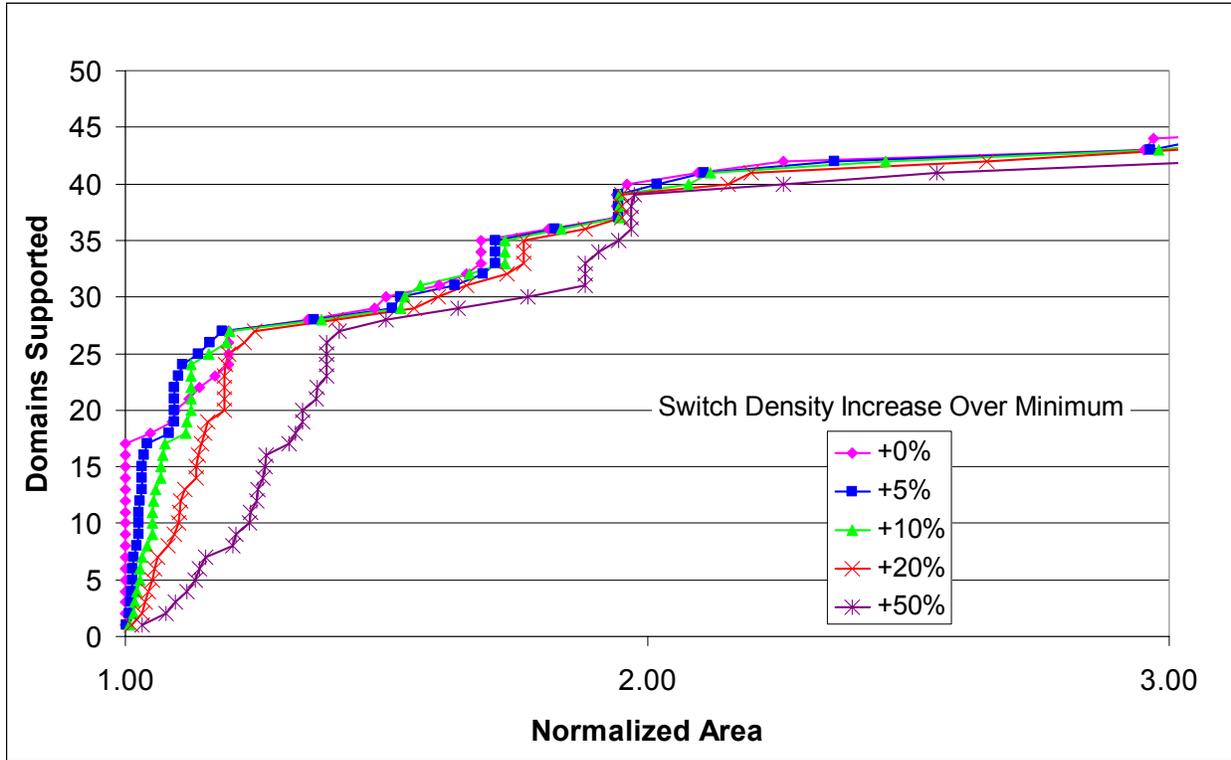
Figure 8. Area increase vs. additional domain circuits supported, by increasing the number of PLAs, given different amounts of increased switch densities in the sparse crossbars. [Holland05].

Figure 8 is similar to the graph of Figure 7, showing troublesome circuits mapped vs. the area increase in the architecture. All curves use the strategy of simply adding more functional units (#PLA from Figure 7), but each curve considers a different base crossbar richness. Specifically, the +0% curve uses the minimum number of switches in the crossbars, while +N% increases the switches per crossbar by N% over the minimum. As can be seen, the +0% and +5% curves are comparable. While the +0% curve performs somewhat better than the +5% line in some cases, in others it is even more significantly worse than the +5% line. Although the minimum switch density is sufficient for the base circuits, it can be too brittle to effectively handle some other designs. Note that to get +0% to work at all for some circuits we have to depopulate some of the PLAs, telling the technology mapper to under-pack the PLAs to allow additional flexibility in the router [Holland05]. The curve for +5% is almost always better than all other curves that add capacity, and is significantly less fragile (and thus often more effective) than the +0% curve. Thus, we believe that adding a small amount of additional flexibility to the interconnect structures is worthwhile, and all Totem-CPLD tests in Figure 7 use +5%. Although we did not find a similar effect in Totem-RaPiD, we believe this is actually inherent in our flexible tool flows. Specifically, our AMO architecture generator actually performs routing during architecture generation, though with an inferior (but fast) left-edge-algorithm heuristic [Compton02, Compton03]. The more powerful Pathfinder algorithm performs the final mapping of circuits to the generated architecture. Thus, we suspect that picking resources to meet the needs of a weak router, yet mapping designs with a high-quality router, may automatically add a sufficient margin of extra flexibility into the interconnect.

7. Do architectures need to support the worst-case resource demands across a domain?

One difficult problem in creating a domain-specific FPGA is balancing the resource requirements of different user designs, particularly when we consider the variable domains possible within an SoC. That is, if the "domain" consists of very different circuits with very different computation styles, the types and quantities of resources required can vary significantly from circuit to circuit. If the reconfigurable logic must be manufactured with the

worst-case resource demands across all circuits, this can yield an unacceptable architecture; unacceptably large, slow, and power-hungry.

In terms of logic functionality, it is often possible to rein in excessive resource requirements from any one circuit via logic resynthesis, often involving time-multiplexing. For example, a highly parallel implementation of the Frog encryption algorithm can produce a new value every two cycles, but requires 64 RAM blocks. Reducing the throughput to a value every 8 cycles reduces the memory requirements to 16 RAM blocks, and a value every 32 cycles requires 8 RAM blocks. By carefully considering different implementations of each design, and balancing resource requirements across the entire domain, we can create efficient architectures [Eguro02, Eguro03, Eguro05a].

