Layout Generation for Domain-Specific FPGAs

Shawn Phillips[1], Akshay Sharma[2], Scott Hauck[3]
1 Annapolis Microsystems, Inc., Annapolis, MD, sphillips@annapmicro.com
2 Actel Corporation, Mountainview, CA, akshay.sharma@actel.com
3 University of Washington, Dept. of EE, Seattle, WA, hauck@ee.washington.edu

***Abstract*— When designing systems-on-a-chip (SoCs), a unique opportunity exists to generate custom FPGA architectures that are specific to the application domain in which the device will be used. The inclusion of such devices provides an efficient compromise between the flexibility of software and the performance of hardware, while at the same time allowing for post-fabrication modification of the SoC. To automate the layout of reconfigurable subsystems for systems-on-a-chip, we present three alternative methods, namely Template Reduction, Circuit Generator, and Standard Cell methods. Template Reduction begins with a full-custom layout as a template that is a superset of the required resources, and removes those resources that are not needed by a given application domain. Circuit Generator takes advantage of the regularity that exists in FPGAs by using circuit generators to create the custom reconfigurable devices. Finally, Standard Cell automates the creation of circuits by using a standard cell library that has been optimized for reconfigurable devices. This paper presents algorithms for each of these approaches, and quantifies the relative quality in terms of area and delay.**

***Index Terms*—Design Automation, Layout, Reconfigurable Architectures**

I. Introduction

TRADITIONAL FPGAs are a very effective bridge between software running on a general-purpose processor (GPP) and application-specific integrated circuits (ASIC). FPGAs are extremely flexible, enabling one device to target multiple application domains. However, to achieve this flexibility FPGAs must sacrifice size, performance, and power consumption when compared to ASICs, making them less than ideal for high performance designs. Domain-specific FPGAs can be created to combine the flexibility of FPGAs with area and performance near that of ASICs.

In the standard FPGA world, there is a limit to the number and variety of FPGAs that can be supported – large nonrecurring-engineering (NRE) costs due to custom fabrication costs and design complexity means that only the most widely applicable devices are commercially viable. However, a unique opportunity exists in the System-on-a-Chip (SoC) world. Here, an entire system, including perhaps memories, processors, DSPs, and ASIC logic are fabricated together on a single silicon die. FPGAs have a role in this world as well, providing a region of programmability in the SoC that can be used for run-time reconfigurability, bug fixes, functionality improvements, multi-function SoCs, and other situations that require post-fabrication customization of a hardware subsystem. This gives rise to an interesting opportunity. Since the reconfigurable logic will need to be custom fabricated along with the overall SoC, the reconfigurable logic can be optimized to the specific demands of the SoC through the creation of domain-specific reconfigurable devices.

A domain-specific FPGA is a reconfigurable array that is targeted at a specific application domain, instead of the multiple domains a traditional FPGA targets. Creating custom domain-specific FPGAs is possible when designing an SoC, since even early in the design stage designers are aware of the computational domain in which the device will operate. With this knowledge, designers could then remove from the reconfigurable array unneeded hardware and programming points that would otherwise reduce system performance and increase the design area. Architectures such as RaPiD [1, 2], PipeRench [3], and Pleiades [4], have followed this design methodology in the digital signal processing (DSP) computational domain, and have shown improvements over reconfigurable processors. This ability to utilize custom arrays instead of ASICs in high performance SoC designs will retain the post-fabrication flexibility of FPGAs, while also meeting stringent performance requirements that until now could only be met by ASICs.

Unfortunately, if designers were forced to create custom reconfigurable logic for every new chip, it would be impossible to meet any reasonable design cycle. However, by automating the generation of the domain-specific FPGAs, designers would avoid this increased time to market and would decrease the overall design cost.

The goal of the Totem project [5, 6, 7, 8, 9, 21] is to reduce the design time and effort in the creation of a custom reconfigurable architecture. The architectures that are created by Totem are based upon the applications and constraints specified by the designer. Since the custom architecture is optimized for a particular set of applications and constraints, the designs are smaller in area and perform better than a standard FPGA while retaining enough flexibility to support the specified application set, with the possibility to support applications not foreseen by the designer.

In this paper, we first present a short background on the RaPiD architecture and on the Totem project. Next, we examine the approaches used to automate the layout process, namely Template Reduction, Circuit Generator, and Standard Cell methods. The experimental setup and procedure that we have used to evaluate the designs created by the various methods will then be presented. Finally, we will show how well our approaches perform.

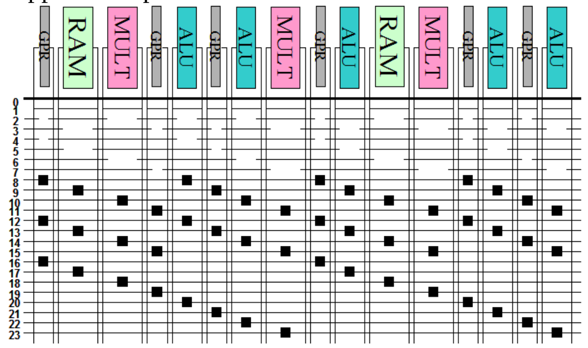


Fig. 1: Block diagram of one RaPiD II cell. Data flows through the array horizontally, with vertical routing providing connections to functional units. The black boxes in the interconnect represent bus connectors, which can be used to connect tracks into long lines or to separate them into short lines.

II. RaPiD

The Reconfigurable-Pipelined Datapath (RaPiD) [1, 2] has been chosen as a starting point for the architectures that are generated by the Totem project. The goal of the RaPiD architecture is to provide performance at or above the level of that of a dedicated ASIC, while also retaining the flexibility that reconfigurability provides. RaPiD is able to achieve these goals through the use of coarse-grain components, such as memories, ALUs, multipliers, and pipelined data-registers. We use a version of Rapid called RaPiD II (Fig. 1), with augmented resources to better support the benchmarks considered in this paper. As such, it represents our baseline for an optimized, fixed structure FPGA.

III. Totem

The Totem design flow attempts to improve the quality of reconfigurable logic by providing only those resources required for a given application domain. Totem automatically creates these custom architectures. The overall Totem design flow (Fig. 2) can be broken into three parts: architecture generation, VLSI layout generation, and place-and-route tool generation.

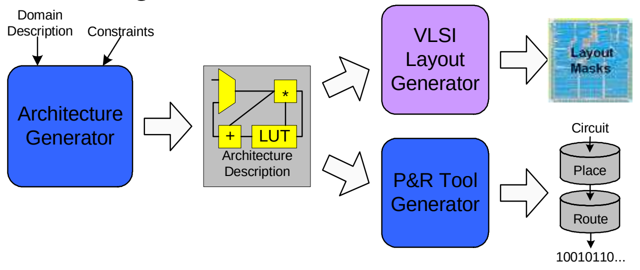


Fig. 2: Totem tool flow.

