Automatic Design of Area-Efficient Configurable ASIC Cores

Katherine Compton
University of Wisconsin
Madison, WI
kati@engr.wisc.edu

Scott Hauck
University of Washington
Seattle, WA
hauck@ee.washington.edu

Abstract

Reconfigurable hardware has been shown to provide an efficient compromise between the flexibility of software and the performance of hardware. However, even coarse-grained reconfigurable architectures target the general case, and miss optimization opportunities present if characteristics of the desired application set are known. Efficiency can be increased by restricting the structure to support a class or a specific set of algorithms, while still providing flexibility within that set. By generating a custom array for a given computation domain, we explore the design space between an ASIC and an FPGA. However, the manual creation of these customized reprogrammable architectures would be a labor-intensive process, leading to high design costs. Instead, we propose automatic reconfigurable architecture generation specialized to given application sets. This article discusses configurable ASIC (cASIC) architecture generation that creates hardware on average up to 12.3x smaller than an FPGA solution with embedded multipliers and 2.2x smaller than a standard cell implementation of individual circuits.

1. Introduction

While FPGAs and reconfigurable systems have been effective in accelerating DSP, networking, and other applications [1], the benefit is in many cases limited by the fine-grained nature of many of these devices. Common operations such as multiplication and addition are more efficiently performed by coarse-grained components. A number of reconfigurable systems have therefore been designed with a coarse-grained structure. These structures target the general case, attempting to fulfill the computation needs of any application that may be needed. However, because different application types have different requirements, this creates a large degree of wasted hardware (and silicon area) if the applications run on the system are constrained to a very limited range of computations. Unused logic and programming points occupy valuable area and can slow down computations, contributing to the overhead of the device without providing benefit. While the flexibility of general-purpose hardware has its place for situations where the computational requirements are not known in advance, frequently specialized on-chip hardware is used to obtain greater performance for a specific set of compute-intensive calculations.

More customized reconfigurable hardware has been examined in an effort to reduce the amount of "useless" hardware and increase the efficiency of the computing device. Architectures such as RaPiD [2], PipeRench [3], and Pleiades [4] have made progress in this direction by targeting multimedia and DSP domains. Commercial devices such as Morpho [5]

and Stretch [6] are also more coarse-grained than traditional FPGAs. The Totem Project[1] [7][8][9][10][11][12] takes this one step further, allowing the user to select the computation domain (such as signal processing, encryption, scientific data processing, or a subset of applications within one of these domains) by providing representative circuits to an architecture generator. This concept was also proposed by the RaPiD group. These customized devices are ideal for situations where the category of computation is known, but the individual circuit set is either not completely known or not fixed. However, if we know the actual circuits to be computed, we can create an even more specialized design called a *configurable ASIC* (cASIC).

cASICs are intended as accelerators on Systems-on-a-Chip (SoCs). While the circuit set that they can implement is fixed, they are different from traditional ASICs in that some level of hardware programmability is retained. This programmability allows the accelerator to implement different circuits from its circuit set at different points of time through run-time-reconfiguration. Because the number of reconfiguration points is limited, not only is performance expected to be close to that of an ASIC, but the circuit will be significantly smaller than an FPGA implementation. Furthermore, because we are reusing one set of hardware to implement multiple circuits, it will be smaller than the sum of the ASIC areas of the individual circuits in almost all cases.

The area benefit of cASICs is critical. For many high-performance applications, special hardware accelerators can become quite large. If each of the desired accelerators were synthesized separately, this could lead to an unreasonably large (and expensive) chip. Allowing

---

[1] The Totem Project is a large multi-person project with numerous publications. The related theses are the most comprehensive documents. Totem conference and journal publications can be found at the referenced website.

the accelerators to share hardware makes their use much more attractive to the SoC designers. This will encourage the use of specialized accelerator circuits, leading to devices with higher performance and lower power consumption than ones that rely on a microprocessor for all computations. Battery-powered devices in particular would benefit from the low-power execution.