Unfortunately, a similar approach may not be possible for the interconnect resources. For example, consider a domain of four circuits that require an interconnect channel width of 8, 9, 9, and 16 respectively. We might want to choose a consensus channel width of perhaps 9. For the interconnect-heavy circuit, we might hope to spread the circuit across a much larger architecture, thus spreading out the interconnect demands and reducing the per-channel requirements. However, such spreading is not generally considered in existing mapping tools [Betz97], which will ignore limited interconnect resources and instead tightly cluster the circuit, resulting in a routing failure [Eguro04]. While there are some mapping algorithms that can support congestion-aware placement and routing [Sharma01, Eguro05, Sharma05, Sharma05a], it is important to realize that the CAD tool flow that targets the domain-specific arrays has assumptions that may limit how well resource demands can be reduced or balanced.

8. *What role do fixed functional units play in a domain-specific architecture?*

One of the attractions of domain-specific FPGAs is the ability to include complex fixed-function units appropriate to the specific domain. This can radically improve the area, power, and performance of those portions of the computation that map to these units. In a sense this is the concept behind the original RaPiD architecture, utilizing only ALUs, multipliers, and memories to implement complete DSP applications.

To investigate the roles of complex functional units in domain-specific FPGAs, we developed RaPiD-AES [Eguro02], a RaPiD-style architecture with logic units optimized to private-key encryption. We began by carefully examining all of the 15 original candidates for the Advanced Encryption Standard competition (AES) [NIST02], to identify the types of operations they perform. We then grouped the operations together into basic functional units optimized to compute exactly these functions; these functional units would then be the basic units within the RaPiD-AES structure. The units we created were:

- *Multiplexer:* 32-bit, 2:1 muxes for computation and time-multiplexing support.
- *ALU:* Addition, subtraction, XOR, AND, OR, NOT, etc.
- *Rotate/Shift Unit:* 32-bit variable shifter with left/right rotate/logical shift/arithmetic shift.
- *Permutation Unit:* 32x32:1 statically controlled muxes, providing arbitrary connection from input to output bits.
- *RAM:* 256 byte memory addressable as eight 4 to 4 lookup tables (each with 4 pages of memory), eight 6 to 4 lookup tables, or one 8 to 8 lookup table.
- *32-bit Multiplier:* 32-bit integer multiplication with 64-bit output.
- *SIMD Multiplier:* 4x8-bit modulus 256 integer multiplications or 4x8-bit Galois Field multiplications.

With these fixed functional units, we were able to efficiently implement all of the computations found in the entire AES domain, including the original circuits as well as circuit modifications made as the competition progressed. Unfortunately, RaPiD-AES was a complete failure. After VLSI layout of all of the units, creation of a new compiler flow to target the device, and implementation of all AES algorithms in RaPiD-C, the domain-specific results were *significantly* worse in area-delay product compared to Verilog implementations we mapped to standard Xilinx FPGAs. Although some of this was due to the relative skill differences in layout between a world-class FPGA company and a bunch of undergraduate and graduate students, we believe much of the penalty is due to inherent challenges with fixed functional units within reprogrammable devices:

- Although a fixed functional unit may be significantly more efficient than generic FPGA units such as look-up tables (LUTs) for a given computation, overall they may be a loss. For example, if our shifter unit was 10x more efficient than LUTs for a rotation, but were less than 10% utilized across the domain of circuits, they

result in a net area loss. Note that such low utilizations are due not just to some circuits not using the resource at all, but also from circuits that only use a small portion of the provided fixed functional units.

- Even when fixed functional units are used, their placement in the architecture imposes a penalty. Specifically, an architecture generator will generally disperse these units throughout the architecture to support many different usage patterns. However, individual units will likely have more clustered unit usage dictated by the overall computation. Thus, we can expect significantly higher interconnect usage to route signals to where those units actually appear in the architecture, costing both area and delay. Also, logic that doesn't use the fixed functional units will also have increased routing costs since their signals must be routed past those unused units.

Both of these problems can be summed up in a single word: *fragmentation*. Fixed functional units break up the architecture into areas of differing functionalities, and the resulting fragmentation of their usage can impose significant area and performance penalties.

While fragmentation doomed our RaPiD-AES efforts, we still believe there is a role for fixed functional units in domain-specific FPGAs; the best indication of this is the increasing inclusion of such units into commodity FPGAs. Commercial architectures now have carry chains, RAM blocks, multipliers, and even complete microprocessors embedded into their fabric. Commodity FPGAs also point the way towards how to use these resources. Specifically, instead of attempting to support an entire computation with a set of fixed functional units, we instead start with an overall flexible FPGA fabric and then add fixed functions as they prove beneficial. Those that provide a definite advantage can be included, while other computations can remain in LUTs, thus avoiding unnecessary fragmentation of resources.

With this methodology of "flexible-first", even for beneficial units we may add a much smaller quantity of each unit than individual benchmarks may desire. For example, one design may want 90 multipliers within the fabric. However, if most other domain members need only 10 multipliers each, the most efficient solution is likely to be to only include those 10 multipliers, and map the remaining 80 multipliers from the worst-case design into the LUTs of the programmable fabric. This concept was not available to us within our RaPiD-AES effort, since there was no fully flexible unit as a fallback. Although we were able to reduce worst-case resource demands somewhat by time-multiplexing (discussed in question 7 above), this still was not sufficient.