A. *Architecture Generation*

The first phase of creating a custom reconfigurable device is high-level architecture generation [5, 6]. The Architecture Generator will receive, as input from the designer, the target algorithms and any associated constraints, such as area or performance. The high-level Architecture Generator will then create a Verilog representation of the architecture that meets all of the designer’s requirements. The more diverse the algorithms specified by the designer, the more flexibility the final architecture will have. The output of the architecture generator is the logic, routing, and programming bits of the domain-specific FPGA. Once fabricated, the architecture can be programmed to support the target or similar circuits.

B. *VLSI Layout Generation*

The next phase in generating the custom architecture is to automatically create mask layouts, which is performed by the VLSI layout generator. The layout generator will receive, as input from the high-level Architecture Generator, the Verilog representation of the custom circuit. The layout generator must be able to create layouts for any conceivable circuit that the high-level architecture generator is capable of producing. We have investigated three possible methods of automating the layout process: Template Reduction [9], Circuit Generators [10], and Standard Cells [7]. This paper is concerned with this aspect of the Totem Project, and each of these methods will be discussed in sections 4, 5, and 6 respectively.

C. *Place and Route Tool Generation*

The final phase in developing a custom architecture is to generate the place-and-route tools that will enable the designer to utilize the new architecture [8]. The Place-and-Route Tool Generator creates mapping tools by using the Verilog provided by the high-level Architecture Generator. The placer uses simulated annealing [11] and a cutsize-based metric to match RaPiD's 1D routing structure [8]. The router uses the Pathfinder algorithm [12], targeted to a routing graph extracted from the Verilog produced by the Architecture Generator.

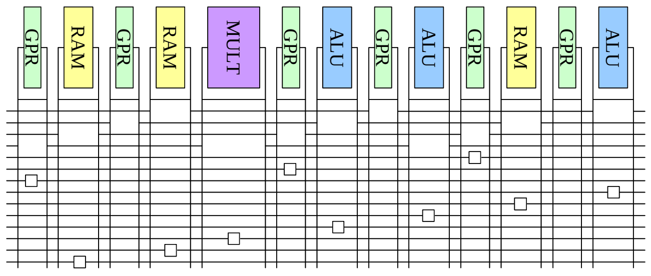
IV. Template Reduction Method

The idea behind template reduction [9] is to start with a full-custom layout that provides a superset of the required resources, and remove those resources that are not needed by a given domain (Fig. 3). This is done by actually editing the layout in an automated fashion to eliminate the transistors and wires that form the unused resources, as well as replacing programmable connections with fixed connections or breaks, for flexibility that is not needed. In this way, we can get most of the advantage of a full-custom layout, while still optimizing towards the actual intended usage of the array. By using these techniques, we leverage high-quality full custom layouts, while retaining the ability to remove unneeded flexibility to create further gains in both area and performance.

Template reduction has been broken into three main tasks. The first is the creation of a feature rich macro cell, which is used as an initial template that will be reduced and compacted to form the final circuit. The second is the creation of the reduction list that identifies the resources that should be removed. This is generated by Totem's place and route tool, which seeks to increase the commonality of resource usage between all of the mappings to the reconfigurable logic, and thus increase the amount of resources that can be eliminated. The final task is the implementation of the reductions on the template, followed by the compaction of the resultant circuit. This involves automated layout restructurings to edit the actual design files based upon the reduction list. Each of these tasks will be outlined in the following sections.

A. Feature Rich Template

The creation of the feature rich template is the most critical aspect related to the Template Reduction Method. A poor template will not be able to support a wide range of applications, which in turn weakens the effectiveness of the method. Therefore, we performed extensive profiling of the potential benchmark sets to create the RaPiD II tile. We then created a high-quality, full custom layout of the RaPiD II tile, which was the feature rich template used for Template Reduction.



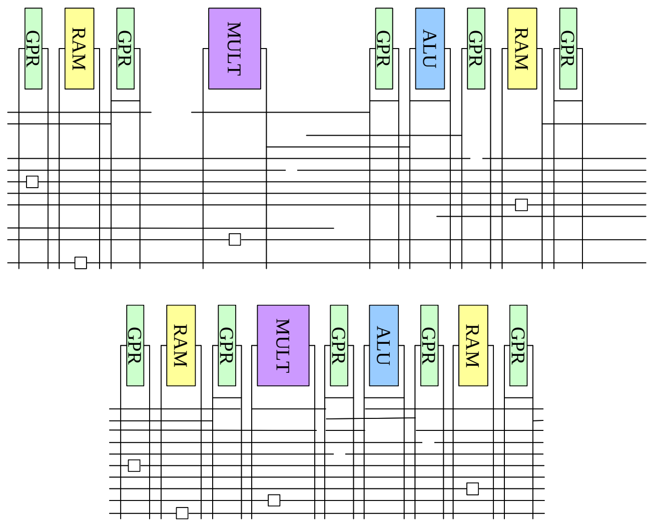


Fig. 3: Template reduction in action. The block diagram of a feature rich macro cell is shown on the top. In the middle, the macro cell has been reduced by the removal of routing resources and functional units that are not needed to support the application domain. On the bottom, the final compacted cell.

B. *Reduction List Generation*

The next task in template reduction is the creation of the reduction list [13]. The creation of the reduction list is performed by a subtractive scheme that eliminates as many functional units and routing resources (functional units and routing resources are collectively called "resources") as possible while placing and routing a set of netlists onto the template architecture. Individual netlists in the set are individually placed and routed on the template architecture. At the end of this first run, the fraction of netlists that used each resource in the template is recorded, and a cost (referred to as usage_cost) is assigned to each resource based on the fraction of netlists that used the resource during the previous run. The usage_cost of a resource is inversely proportional to the fraction of netlists that used the resource. Thus, the usage_cost of a resource that was used by none of the netlists is highest, while the usage_cost of a resource that was used by all netlists in the set is zero.