Specialized cASICs, while beneficial in theory, would be impractical in practice if they had to be created by hand for each group of applications. Each of these optimized reconfigurable structures may be quite different, depending on the application set desired. One could manually specify resource sharing in an HDL description or in circuit layout. Unfortunately, this would contribute significantly to the design costs of the hardware. Synthesis tools do have some ability to find resource sharing opportunities, but unfortunately this would still require the designer to merge the needed circuits into a single design. Simply instantiating the circuits in a larger framework generally prevents the synthesis tools from finding sharing opportunities. Flattening the circuit can help, but it will increase the synthesis time dramatically, and even so, many sharing opportunities may be overlooked by tools optimized for sharing within smaller areas of a single circuit. Research into automated cASIC sharing is therefore essential in order to decrease the cost of customized architecture development.

The Totem Project focuses on the automatic generation of customized reconfigurable architectures. While most of the Totem Project research focuses on more flexible architecture design, this article describes our work towards a cASIC generator including experiments and data beyond the initial preliminary results [13]. This generator takes as input a set of RaPiD-format netlists and creates as output an architecture capable of implementing any of the provided circuits. Like RaPiD, the architecture is a 1D bidirectional datapath composed of coarse-grained

computational units and word-size busses. Although the current version of the cASIC generator creates RaPiD-style datapaths, many of the techniques that we will discuss can apply more generally as well. This possibility is discussed in more depth in section 5. As we will show later in this article, the generated cASIC architectures are significantly smaller than required by a traditional FPGA implementation, and can even be under half the size of standard cell implementations of the individual circuits.

2. Background

Current efforts in the Totem Project focus on coarse-grained architectures suitable for compute-intensive application domains such as digital signal processing, compression, and encryption. The RaPiD architecture [2][14] is presently used as a guideline for the generated architectures due to its coarse granularity, one-dimensional routing structure, and compiler. Coarse-grained units match the coarse-grained computations currently targeted. The one-dimensional structure is efficient for many DSP applications, but also simplifies the architecture generation process significantly. Future work in the Totem Project focuses on the two-dimensional case, discussed in part in section 5. Finally, a compiler [15] for this system is already in place, which aids in the development of application circuits for Totem. The compiler takes a description written in RaPiD-C and creates circuit netlists suitable for RaPiD or Totem implementation.

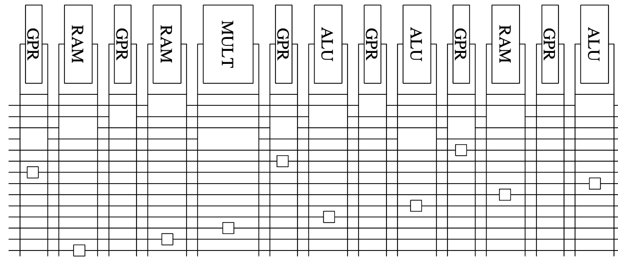


Figure 1: A single cell from the RaPiD architecture [14][16]. A full architecture is composed of multiple cells laid end-to-end.

The RaPiD architecture is composed of a set of repeating cells (Figure 1) tiled horizontally. 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 choose between the signals of each routing track. Each component also has a demultiplexer on each of the outputs that allow the unit to directly output to any of the routing tracks. Inputs are on the left side of a unit, while the outputs are on the right side of the unit. Global inputs also reside on the 1D datapath at the ends of the architecture.

The routing architecture is a one-dimensional segmented design, where each track is word-width. The top routing tracks are local routing tracks that contain short wires for fast short-distance communication. The bottom ten tracks provide longer distance routing, allowing wires to be connected to form longer wires. The bus connectors are shown as small squares in the figure, and provide optional pipeline delays to mitigate the delay added through the use of longer wires and more routing switches.

3. cASIC Generation

While the flexibility of traditional FPGA structures is one of their greatest assets, it is also one of their largest drawbacks — greater flexibility leads to greater area, delay, and power overheads. Creating customized reconfigurable architectures presents the opportunity to greatly reduce these overheads by discarding excess flexibility. This article discusses taking this idea to the extreme end of the spectrum – removing *all* unneeded flexibility to produce an architecture as ASIC-like as possible. We call this style of architecture "configurable ASIC", or cASIC.