9. How do you create the layout for a domain-specific FPGA?

Once architecture generation has completed, we must somehow create an implementation of that architecture. At this point, the reconfigurable logic is expressed as a circuit, and can be implemented like any other. The only difference from a normal design is that this circuit contains programming bits to control various features in the array, and these programming bits will be fabricated along with the rest of the design. Thus, the circuit remains reprogrammable, and once fabricated can be configured to implement whatever domain members are desired.

The simplest method for layout generation is to feed the architecture description directly into a standard cell flow. This technique will be able to support any architecture desired, and is compatible with an overall SoC flow. Note that this can be important, since if the rest of the design is in standard cells, the inclusion of another implementation style may be counter-productive. However, standard cell systems can have difficulty implementing FPGA hardware. Specifically, to a standard cell system an FPGA appears to be a huge set of combinational cycles, since there is generally an unregistered path between each gate of the design, including a path back to itself. Standard cell systems can have significant problems with combinational cycles, particularly in connection with timing optimization. One solution is to create directional FPGAs, architectures without combinational cycles [Kafafi03, Yan03]. Alternatively, a custom standard cell system can be developed to implement FPGAs [Padalia03, Kuon05]; this can also take advantage of FPGA-specific optimizations based on the regularity of FPGA tiles, and the interchangeability of programming bits.

Better implementations than basic standard cells are possible. An FPGA is composed of relatively few basic elements: multiplexers, programming bits, tristate drivers, basic switches, and D-flip-flops. Thus, it pays to have as optimized an implementation of these basic blocks as possible. We have found [Phillips01, Phillips02] that significant benefits can be obtained by adding hand-crafted standard cells to a standard cell library to support the

FPGA's exact needs. For example, in implementing RaPiD architectures the addition of five FPGA-specific standard cells yields an area only 0.83x the size of a version using normal standard cells.
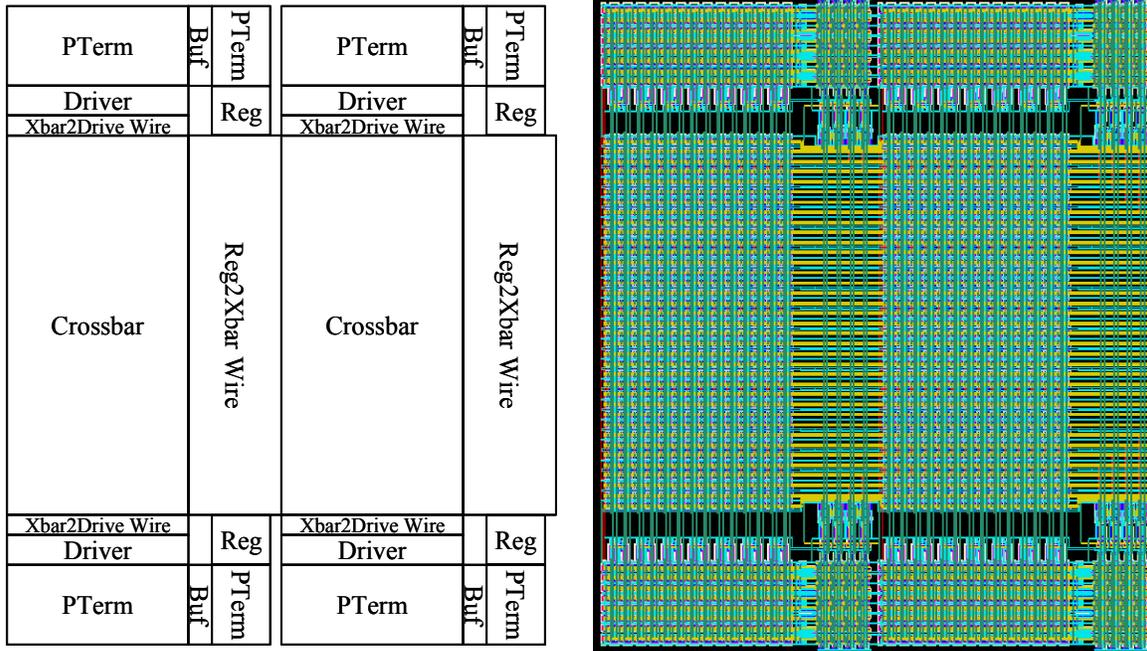
Figure 9. Floorplan of CPLD circuit generator structures (left) and the resulting layout (right) [Holland05].

We can not only custom develop FPGA-specific standard cells, but can also automatically create full-custom layouts. Consider a structure such as RaPiD, where complex – but regular – functional units are combined together to create an entire programmable device. In this case, we can develop a *circuit generator* to instantiate each individual element [Phillips04, Phillips05] parameterized based on possible architecture generator optimizations, similar to a datapath compiler. Then, by abutting these basic elements together we can create a complete reconfigurable array. Similarly, our circuit generator for CPLDs [Holland05, Holland05a] creates an overall design by abutting simple, parameterized elements (Figure 9). Note that circuit generators restrict the optimizations available to the architecture generator, since each hardware unit's generator will have some limitations on the types of elements it will create. For example, an ALU generator might be parameterized to different bit-widths, but might not be able to implement a hard-coded incrementer.
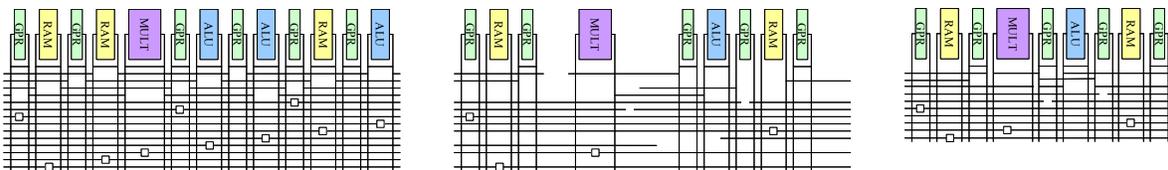
Figure 10. Abstract representation of template reduction. The initial architecture (left) is reduced by eliminating unneeded resources (center), and then compacted (right).