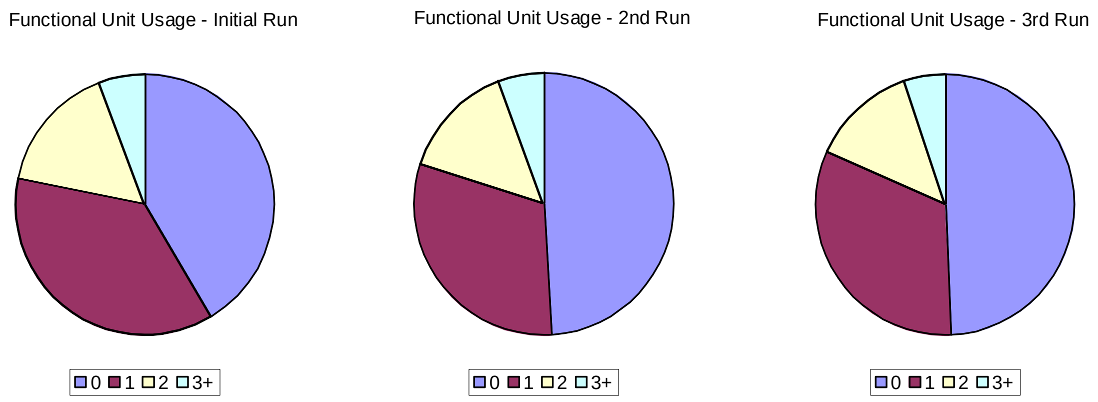


Fig. 4: A comparison of the number of functional units used by zero, one, two, and three or more netlists utilizing the RADAR, Image Processing, FIR, Matrix Multiply, and Sorters application domains.

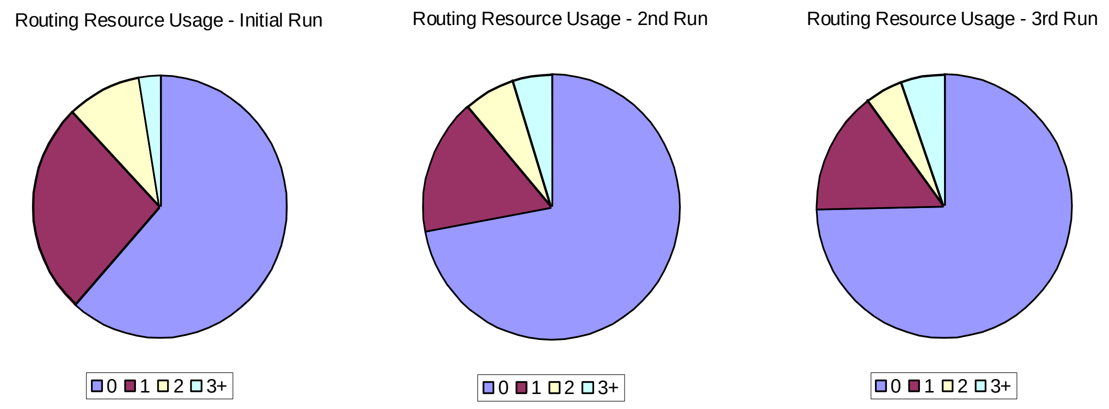


Fig. 5: A comparison of the number of routing resources used by zero, one, two, and three or more netlists utilizing the RADAR, Image Processing, FIR, Matrix Multiply, and Sorters application domains.

After completion of the first run on all netlists, a second run is commenced during which the netlists in the set are individually placed and routed again on the template architecture. However, for any given netlist, the cost of using a resource during the second run is influenced by the usage_cost of that resource. During placement, assigning a logic block to a functional unit penalizes the cost of the placement by a factor proportional to the usage_cost of the functional unit. The cost of assigning a logic block to a functional unit with high usage_cost is higher than the cost of assigning the logic block to a functional unit that has a relatively lower usage_cost. Similarly, while routing a netlist, the base cost of using a routing resource is proportional to the usage_cost of that resource. In general, if the usage_cost of a resource is high (i.e. the fraction of netlists that used this resource in the previous run was low), the place-and-route tool is influenced to select another resource with a lower usage_cost (i.e. a resource that was used by a large fraction of netlists during the previous run). Thus, during the second run, we try to direct the placement and routing of individual netlists toward using resources that were used heavily during the previous run. At the same time, we also attempt to drive down the fraction of netlists that use a resource to zero, so that we can eliminate that resource eventually. At the end of the second run, the usage_cost of each resource is again adjusted in a manner identical to that at the end of the first run, and a third run is begun. We are only reporting three runs, because the third run only deviates slightly from the second run in increasing the amount of resources that can be eliminated. Thus, any gains from subsequent runs are negligible. Once the three runs are completed, we have a list of the resources that can be eliminated from the template architecture. The results of the forced sharing after each of the three runs are shown in Fig. 4 and Fig. 5.

Equation (1) describes the variation in the usage_cost with the fraction of netlists that used that resource during the previous run.

$$\text{usage\_cost} = k*(1 - f)^2 \qquad (1)$$

In equation (1), f is the frequency of resources used. For placement, the value of k is chosen in a manner that ensures that the total usage_cost of a placement never exceeds 20% of the total cost of a placement. For routing, the value of k is selected so as to ensure that the usage_cost of a routing resource never exceeds 10% of the base cost of the routing resource.

C. *Reduction and Compaction*

Once the reduction list is generated, the final task is to actually edit the template in an automated fashion, followed by a compaction step to reduce the template size. To reduce the template, the layouts were automatically edited within the Cadence CAD tools. To achieve the required automation, Cadence SKILL code [14] is created by a SKILL code generator written in Perl. The SKILL code generator parses the reduction list and automatically creates a list of SKILL code reductions. Cadence SKILL Code enables interaction with the Cadence tools at a very low level. Therefore, each reduction that the subtractive method is able to perform has a corresponding SKILL routine that will implement the reduction on the template.

To remove as much overhead as possible we have implemented a wide range of reductions. First among them is the elimination of any unused cells (that is, complete RaPiD II tiles). The next reduction is the elimination of any functional units in any cell that are not needed. Next, we remove any of the bidirectional bus-connectors that are not needed in the interconnect. The final reduction is the removal of any unused wires. When an unused wire is removed, the corresponding transistors and programming bits in any muxes and drivers that the wire interacts with are also removed.

The arrays were then compacted by the Cadence compactor along the horizontal axis. Since some of the functional units are unaltered in template reduction, these units dictate the height of the array, and thus vertical compaction is not useful.

D. Template Reduction Summary

By leveraging a full custom layout structure, template reduction offers the potential to achieve very high quality implementations. If a generated architecture closely matches the full custom template, then Template Reduction will likely outperform any other approach. However, the approach also has significant limitations. First, if the desired architecture requires more resources than are present in the template, there is no way to add those resources. Also, while template reduction can make modifications scattered throughout the array, turning the layout into “swiss cheese”, the compactor will likely not be able to reduce the resulting area of the overall design. Thus, while we may get performance and power improvements by reducing capacitance in the array, we may get lower area improvements for architectures significantly different than the input template.

V. Circuit Generator Method

SRAM units in SoC designs are typically created by memory generators. One reason why memory generators are so efficient is their flexibility in tailoring the array to meet the design specifications, while at the same time minimizing area and maximizing performance. The Circuit Generator Method performs in much the same way (see Fig. 6). However, instead of providing just a single memory generator, to create a full RaPiD array we must provide a wide range of generators for all of the RaPiD components.