Like RaPiD [2][14], the cASIC architectures we create are very coarse-grained, consisting of optimized components such as multipliers and adders. Unlike RaPiD, cASICs do not have a highly flexible interconnection network — the only wires and multiplexers available are those which are required by the netlists. This is because cASICs are designed for a specific set of netlists, and are not intended to implement netlists beyond the specification. In fact, unless a circuit is from the specified input set, or extremely similar to one of the circuits in the set, it unlikely to be implementable in the generated hardware. This hardware is intended to be optimized to the exact circuits in the specification, and to be an alternative to a set of separate ASIC circuit structures.

Hardware resources are still controlled in part by configuration bits, though there are much fewer present than in an FPGA. In particular, they are used to control any multiplexers on the inputs of logic units, as well as ALU modes. These configuration bits allow for hardware reuse among the specification netlists. Each netlist in the specification is implemented in turn by programming these bits with the appropriate set of values.

The cASIC architecture generation occurs in two distinct phases. In the logic stage of the generation the computation needs of the algorithms are determined, the computational components (ALUs, RAMs, multipliers, registers, etc) are created, and the physical elements are ordered along the one-dimensional datapath. Also, the netlist instances must be bound to the physical components. In the routing stage, wires and multiplexers are created to connect the different logic and I/O components.

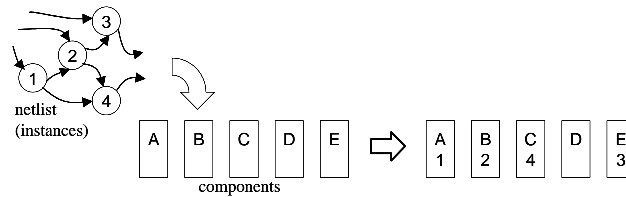
3.1. Logic Generation

The generation of logic structures for cASIC architectures involves first determining the type and quantity of functional units required to implement the given netlists. Because the ability to reuse hardware is a key feature of reconfigurable computing, maximum hardware reuse between netlists is ensured. The minimum number of total logic units is chosen such that any one of the netlists given as part of the architectural specification can operate in its entirety. In other words, unit use within a netlist is not modified or rescheduled. Therefore, if netlist A uses 12 multipliers and 16 ALUs, while netlist B uses 4 multipliers and 24 ALUs, a cASIC architecture designed for these two netlists would have 12 multipliers and 24 ALUs. If a designer were to require some flexibility in the design beyond the minimum, additional units could be added. However, this would defeat the purpose of the minimalist cASIC. Techniques from the Totem Project to generate more flexible domain-specific reconfigurable hardware have been published elsewhere [17].

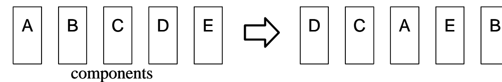
After the unit quantities and types have been selected, they must be arranged along the horizontal axis. A good ordering will ensure that the number of signals passing through any one

vertical cut of the architecture is kept low, which reduces the area consumed by the routing structures. Similarly, units communicating with one another should be located in close proximity to reduce the delay on the wires between them. Therefore, the best physical ordering of the units depends on the communication between them. The communication needs between physical units, however, depend on how the netlists are implemented on that hardware. Although this specific work targets a 1D architecture, we will later discuss extensions to support 2D designs.

Before discussing the positioning of logic resources further, we must define some key terminology. The architectural *components* represent physical structures to be created in silicon. These differ from netlist *instances*, which are implemented by the physical components. A netlist instance represents a "need" for a given type of computation at a given point of the circuit. In traditional FPGAs, the LUTs are the physical components, while the netlist instances are low-level gates or small logic functions. In the Totem Project, coarser-grained architectures and netlists are currently used. For example, a multiply-accumulate netlist contains a multiplier instance followed by adder and register instances. These instances must be implemented by the appropriate type of physical components in the hardware.



(a)



(b)

Figure 2: Binding vs. physical moves. (a) *Binding* assigns instances of a netlist to physical components. (b) *Physical moves* reposition the physical components themselves.

There may be multiple units appropriate for a circuit instance, in which case the instance must be matched to a specific physical unit. When using traditional FPGAs, this matching is referred to as placement or binding. For this work, the terms *binding* or *mapping* are used to describe the process of matching an instance to a component. A *physical move* describes the act of assigning a physical location to a physical component. Figure 2 illustrates the difference between binding and placement. Using this terminology, traditional synthesis for FPGAs requires only bindings, whereas synthesis for standard cells involves only physical moves.