A final option is to leverage premade, full-custom FPGAs, yet still optimize the architecture to a specific domain. In *template reduction* [Phillips04, Phillips04a], architecture generation is restricted to use a common template that represents a superset of all possible allowed resources. The architecture generator selects a subset of those resources to form a domain-specific architecture. A highly optimized layout for the common template can be used as a starting point for layout generation, which simply removes whatever resources are not used by the generated architecture (Figure 10). This directly improves performance and power, since capacitance and leakage currents are reduced, and layout compaction can improve the layout's area. While this technique benefits from an initial full-

custom layout, the downside is that architecture generation can never add new styles of resources. Also, the template reduction operations can be complex to implement and are often tied to a specific fabrication process.
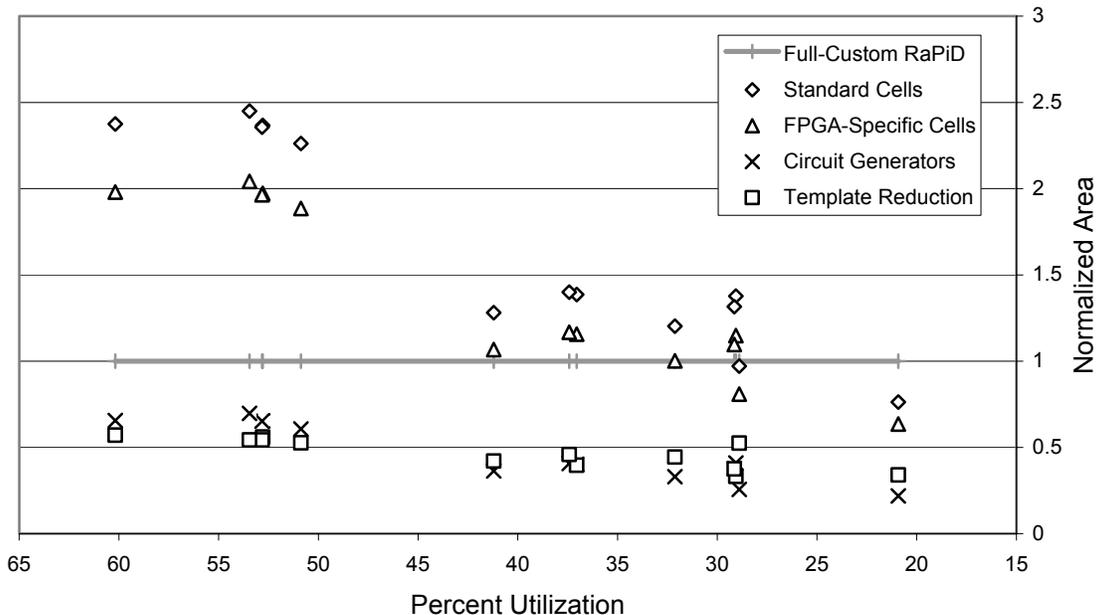

Figure 11. Area comparison of different layout generation techniques [Phillips04].

In order to compare all four layout generation techniques (Standard Cells, Standard Cells with FPGA-specific cells, Circuit Generators, and Template Reduction) we implemented all four techniques within a RaPiD flow [Phillips04]. Figure 11 shows an area comparison of these techniques. The vertical axis is area, normalized to the size of a full-custom RaPiD architecture implementation of that benchmark. The horizontal axis sorts the benchmarks based on utilization of the RaPiD architecture – designs that use relatively few of the resources provided by the standard architecture are likely to benefit the most from automatically customized domain-specific arrays. The graph demonstrates several key issues. First, the standard cell flow is 2.5x worse than the full-custom, fixed RaPiD architecture for designs that use most of RaPiD's resources, but as the designs deviate more significantly the standard cell flows approach or even surpass full-custom, though fixed, designs. Second, the circuit generator and template reduction flows achieve roughly the same results, providing approximately a factor of 2 improvement over full-custom, fixed reconfigurable architectures.

10. When can you apply full-custom layout techniques instead of just standard cells?

As discussed in 9 above, significant quality improvements are possible if we utilize a more aggressive layout approach than just vanilla standard cells. However, using such an approach may not be appropriate in all circumstances. For example, consider the advice to implement custom FPGA-specific standard cells. Although it provides a modest area improvement, each standard cell created is likely to be specific to a given fabrication process, since design rules vary from process to process. Thus, we either create a single set of additional standard cells, locking in the reconfigurable logic to a single process, or we must spend significant effort creating new libraries for most/all technologies, since commercial libraries for a new technology likely do not have the "right" cells.