The current approach for the Circuit Generator Method is a mix of two types of generators. One type of generator enables the designer to modify certain parameters for units like the mux, demux, pipeline register, and bus connector. The other type of generator does not allow the designer to modify any parameters for units like memory blocks, the ALU, and the multiplier. This last type of generator is just placing the original full-custom circuits into the array, with modified interconnect for the inputs and outputs of the units that enable the unit to be tied into the overall array. This mix of approaches is necessary in our prototype system since it was not feasible to create true generators that extract regularity from all units.

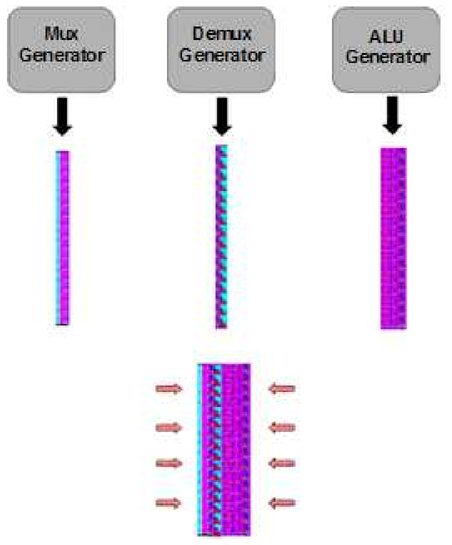


Fig. 6: The top figure shows the initial generation of circuits by three generators. Once the circuits have been generated, they are abutted together to create the functioning reconfigurable array, which is shown in the bottom figure.

The height of the generated circuits are loosely fixed based upon the number of buses and the number of bits on each bus needed to support the specified architecture. In essence, the minimum number of tracks is ascertained from the architecture description, which allows us to establish the height of the array used by all of the generators.

A. Approach

The first step in the generation of circuits is to receive the Verilog representation of the custom reconfigurable architecture from the Architecture Generator [5]. The Verilog is then parsed into separate generator calls, including any required parameters. For example, the following Verilog code:

bus_mux16_28data_reg_0_In(.In0(ZERO),….,Out(WIRE));

would be parsed so that the MUX generator would create a structure that contains sixteen 28-to-1 muxes that are stacked on top of each other with their control tied together.

After the Verilog has been parsed, the tool automatically generates the Cadence SKILL [14] code needed to implement the specified circuit. This is done by using Cadence SKILL code generators written in Perl. The Perl SKILL code generators call primitive Cadence SKILL functions that are able to automatically do simple tasks in Cadence, including opening, saving and closing files, drawing polygons in the layout, and instantiating cells. The code generators create circuits for all of the units needed to create the custom reconfigurable architectures, including muxes, demuxes, pipelined registers, bus-connectors, ALUs, multipliers, and SRAM blocks.

The generated circuits are targeted at the TSMC .18µm process. The height of the generated circuits is set by the number of routing tracks needed to support the number of bits per bus specified by the architectural description. In the TSMC .18µm process, a minimum size tristate inverter, laid out in a horizontal fashion, is equivalent in height to three routing tracks. By using metal four and metal six for horizontal routing, and layer five for vertical routing, three routing tracks are able to support a maximum of five bits, which is also shown in Fig. 7.

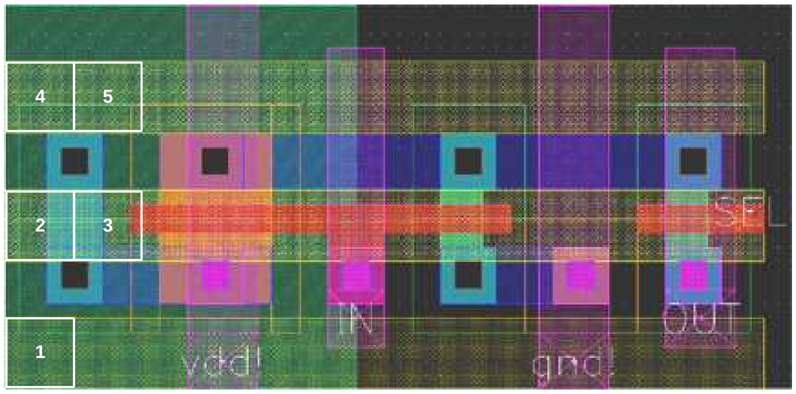


Fig. 7: One tristate inverter laid out in a horizontal fashion, which is the smallest building block of both the muxes and demuxes, has enough length in the vertical direction to support up to three horizontal routing tracks. Three routing tracks are able to support up to five bits via multiple metal layers. The metal lines pictured in the figure are on the fourth and sixth metal layers, of the six metal layer TSMC .18µm process.

Once the SKILL code has been generated that will produce the circuits, the next phase of circuit generation involves the creation of SKILL code that will automatically abut the generated circuits together. In the current version of the Circuit Generator Method, we are only dealing with circuits that utilize sixteen-bit functional units. Consequently, the routing complexity is greatly reduced, since the vertical distance between units is known in advance.

The last step is to actually run the Cadence SKILL code on Cadence to automatically create the units and to place the generated units together along the horizontal axis with the corresponding glue logic establishing connections between the various generated units. It should be noted that the Circuit Generator Method is highly automated. The designer only needs to provide the Verilog file, which the Circuit Generator Method uses to produce the mask layout with minimal user intervention.

Once the circuits are automatically generated by Cadence, wire lengths are extracted to tune the Place-and-Route Tool Generator’s delay estimator. The Place-and-Route tool maps the various netlists from the application domains onto the architecture to determine the delay numbers, using its detailed wire and functional unit models to compute these numbers. The next sections will go over the various generators in more detail.

B. Generators

We have created a generator for each of the components present in the RaPiD II template. The mux and demux generators create arbitrary interconnect structures tailored to the Architecture Generator’s requirements. The BC and register file generators similarly allow for arbitrary numbers of registers to be inserted, though currently we only use 1-delay and 3-delay structures. For the ALU, multiplier, and memory generators, we combine fixed functional units with an interface to the flexible interconnect structures.

The mux and demux generators are used to set the initial height of the reconfigurable arrays that the Circuit Generator Method creates. One goal of the Circuit Generators is to ensure that the capacitance and the delay of the muxes and the demuxes that are generated are as similar as possible to the full-custom muxes and demuxes used in the full-custom RaPiD II tile. Towards this end, all muxes and demuxes that are generated use the same full-custom tristate inverters that are used in the full-custom RaPiD II tile.