Reconfigurable architecture generation is a unique situation in which both binding and physical moves must be considered. The locations of the physical units must be known in order to find the best binding, and the binding must be known to find the best physical moves. Since these processes are inter-related, both binding and physical moves are performed simultaneously in this work. The term *placement* in Totem architecture generation refers to the combined process of determining a binding of netlists to units and determining physical locations for the units.

Placement during cASIC generation utilizes a simulated annealing algorithm, which is commonly used in FPGA placement (binding) to assign netlist instances to physical computation units, and standard cell placement to determine locations for actual physical cells. This algorithm operates by taking a random initial placement of elements, and repeatedly attempts to move the location of a randomly selected element. The move is accepted if it improves the overall cost of the placement. To avoid settling in a local minima of the placement space, moves that do not improve the cost of the placement are sometimes accepted. The probability of accepting a non-improving move is governed by the current "temperature". At the beginning of the algorithm, the temperature is high, allowing a large proportion of bad moves to be accepted. As the algorithm progresses, the temperature decreases, and therefore the probability of accepting a bad move also decreases. At the end of the algorithm almost no bad moves are permitted.

In the Totem architecture generation, simulated annealing performs both the binding and physical moves simultaneously. Therefore, a "move" can be either of these two possibilities – either rebinding a netlist computational instance from one physical unit to another compatible physical unit, or changing the position (ordering) along the 1D axis of a computational component.

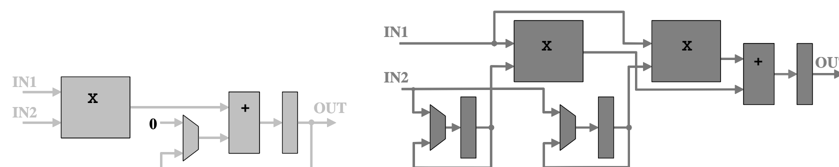


Figure 3: Two different example netlists that could be used in architecture generation. The light netlist performs a multiply-accumulate (MAC), while the dark netlist is a 2-tap FIR filter. These two netlists are used in the example placement process given in the next few figures.

In order to create a single architecture optimized for all of the target netlists, we perform placement and binding of all netlists simultaneously using a modified simulated annealing algorithm. The instances of each netlist are arbitrarily assigned initial bindings to physical components, which are ordered arbitrarily along the 1D axis. An example initial placement created for the two netlists presented in Figure 3 appears in Figure 4. Next, a series of moves is used to improve the placement. However, for cASIC generation, there are two different types of moves that can be attempted within the simulated annealing algorithm: rebinding and physical moves. The probability of attempting a rebinding when making a simulated annealing move is equal to the number of netlist instances divided by the sum of the netlist instances and physical components. In other words, the ratio of the physical moves to the binding moves is the same as the ratio of physical components to netlist instances.

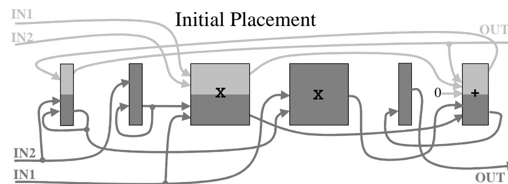


Figure 4: The initial physical placement and bindings of an architecture created for the netlists of Figure 3. Color shading indicates component use by the netlists. Only one netlist can be active at a time.

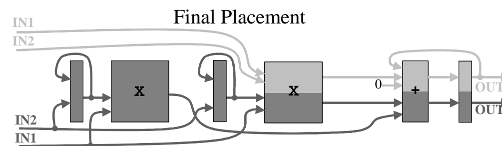


Figure 5: The final placement of the architecture created for the netlists from Figure 3. The signal cross-section has been greatly reduced from the initial placement shown in Figure 4.

The cost metric is based on the cross-section of signals communicating between the bound instances. At each physical unit location the cross-section of signals for each netlist is determined. The maximum across the netlists becomes the overall cross-section value at that

point. After the cross-section value is calculated for each location, the values are squared, then summed across the locations to yield the overall cost value. By squaring the values before summing across positions, areas with a high cross-section are heavily penalized. The goal in reducing these cross-sections is primarily to minimize the area of the routing structure that will be created, because a larger cross-section can lead to a taller architecture. A secondary goal is to decrease the delay of the nets, because the longer (and slower) a wire is, the more likely it is to share a span with other wires and contribute to a larger cross-section.