For template reduction and circuit generators, there are greater concerns than just locking in to a given fabrication process. By using these technologies we restrict the type of optimizations that are useful during architecture generation. For example, in a circuit generator approach each generator makes assumptions on the overall structure, and set of possible optimizations, in order to create an efficient implementation. The generator for an ALU within a

18

RaPiD structure may assume that the I/O connections come from the left and right, and the height of all units in the architecture will be the same in order to "pitch-match" these connections. Thus, an architecture generator transforming an ALU to reduce its height may simply not be supported. Template reduction may allow the architecture generator to remove subcomponents of the ALU, since individual transistors and connections can be deleted in a straightforward manner. However, since the neighboring units may not be reduced in the same way, compaction will be unable to reduce the overall area because of cross-constraints.
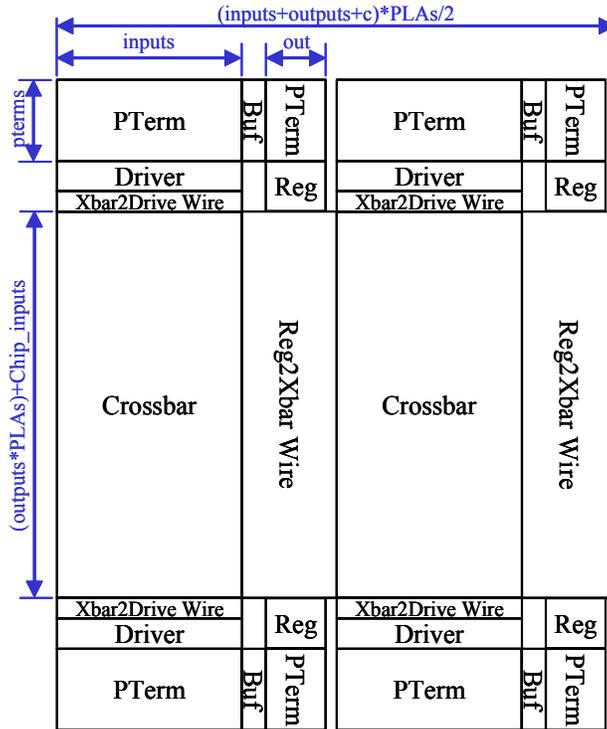
Figure 12. CPLD layout augmented with the scaling dimensions. Some constants are not shown on the dimensions for simplicity.

We have found that an automatically generated reconfigurable system's full-custom layout generally has a set of "scaling dimensions", features of a layout that scale with properties. For example, the height of a RaPiD architecture is dictated by the bitwidth of the computations, and the width based on the set of operations included. For a CPLD, the component areas are dependent on the number of PLA inputs, outputs, and product terms, as well as the number of PLAs in the CPLD. This is shown in Figure 12. An architecture generator can alter these features almost at will and the layout will adjust automatically. However, if the architecture generator makes an optimization that is not fully aligned with a "scaling dimension" the layout generator may not be able to efficiently support that transformation. For example, in the CPLD if we choose to take one PLA and decrease its number of product terms by half, there will be no improvement in the layout area, since the height of the other PLAs will still dictate the overall layout dimensions. This change may improve performance or power, but even these are limited by the inability to restructure the layout.

19
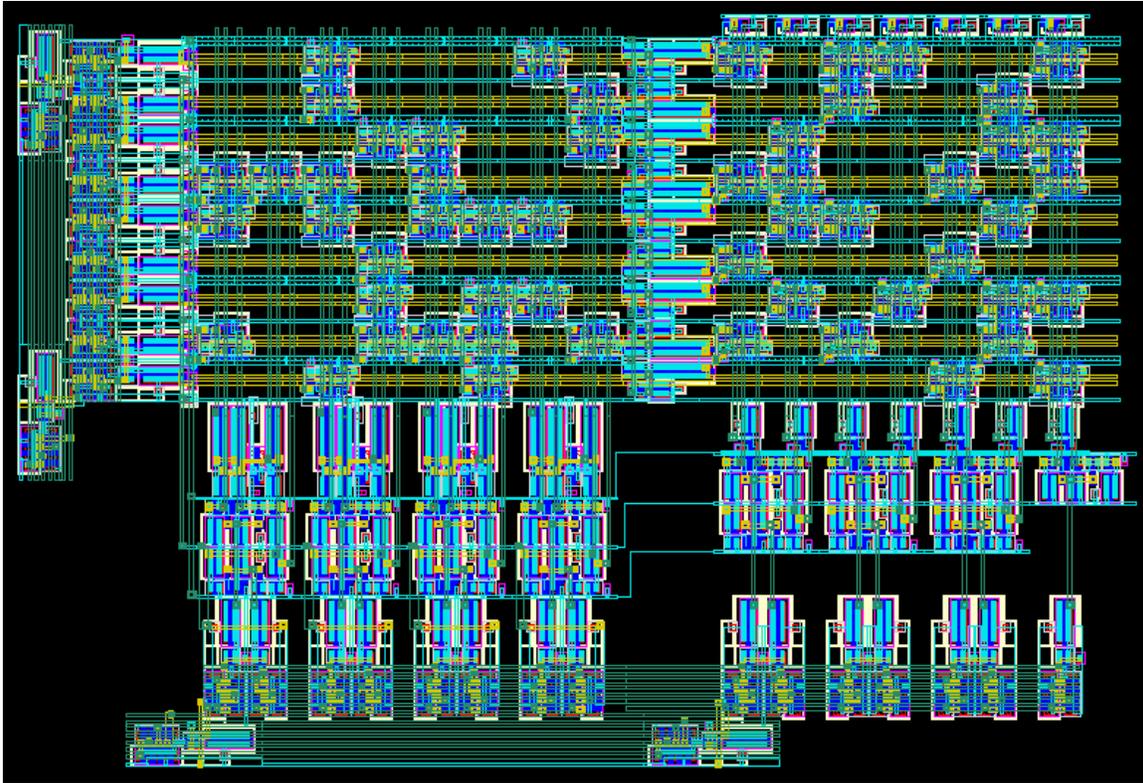
Figure 13. Example of a PLA with switches eliminated [Holland04, Holland05].