The process used in the generation of muxes and demuxes is modeled after the process used to create the full-custom muxes and demuxes in the full-custom RaPiD II tile, only our approach is automated. The decision to create a new row of mux bits (and thus increase the height of the mux) is based on the number of metal wires that can fit in the vertical area of one horizontally placed tristate inverter, which happens to be five bits. The formula to determine the number of rows is max(1,floor((n+1)/5)). Fig. 8 shows the configurations of muxes from 4 bits to 20 bits, in 4 bit increments. When minimizing wasted area, the most efficient structures are muxes or demuxes of bit size p, where p mod 5 is equal to zero, since each horizontal tristate inverter is three tracks, or 5 bits, high. Structures with size q, where q mod 5 is equal to one, are the most inefficient (the 16:1 mux case is an example).

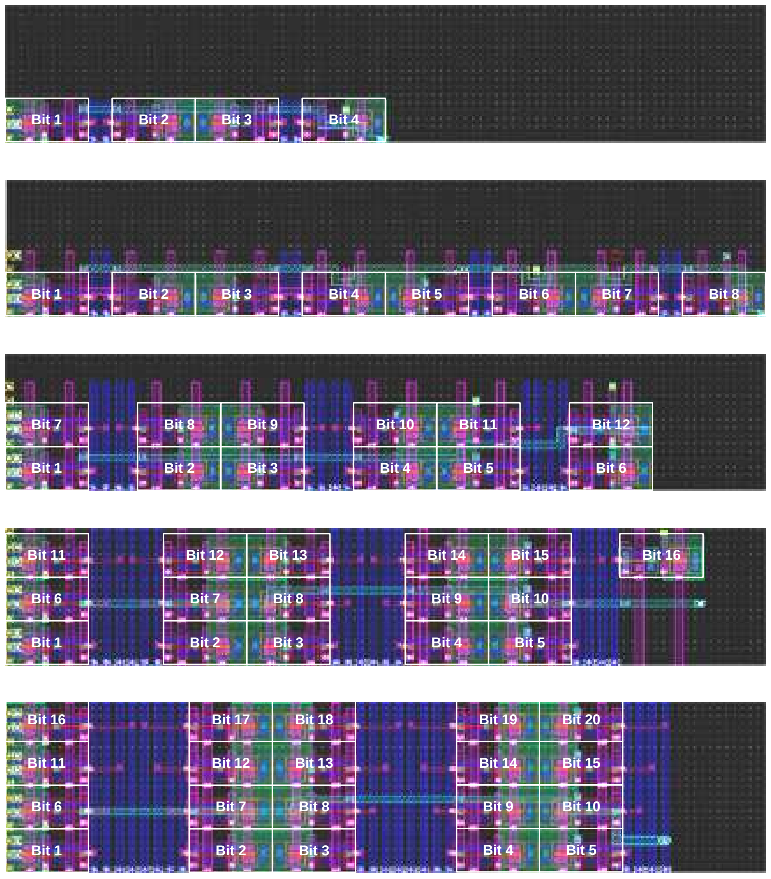


Fig. 8: Various configurations of muxes based upon the number of bits, and the number of routing tracks. The top figure is a 4 bit mux, followed by 8, 12, 16, and 20. All of the figures are to scale. Notice the increase in the width of the control routing channel as the number of tristate rows increases, and the wasted space in the 16 bit mux.

C. *Circuit Generators Summary*

The Circuit Generator Method is able to leverage the regularity that exists in FPGA designs in a method very similar to the creation of memory arrays by memory generators. This method is able to create circuits that perform better than that of the RaPiD II full custom fixed tile, as long as the specified architecture does not require functional units or routing resources that do not have a corresponding generator. However, it can require significantly more effort to create a flexible circuit generator than implementing a single instance of a circuit type for a fixed architecture.

VI. Standard Cell Method

Instead of creating a new flow to implement FPGAs, we can leverage standard ASIC flows. If we take the Verilog produced by the Architecture Generator and send it to a standard cell layout tool, we can get an implementation of any architecture desired. However, even better results can be achieved based on a simple observation: FPGAs are composed of a relatively small number of basic elements, so there are significant benefits to providing optimized implementations of these basic elements within the standard cell library. Thus, an FPGA-optimized standard cell library [7] would consist of optimized cells containing typical FPGA components such as LUTs, SRAM bits, muxes, and demuxes.

A. Approach

To retain as much flexibility as possible in our standard cell implementation, behavioral Verilog representations were created for all of the RaPiD components. The Architecture Generator used these behavioral components as leaf cells when it generated Verilog versions of RaPiD that support a particular application domain. Synopsys was used to synthesize the behavioral Verilog to produce structural Verilog that has been mapped to our standard cell library [5]. This gives us the ability to swap out standard cell libraries, since we would only need to re-synthesize the behavioral Verilog with a new library file generated for the new standard cell library. The ability to easily and efficiently use different libraries is a very powerful feature of the Standard Cell Method. It enables designers to choose different libraries that provide different capabilities, such as lower power, smaller area, or higher performance.

Silicon Ensemble was used to place and route the cells. Silicon Ensemble is part of the Cadence Envisia Tool Suite, and is capable of routing multiple layers of metal, including routing over the cells. We used the NCSU TSMC 0.18µm design rules for all layouts created in Cadence.

The choice of a standard cell library was based upon the need to find an industrial strength library that has been laid-out for the TSMC 0.18µm process. Unfortunately, we were not able to find a library targeted at the TSMC 0.18µm process, but we were able to find two libraries targeted at the TSMC 0.25µm process, namely the VTVT standard cell library and the Tanner standard cell library [22]. We chose the VTVT standard cell library, which was available from the Virginia Tech VLSI for Telecommunications group [15, 16], over the Tanner standard cell library. We arrived at this decision because the VTVT library also included Synopsys synthesis files, VHDL simulation libraries, and LEF files for Silicon Ensemble, while the Tanner library did not. The VTVT library was then migrated to the TSMC 0.18µm process.

B. Standard Cell Summary

The greatest strength of this method is its high level of flexibility. This method is always capable of producing a result, even when the other methods fail. Also, with a wider range of libraries, including libraries optimized for power, performance, and area, this method has a lot of potential for improvement. However, the overheads of standard cells vs. full custom design impose a significant penalty to this approach.

VII. Testing Framework

A. Application Domains

To evaluate the automatic generation of domain-specific reconfigurable circuits we used thirteen different application domains. All of the netlist sets that make up each application domain have been compiled using the RaPiD compiler [17]. Two of the netlist sets, RADAR and Image, are complete applications. The RADAR application is used to observe the atmosphere using FM signals, while the Image application is a minimal image processing library. The other eleven applications represent the cross product of two domains, like the Image and RADAR application, domains of similar netlists, like FIR, Matrix Multiply, and Sorters, or reduced domains, like Reduced Image 1 through 4 and Reduced RADAR 4 through 6. All of the application domains and their member netlists are shown in Table 1. It should be noted that only five of the thirteen application domains are run on the circuits created by the Template Reduction due to limitations of the toolset.