Before we realized the importance of scaling dimensions, we developed a PAL and PLA generator that reduces the number of connections in the array by placing product terms with similar connection patterns from different circuits into the same position [Holland04, Holland05]. This system was able to eliminate 66% to 74% of the programmable connections from a PAL or PLA, with a delay improvement of 16% to 31%. Unfortunately, there were no area improvements from this technique, for reasons apparent in the resulting layout shown in Figure 13. Notice that although the central arrays are relatively lightly populated, there is little a compaction algorithm can actually do with this layout. For example, through either direct adjacencies or by touching corners, a path of programmable elements can be found through the entire width and height of the PLA. Thus, when we invoked a compactor on this array, no benefit was found. For a system such as this, a standard cell implementation might be preferred, since with the area overhead of perhaps 2x (assuming PLA-specific cells), but an elimination of almost 75% of the logic elements, a net area benefit will likely be achieved.

Note that if the layout generator is specifically designed to minimize cross-constraints, some irregularities in layouts can be supported. For example, to support sparse crossbars as interconnect matrixes within a CPLD, we knew that perhaps only 20% of the crossbar positions would be populated. Thus, our basic crossbar switchpoint was designed to easily slide vertically in the array, with vertical routing to connect the switchpoint to the appropriate horizontal wire, and with very tight packing of horizontal wires. However, these alterations may come at a price; reducing cross-constraints may require greater area or circuit delay.

The main conclusion is that if we expect to use a full-custom layout generator such as template reduction or circuit generators, the architecture generator and layout generator tools must be coupled. The layout generator must be designed to support the optimizations included in the architecture generator, and the architecture generator can only create architectures that the layout generator can efficiently implement. This is of particular concern in 2D architectures such as standard island-style FPGAs; in an island-style FPGA the size of a routing channel must be fixed for the channel running the entire length or width of the array, and the switchboxes (locations where vertical and horizontal tracks are cross-connected) may impose a coupling between the sizes of the vertical and horizontal

channels. Thus, the types of optimizations available may be significantly restricted. In RaPiD and in CPLDs, there are many more scaling dimensions available, since many of the features are independent.

11. How do you provide a CAD suite for mapping user designs onto automatically generated domain-specific FPGAs?

A domain-specific reconfigurable system is useless unless there is a corresponding toolsuite to map user designs onto this substrate. While an architecture generator will generally map the initial user designs to the architecture as it is created, the SoC user will inevitably want to map new designs to the array, either to handle new functionality, or for bug fixes on the initial circuits.

Several steps in the mapping process for domain-specific logic are easy to support. For synthesis and technology mapping, the logic inside a domain-specific device is usually chosen from a set of standard styles of units, units that are supported by existing tools. Thus, our tool to generate a customized RaPiD will choose the number of ALUs and Multipliers, but will not alter the type of units included. For CPLDs, the system can choose the number of inputs, outputs, and product terms for the PLAs. However, existing mapping tools for PLAs such as PLAmap [Chen01] already allow the mapping to be parameterized based on these factors.

Routing is also easy to support. The main routing algorithm for FPGAs, Pathfinder [McMurchie95], is already architecture-adaptive; a new architecture is modeled by a routing graph and given to Pathfinder, which automatically maps to this interconnect structure.

Placement is much harder. Placement algorithms are generally tied directly to the underlying FPGA architecture by the tool's interconnect estimator used to calculate the placement cost. Specifically, placement is really the process of assigning logic computations to FPGA logic blocks in order to improve the resulting routing. Most current placement algorithms, including the prevalent academic placer in VPR [Betz97], abstract the routing problem into a fast estimator, which makes assumptions on the features of the interconnect. If the generated architecture matches these assumptions, the placement algorithm will still work. Thus, the VPR placer can be used for island-style architectures with abundant routing resources and homogeneous routing channels. For RaPiD, our placer can handle any resource mix and channel capacity (including highly congested channels), as long as the channel width is constant across the array [Sharma01].

Unfortunately, many architecture modifications will not conform to the assumptions of an existing architecture-dependent placer. For example, as discussed in question 7 above, we might develop an island-style architecture with a channel width dictated by the typical, not worst-case, requirement across the circuits of the domain. This would boost the overall system quality. However, this violates the assumptions in VPR, which cannot handle congested channels.

A solution we have explored is to eliminate heuristic routing estimates. Specifically, if the goal of placement is to facilitate a high-quality routing, then the router itself can be used to estimate routeability. Since Pathfinder is architecture-adaptive, we can make calls to Pathfinder during placement to estimate the quality of a given placement. Note that complete routings are not necessary for each placement, since a placer normally makes a series of small perturbations to an existing placement, and thus requires only slight perturbations to the routing as well. The runtime penalties for this technique are significant, but since we expect these domain-specific reconfigurable systems to only occupy a small portion of a System-on-a-Chip, this does help limit the size (and thus runtime) of the placement approach. We have developed an architecture-adaptive placer called Independence [Eguro05, Sharma05, Sharma05a, Sharma05b], which achieves similar or better results that architecture-specific tools on a wide range of architecture types, including VPR/island-style, tree-based HSRA [DeHon99], and RaPiD.

Future Work

In our efforts since 1999 we have been able to answer many of the questions surrounding automatic generation of domain-specific FPGAs. However, there is still much left to do. Perhaps the biggest open question is how these results track to different FPGA styles. We chose to focus on RaPiD because of local expertise and collaboration with Carl Ebeling's group, the simplifications possible from using a 1D structure, and the use of complex functional

units. This meant many questions could be quickly answered within the RaPiD structure. Similarly, our CPLD work was motivated both by the ability to explore an architecture that is quite different than RaPiD, but is still very flexible, and has an interesting interconnection structure. Also, the simplicity of the mapping flow to PLAs/PALs meant that interactions with synthesis could be considered.

However, the major growth area in reconfigurable logic is in island-style, 2D architectures, and it is likely that such architectures will form a significant portion of the commercial market for reconfigurable logic in SoC. Whether island-style architectures will exhibit the same characteristics as the ones studied here is a major open question. We believe they will, except that the layout constraints of scaling dimensions (question 10 above) will have an even greater restriction on architecture generators that target aggressive layout styles such as template reduction and circuit generators.

A second major open question is logic synthesis support. While synthesis interactions were a major consideration in Totem-CPLD, resynthesis of troublesome designs requires careful consideration. At times, one circuit in a domain will have features disproportionate to the other circuits within the domain. If this circuit is used in architecture generation, it can distort the resulting architecture. If the circuit is not used in architecture generation, it may not fit on the previously-generated architecture. For both of these cases, the circuit should be resynthesized to limit the impact of its excessive resource requirements. For example, a tool to automatically time-multiplex designs would be invaluable. Our efforts to time-multiplex encryption circuits by hand for RaPiD-AES clearly demonstrated the advantage of (and need for) this approach.

A final issue is the need for accurate quality metrics for domain-specific FPGA creation systems. We made extensive investigations into area and flexibility, and some considerations of performance. However, studies of the power implications of these systems will be very useful. While we believe the power improvements will track somewhere between the area and performance improvements, this should still be explored. Also, our performance studies have been somewhat simplistic. Although we alter circuit structures in ways that decrease capacitance, and thus improve performance, we do not currently have mechanisms to change transistor sizings in response to these changes.

Conclusions

It seems clear that SoC design flows will by necessity incorporate reconfigurable logic.. While the continuing increase of fabrication costs decreases the number of new ASIC starts, it also demands that each ASIC actually produced be capable of being used in a large number of systems. Reconfigurable logic provides a mechanism to adjust the hardware to support a wide range of computations with a single chip, as well as a way to reduce the likelihood of fatal design errors. These features make reconfigurable logic a natural choice for SoC design

While current commercial ventures are seeking to exploit this opportunity by deploying premade IP tiles that can be tiled to form NxM arrays, they miss a huge opportunity: customizing the reconfigurable logic to the specific needs of the target SoC. As we have demonstrated, this optimization yields 5x improvements in area and up to 11x in area-delay product compared to standard reconfigurable systems. In some cases these implementations even rival those of fixed ASIC solutions. The creation of these architectures, layouts, and CAD tools are quite manageable, though care must be taken in the architecture/layout coupling. Architectures must support efficient routing estimators or be prepared to use significantly slower placement approaches. Finally, by using regular, flexible structures and carefully adding spare resources, these systems exhibit excellent flexibility to support new or upgraded circuits developed after chip fabrication.

Acknowledgements

References

[Actel04]   Actel Corporation, "VariCore™ Embedded Programmability - Flexible by Design", <http://varicore.actel.com/cgi-bin/varicore.cgi?page=overview> (30 January 2004).

[Betz97]   2. V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", *7th International Workshop on Field-Programmable Logic and Applications,* pp 213-222, 1997.

[Biehl93]   G. Biehl, "Overview of Complex Array-Based PLDs", in H. Grünbacher, R. W. Hartenstein, Eds., *Lecture Notes in Computer Science 705 - Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping*, Berlin: Springer-Verlag, pp. 1-10, 1993.

[Chen01]   D. Chen, J. Cong, M. Ercegovac, Z. Huang, "Performance-Driven Mapping for CPLD Architectures", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays,* 2001, pp. 39-47.

[Compton01]   K. Compton, S. Hauck, "Totem: Custom Reconfigurable Array Generation", *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.

[Compton02]   K. Compton, A. Sharma, S. Phillips, S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", *International Conference on Field Programmable Logic and Applications,* pp. 59-68, 2002.

[Compton03]   K. Compton, *Architecture Generation of Customized Reconfigurable Hardware*, Ph.D. Thesis, Northwestern University, Dept. of ECE, 2003.

[Compton03a]   K. Compton, S. Hauck, "Track Placement: Orchestrating Routing Structures to Maximize Routability", *International Conference on Field Programmable Logic and Applications*, 2003.

[Compton04]   K. Compton, S. Hauck, "Flexibility Measurement of Domain-Specific Reconfigurable Hardware", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 155-161, 2004.

[Compton06]   K. Compton, S. Hauck, "Automatic Design of Configurable ASICs", submitted to *IEEE Transactions on VLSI Systems*.

[Cronquist99] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Twentieth Anniversary Conference on Advanced Re-search in VLSI*, 1999.

[Darnauer96]   J. Darnauer, W.W.-M. Dai, "A Method for Generating Random Circuits and its Application to Routability Measurement", *ACM Symposium on Field Programmable Gate Arrays*, 1996.

[DeHon99]   A. DeHon, "Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization)," *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 69 – 78, 1999.

[DeMicheli94] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, New York: McGraw-Hill, Inc. 1994.

[Ebeling96]   C. Ebeling, D. C. Cronquist, P. Franklin, "RaPiD – Reconfigurable Pipelined Datapath.", *Lecture Notes in Computer Science 1142—Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, R.W. Hartenstein, M. Glesner, Eds. Springer-Verlag, Berlin, Germany, pp. 126-135, 1996.

[Eguro00]   K. Eguro, S. Hauck, "synFPGA: Application Specific FPGA Synthesis", *Northwestern University, Dept. of ECE Technical Report,* 2000.

[Eguro02]   K. Eguro, *RaPiD-AES: Developing an Encryption-Specific FPGA Architecture*, Master's Thesis, University of Washington, Dept. of EE, 2002.