TABLE I
APPLICATION DOMAINS

Application Domain	Member Netlist	Percent Utilization
Reduced RADAR 6	decnsr, psd	20.92
FIR	firsm2, firsm3, firsm4, firsymeven	28.90
Reduced Image 1	firtm_2nd, matmult	29.07
Reduced Image 2	1d_dct40, fft16_2nd, matmult	29.15
Sorters	sort_g, sort_rb, sort_2d_g, sort_2d_rb	32.12
Image	1d_dct40, firtm_2nd, fft16_2nd, matmult	37.05
Matrix Multiply	limited, matmult, matmult4, vector	37.43
Image and RADAR	1d_dct40, fft16_2nd, firtm_2nd, matmult	41.21
Reduced RADAR 4	decnsr, fft16_2nd	50.88
RADAR	decnsr, fft16_2nd , psd	52.79
Reduced Image 4	1d_dct40, fft16_2nd	52.82
Reduced RADAR 5	fft16_2nd, psd	53.54
Reduced Image 3	1d_dct40, fft16_2nd, firtm_2nd	60.18

The benchmark application domains and their corresponding member netlists. The applications are ordered in the table by their percent utilization, from lower to higher values.

B. Percent Utilization

The netlists in Table 1 are ordered by their percent utilization. Percent utilization is a measure of the resources that an array of full-custom fixed tiles would need to support a particular application domain. Resources include multipliers, ALUs, wires, bus connectors (BC), routing muxes and demuxes, data and pipeline registers, and memories. For example, an application domain that requires half of the resources provided by the full-custom fixed tile would fall at 50% utilization. The percent utilization calculated in Table 1 was generated using the RaPiD II fixed tile. To actually calculate the percent utilization we use the place-and-route tool to map the application domain onto an array of RaPiD II tiles. The length of the RaPiD II array is determined by iteratively adding another fixed RaPiD II tile to the array until the mapping is successful.

Once the array length is set, we look at all of the resources that are used by the application domain mapping. In essence, if only one of the netlists in an application domain uses any resource in the array, then that resource is part of the percent utilization for that application domain. We divide the sum of the area of all of the resources needed to support an application domain by the total area of the RaPiD II array to arrive at the value of the percent utilization for an application on a particular array of fixed tiles. In essence, the percent utilization metric is a measure of how well a fixed tile is tuned to a particular application domain. If the percent utilization of an application domain is very high, then the resource mix of the fixed tile is well suited for that application domain. We use percent utilization here because we intuitively felt that the quality of the various implementation strategies would be highly correlated to percent utilization.

VIII. Results

A. Area and Delay Evaluation

To evaluate the three methods, we are concerned with two metrics, namely the overall area of the generated circuits, and the delay of the circuit when each of the application domains are mapped, as evaluated by the static timing analyzer contained in the place and route tool. The area of the generated circuits is evaluated by measuring the area of the layout that is generated by each of the methods. This is a straightforward process, since all three methods generate circuits using the NCSU CDK [18] for the TSMC .18µm process.

The delay of each circuit is evaluated by using the Totem place and route tool to map, or bind, each of the netlists in the application domain onto the generated circuit. The place and route tool is then able to determine the delay of the mapped netlists on the circuit by performing static timing analysis of the critical path. The place and route tool is aware of the critical path of the netlist since it places and routes all of components and the signals that constitute a netlist. The models used in the static timing analysis were created by running spice simulations of all of the RaPiD components. It should be noted that this version of the place and route tool is unable to retime signals. Therefore, any delay numbers generated by the place and route tool should only be used for relative comparisons of the three methods.

B. Area Comparison

The area of the circuits created varies greatly, depending on both the specified application domain and the proposed method. The graph shown in Fig. 9 presents the three methods, along with the original full custom RaPiD II tile. The x-axis is percent utilization, which is an indication of the amount of resources that an application domain would require to run on the full custom RaPiD II template. The y-axis is the area normalized to the full custom RaPiD II template. The points for the Circuit Generator Method are an average of the AML, AMO, and GH Architecture Generators, detailed further in [19]. The points for the Standard Cell Method are an average of the generic VTVT standard cell library and a modified VTVT standard cell library targeted at FPGAs, detailed further in [7].

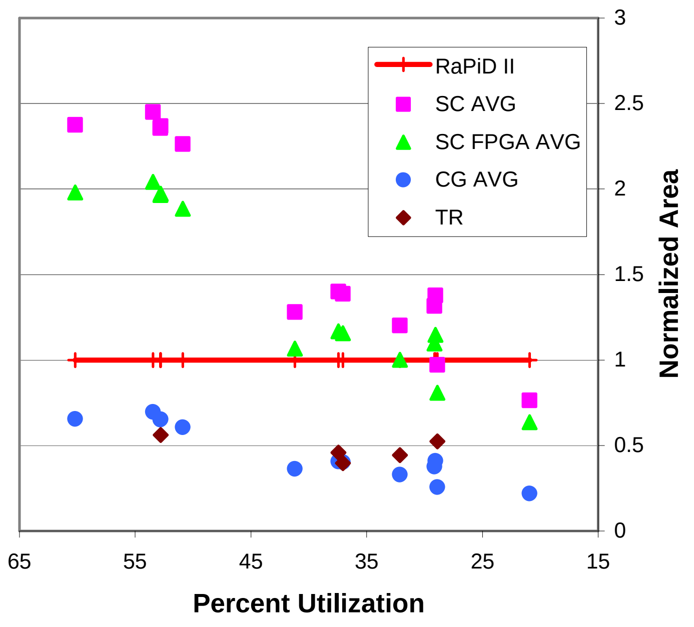


Fig. 9: Area comparison of the circuits created to support the benchmark sets.

It is evident from Fig. 9 that the Template Reduction and the Circuit Generator Methods create circuits that are roughly comparable to each other in area. The Template Reduction Method is more efficient when the percent utilization is high, while the Circuit Generator Method is more efficient when the percent utilization is lower. This is a strong showing for the Circuit Generator Method, since it is creating circuits from scratch that can compete with reduced full custom circuits. These results may be an indication that the compaction of circuits is less efficient as the percent utilization drops. This is because the circuits created by the Template Reduction Method are becoming less and less regular. The Circuit Generator Method is not affected by this, since it is creating circuits from the ground up, as opposed to reducing existing structures.