[Eguro03]   K. Eguro, S. Hauck, "Issues and Approaches to Coarse-Grain Reconfigurable Architecture Development", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 111-120, 2003.

[Eguro04]   K, Eguro, S. Hauck, "Issues of Wirelength Cost Models in Routing-Constrained FPGAs", *University of Washington, Dept. of EE Technical Report UWEETR-2004-0006*, 2004.

[Eguro05]   K. Eguro, S. Hauck, A. Sharma, "Architecture-Adaptive Range Limit Windowing for Simulated Annealing FPGA Placement", *Design Automation Conference*, 2005.

[Eguro05a]   K. Eguro, S. Hauck, "Resource Allocation for Coarse Grain FPGA Development", to appear in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October, 2005.

[Elixent03]   Elixent, "Applications of D-Fabrix", 2003.

[Holland04]   M. Holland, S. Hauck, "Automatic Creation of Reconfigurable PALs/PLAs for SoC", *International Conference on Field Programmable Logic and Applications*, pp. 536-545, 2004.

[Holland05]   M. Holland, *Automatic Creation of Product-Term Based Reconfigurable Architectures for System-on-a-Chip*, Ph.D. Thesis, University of Washington, Dept. of EE, 2005.

[Holland05a]   M. Holland, S. Hauck, "Automatic Creation of Domain-Specific Reconfigurable CPLDs for SoC", *International Conference on Field Programmable Logic and Applications*, 2005.

[Hutton02]   M. Hutton, J. Rose and D. Corneil, "Automatic Generation of Synthetic Sequential Benchmark Circuits", *IEEE Transactions on CAD*, Vol. 21, No. 8, pp. 928-940, August 2002.

[Hutton98]   M. Hutton, J. Rose, J. Grossman, and D. Corneil, "Characterization and Parameterized Generation of Synthetic Combinational Benchmark Circuits", *IEEE Transactions on CAD*, Vol. 17, No. 10, pp. 985-996, October 1998.

[Kafafi03]   N. Kafafi, K. Bozman, S.J.E. Wilton, ``Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores'', in the ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, Feb 2003, pp. 1-9.

[Kouloheris91] J. Kouloheris, A. El Gamal, "FPGA Performance vs. Cell Granularity", *IEEE Custom Integrated Circuits Conference*, 1991, pp. 6.2/1-6.2/4.

[Kuon05]   I. Kuon, A. Egier and J. Rose, "Design, Layout and Verification of an FPGA using Automated Tools" ACM Symposium on FPGAs, February 2005, pp 215-226.

[LSI04]   Advanced Products: Introducing LiquidLogic Embedded Programmable Logic Core, <http://www.lsilogic.com/products/asic/advanced_products.html> (30 January 2004).

[McMurchie95] L. McMurchie, C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *ACM/SIGDA Symposium on FPGAs*, pp 111-117, 1995.

[NIST02]   National Institute of Standards and Technology. *Advanced Encryption Standard (AES) Development Effort*. Nov. 11, 2002. <http://csrc.nist.gov/encryption/aes/index2.html>.

[Padalia03]   K. Padalia, R. Fung, M. Bourgeault, A. Egier, J. Rose, "Automatic transistor and physical design of FPGA tiles from an architectural specification", *ACM/SIGDA Symposium on FPGAs*, pp. 164–172, 2003.

[Phillips01]   S. Phillips, *Automatic Layout of Domain Specific Reconfigurable Subsystems for System-on-a-Chip,* Master's Thesis, Northwestern University, Dept. of ECE, 2001.

[Phillips02]   S. Phillips, S. Hauck, "Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 165-173, 2002.

[Phillips04]   S. Phillips, *Automating Layout of Reconfigurable Subsystems for Systems-on-a-Chip*, Ph.D. Thesis, University of Washington, Dept. of EE, 2004.

[Phillips04a]   S. Phillips, A. Sharma, S. Hauck, "Automating the Layout of Reconfigurable Subsystems Via Template Reduction", *International Conference on Field Programmable Logic and Applications*, pp. 857-861, 2004.

[Phillips05]   S. Phillips, S. Hauck, "Automating the Layout of Reconfigurable Subsystems Using Circuit Generators", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[Sharma01]   A. Sharma, *Development of a Place and Route Tool for the RaPiD Architecture*, Master's Thesis, University of Washington, Dept. of EE, 2001.

[Sharma05]   A. Sharma, *Place and Route Techniques for FPGA Architecture Advancement*, Ph.D. Thesis, University of Washington, Dept. of EE, 2005.

[Sharma05a]   A. Sharma, C. Ebeling, S. Hauck, "Architecture-Adaptive Routability-Driven Placement for FPGAs", *International Conference on Field Programmable Logic and Applications*, 2005.

[Sharma06]   A. Sharma, S. Hauck, "Accelerating FPGA Routing Using Architecture-Adaptive A* Techniques", submitted to *IEEE International Conference on Field Programmable Technology,* 2005.

[Wilton01]   S. Wilton, J. Rose, Z. Vranesic, "Structural Analysis and Generation of Synthetic Digital Circuits with Memory", *IEEE Transactions on VLSI*, Vol. 9, No. 1, pp. 223-226, February 2001.

[Xilinx04]   "IBM, Xilinx shake up art of chip design with new custom product", <http://www-3.ibm.com/chips/news/2002/0624_xilinx.html>, (15 August 2004).

[Yan03]   A. Yan, S.J.E. Wilton, ``Product Term Embedded Synthesizable Logic Cores'', *IEEE International Conference on Field-Programmable Technology*, Tokyo, Japan, Dec. 2003, pp. 162-169.