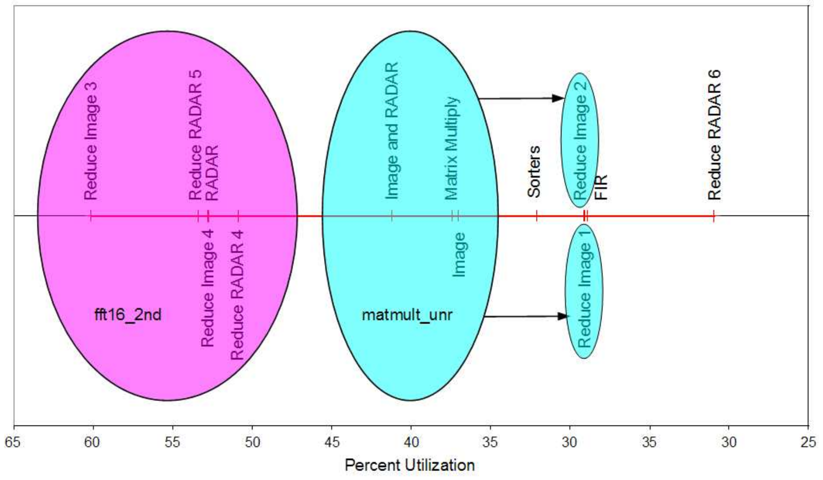


Fig. 10: The thirteen application domains ordered along the horizontal axis by percent utilization. The fft16_2nd and the matmult netlists dominate ten of the thirteen application domains, which is indicated by the red and blue circles.

A noticeable feature of Fig. 9 is the fact that the benchmarks seem to cluster into two groups, one group that has a high percent utilization, and another that has a low percent utilization. This is due to the domination of certain netlists in each application group, which can be seen more clearly in Fig. 10. The first cluster is dominated by the fft16_2nd netlist, and the second cluster is dominated by the matmult netlist.

C. Delay Comparison

Fig. 11 shows the delay of each benchmark set after it has been normalized to the delay of the fixed RaPiD II tile, where lower delay indicates a higher quality circuit. The delay results of the application domains on the circuits created by the various methods, are more scattered and do not show the same level of improvement as the area improvements. The Standard Cell Method cannot overcome the overhead associated with this method. Therefore, the circuits created by the Standard Cell Method never perform better than the full-custom RaPiD II tile, and are approximately 3.10 times to 1.40 times slower than the other three methods. The shortcomings of the Standard Cell Method are even more magnified when it is pointed out that the full-custom RaPiD II tile is unaltered, and therefore capable of handling application domains that require 100% utilization, while the circuits generated by the Standard Cell Method have been reduced, and are therefore less capable.

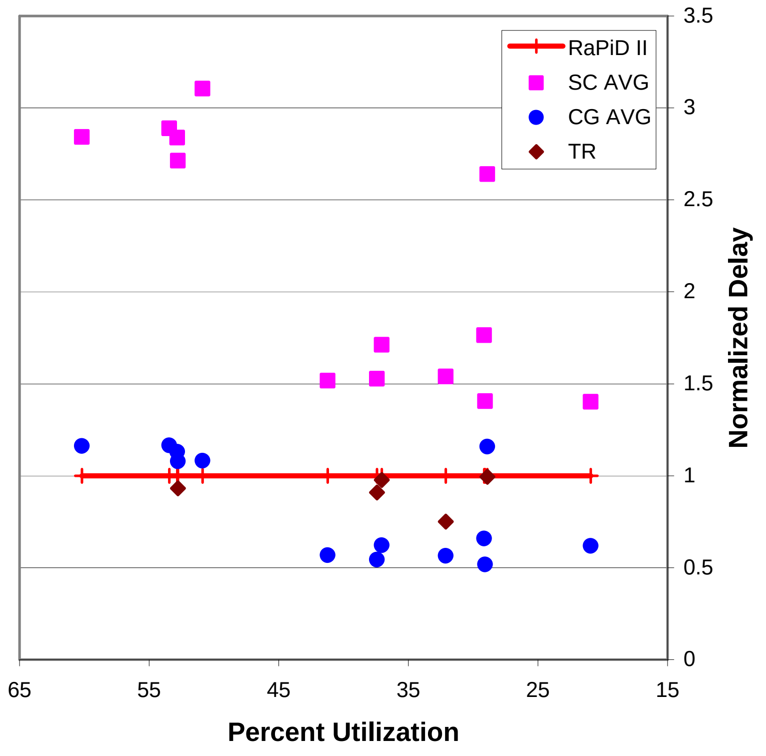


Fig. 11: Delay comparison of the benchmarks run on the full-custom RaPiD II tile, and the Template Reduction, the Circuit Generator, and the Standard Cell Methods. The y-axis is the delay normalized to the RaPiD II cell, while the x-axis is the percent utilization.

The Circuit Generator and the Template Reduction Methods produce circuits that have an average delay improvement of approximately 16% to 9% over the benchmarks run on the full custom RaPiD II tile. When the percent utilization is high, the Template Reduction Method appears able to produce higher performing circuits than the Circuit Generator Method. When the percent utilization is low, the Circuit Generator Method is able to produce circuits that perform better than the Template Reduction Method. Once again, it should be noted that only five of the thirteen application domains are run on the circuits created by the Template Reduction due to limitations of the toolset.

Once again, as seen in Fig. 9, the benchmarks are clustered into two groups depending upon which netlists are dominating within the application domains. Another feature that can be seen in the graph is the fact that the performance of the benchmarks increases as percent utilization decreases. This is an overall trend with some outliers, and these results are highly dependent on the efficiency of the P&R tool. The most noticeable outlier is the FIR application domain. Two netlists dominate the performance of this application group, namely the firsm3 and the firsymenven, causing it to perform poorly.

IX. Conclusions

The focus of this work has been the automation of the layout portion of the Totem design flow. Towards this end, we have implemented the VLSI layout generator, which automates the creation of mask ready layouts from the circuit descriptions provided by the Architecture generator. The VLSI layout generator consists of three methods of automating the layout process: Template Reduction, Circuit Generators, and Standard Cell generation.

The Template Reduction Method is able to leverage full custom designs, while still removing any resources that are not needed to support the specified application domain. This enables the Template Reduction Method to create circuits that perform at or better than that of the initial full-custom template, with an average area decrease of approximately 48% and an average delay decrease of approximately 9%.

One of the drawbacks associated with the Template Reduction Method is its reliance on the existence of a feature-rich macro cell that is a superset of the specified application domain. The Circuit Generator Method is able to produce efficient circuits in both area and performance in an additive fashion, while removing the need for feature-rich templates. Circuits created by the Circuit Generator Method are approximately 46% smaller and 16% faster than the full custom RaPiD II tile.

The Standard Cell Method, while extremely flexible, was able to produce competitive circuits with regard to area, only when the resources were reduced to approximately 25% of the full-custom template. Unfortunately, the Standard Cell Method was never able to produce a circuit that performed better than the full-custom RaPiD II template. The Standard Cell Method is capable of producing circuits with areas ranging from 2.45 times larger to 0.76 times smaller than comparable full-custom circuits. The Standard Cell Method can also produce circuits with delays ranging from 3.10 times to 1.40 times longer than comparable full-custom circuits. However, the strength of the Standard Cell Method lies in its ability to produce a circuit for any application domain, even when the Template Reduction and Circuit Generator Methods fail.

Choosing an appropriate method is based on many factors. If a robust template along with suitable reductions exists, then the Template Reduction Method is quite capable of producing competitive circuits. While this suggests that the Template Reduction Method should be competitive with the other methods, we feel that it is the weakest method. The Template Reduction Method is too inflexible. Its reliance on templates is its biggest liability, since the generation of even one template is a costly endeavor. In future systems, the designer might want to specify certain design constraints, like low power, small area, or high performance. This implies that a single template will not be able to cover all of these design areas, forcing the Totem Project to have at its disposal multiple templates that have been laid out with different design goals in mind. This would entail considerable effort, thus defeating the purpose of the Totem Project to automatically provide custom reconfigurable circuits in a timely fashion.

Another problem with the Template Reduction Method is its propensity for causing errors in circuits. Of the three methods, the Template Reduction Method was the most error prone method, and was the most complicated method to implement and debug. In essence, the Template Reduction Method is manipulating full-custom circuits at the lowest level. This can lead to numerous DRC errors, including n-implant, p-implant, and well errors. In addition, when manipulating full-custom designs in this manner, the Template Reduction Method is changing the dynamics of the circuits in potentially unforeseen ways. For example, the transistors in a full-custom circuit are sized to ensure that they are capable of driving their load in an efficient manner. By cutting out transistors, wires, etc, the Template Reduction Method is altering those loads, which can lead to a poorly performing circuit.

The Circuit Generator Method is able to leverage the regularity that exists in FPGAs when creating RaPiD-like structures. It can create structures that are more efficient than the Template Reduction Method, while not being bound to a particular template. In addition, the Circuit Generator Method is an additive method. Therefore, this method is less error prone than the Template Reduction Method since we are not cutting low-level components out of full-custom circuits.

However, the Circuit Generator Method has problems of its own. The creation of a wide range of generators can be as costly a proposition as creating a wide range of templates. But, to improve the Circuit Generator Method, providing a wide range of different types of generators is critical. To increase the quality of the circuits that the method creates, all of the generators should be able to handle a wide range of parameters. For example, in the current implementation of the Circuit Generator Method, the generators that create the functional units are unable to change the bit width of units that they create. However, it has been shown in previous work [20] that the largest impact on area is achieved through reducing the overall bit-width of the device that is created. Therefore, the creation of generators that are able to modify the bit width of the functional units could drastically increase the ability of the Circuit Generator Method to create higher quality circuits. Finally, if the Circuit Generator Method needs to create circuits that are targeted at low power, high performance, or small area, even more types of generators will be needed.

This leads us to the Standard Cell Method. As anticipated, the Standard Cell Method has inherent inefficiencies that it must overcome to become competitive with the other two methods. However, it is extremely flexible and is able to create a circuit in any circumstance. This is important because the overall goal of the Totem Project is to support any designer defined application domain. To build upon this flexibility, the ability to utilize a wide range of industrial strength standard cell libraries is needed. With a wide range of libraries, the designer could select the library most suited to the specifications of their design. Specifications could include higher performance, lower power, or smaller area, and if there was a corresponding library, the Standard Cell Method has the potential to create high quality circuits with a minimal amount of effort.

X. Acknowledgments

The authors would like to thank the RaPiD group, especially Carl

Ebeling and Chris Fisher, for the RaPiD I layout used in this research. We also are indebted to Larry McMurchie for support on the Cadence tool-suite. This work was funded in part from grants from NSF and NASA. Shawn Phillips was supported in part by a MIT Lincoln Labs Research Fellowship. Scott Hauck was supported in part by an NSF CAREER award and an Alfred P. Sloan Research Fellowship.

REFERENCES

[1] Darren C. Cronquist, Paul Franklin, Chris Fisher, Miguel Figueroa, and Carl Ebeling, “Architecture Design of Reconfigurable Pipelined Datapaths”, *Twentieth Anniversary Conference on Advanced Research in VLSI*, pp 23-40, 1999.

[2] C. Ebeling, D. C. Cronquist, P. Franklin, “RaPiD – Reconfigurable Pipelined Datapath”, *6th Annual Workshop on Field Programmable Logic and Applications*, 1996.

[3] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Taylor, “PipeRench: An Architecture and Compiler for Reconfigurable Computing”, *IEEE Computer*, 2000.

[4] A. Abnous and J. M. Rabaey, "Ultra-low-power domain-specific multimedia processors," *Proc. of IEEE VLSI Signal Processing Workshop*, Oct. 1996.

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

[6] 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.

[7] 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.

[8] A. Sharma, C. Ebeling, S. Hauck, "PipeRoute: A Pipelining-Aware Router for FPGAs", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 68-77, 2003.

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

[10] S. Phillips, S. Hauck, "Automating the Layout of Reconfigurable Subsystems for Systems-on-a-Chip", *IEEE Symposium on FPGAs for Custom Computing Machines Conference*, 2005.

[11] C. Sechen, VLSI Placement and Global Routing Using Simulated Annealing, Kluwer Academic Publishers, Boston, MA: 1988.

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, MA, Prim’s algorithm, pp 505-510, 1990.

[13] A. Sharma, “Development of a Place and Route Tool for the RaPiD Architecture.” M.S. Thesis, University of Washington, Dept. of EE, 2001.

[14] Cadence Design Systems, Inc., “Openbook”, version 4.1, release IC 4.4.5, 1999.

[15] J. B. Sulistyo, J. Perry, and D. S. Ha, "Developing Standard Cells for TSMC 0.25um Technology under MOSIS DEEP Rules", Department of Electrical and Computer Engineering, Virginia Tech, Technical Report VISC-2003-01, November 2003.

[16] Jos. B. Sulistyo and Dong S. Ha, "A New Characterization Method for Delay and Power Dissipation of Standard Library Cells", VLSI Design 15 (3), pp. 667-678, 2002.

[17] D. C. Cronquist, P. Franklin, S.G. Berg, C. Ebeling, "Specifying and Compiling Applications for RaPiD", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[18] Shaffer, Stanaski, Glaser, and Franzon, “The NCSU Design Kit for IC Fabrication through MOSIS", 1998 International Cadence User Group Conference in Austin, Texas.

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

[20] S. Phillips, “Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip.” M.S. Thesis, Northwestern University, Dept. of ECE, July 2001.

[21] K. Eguro, S. Hauck, “Resource Allocation for Coarse-Grain FPGA development”, IEEE TCAD, Vol 24, No 10.