The guidelines presented for VPR [18], a place and route tool for FPGAs, govern the initial temperature calculation, number of moves per temperature, and cooling schedule. These values are based on N_{blocks}, the number of "blocks" in a netlist. Since both netlist instances and physical components are being used, N_{blocks} is calculated as the sum of the instances in each netlist provided plus the number of physical components created. The initial temperature and number of moves per temperature are derived from this value. The cooling schedule specified by VPR is also used, where the new temperature T_{new} is calculated according to the percentage of moves that were accepted (R_{accept}) at the old temperature T_{old}.

3.2. Routing Generation

While RaPiD uses a series of regular routing tracks, multiplexers, and demultiplexers to connect the units, cASIC architectures provide a more specialized communication structure. The only routing resources included are those which are explicitly required by one or more of the netlists. This section discusses the different techniques developed to create the routing structures. The routing structure of a custom generated architecture will depend on the

placement achieved using the techniques of section 3.1. At this point, the physical locations of the components are fixed, as are the bindings of the netlist instances to those components. The specification netlists define the signals that connect the netlist instances to form a circuit. These instances have been bound in the placement stage, so the physical locations of the ports of the signals are known. Wires are then created to implement these signals, allowing each netlist to execute individually on the custom hardware.

In addition to wires, we may create multiplexers and demultiplexers on the ports of the components to accommodate the different requirements of the specification netlists. For example, if netlist A needs an adder to output to a register, but netlist B needs the adder to output to a RAM, a demultiplexer is instantiated on the output of the adder to direct the signals properly for each netlist. Similarly, if netlist A has a register that receives an input from an adder, but netlist B needs that register to input from a multiplier, a multiplexer is created to choose the register input based on which netlist is currently active in the architecture. Figure 6 continues the example in section 3.1, showing the generated routing structure for the placement of Figure 5. Note that several of the wires here are used to implement signals from both netlists. Like the logic resources, the wires are only used by one netlist at a time — whichever netlist is currently programmed onto the architecture. "Sharing" of routing resources between netlists is critical, as the routing architecture can become extremely large if each signal is implemented by a dedicated wire.

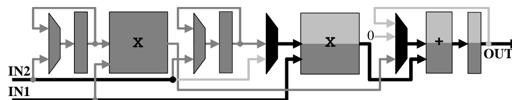


Figure 6: cASIC routing architecture created for the example from section 3.1. Black wires are used by both netlists of Figure 3, while the light and dark grey wires are only used by the corresponding netlist.

The object of the routing generation phase is to minimize area by sharing wires between netlists, while adding as few multiplexers and demultiplexers as necessary. Heuristics are used to group signals with similar connections from different netlists into wires. In order to understand the motivations for the algorithms presented below, the routing problem itself must first be discussed. As with the placement problem, creating the routing is two problems combined into one: creating the wires, and assigning of the signals to wires. In many current FPGA architectures, wire lengths can be adjusted for each netlist by taking advantage of programmable connections (segmentation points) between lengths of wire, potentially forming a single long wire out of several short wires. For simplicity, the current Totem cASIC generation algorithms do not provide this flexibility.

The algorithms must somehow determine which sets of signals belong together within a wire. One method is to simply not share at all, which is explored in the No Sharing algorithm. The remaining algorithms, Greedy, Bipartite and Clique, use heuristics to determine how the wires should be shared between signals. The heuristics operate by placing signals with a high degree of similarity together into the same wire. However, "similarity" can be computed several different ways. In this work, two different definitions of "similarity" were used. *Ports* refers to the number of input/output locations the signals or wires have in common. *Overlap* refers to a common "span", where the span of a signal or wire is bounded by the leftmost source or sink and the rightmost source or sink in the placed architecture. Results for each of these similarity types are given in section 4. The procedures used by the Greedy, Bipartite, and Clique heuristics are described in the next sections.

3.2.1. Greedy

The greedy algorithm operates by repeatedly merging wires that are very similar. To begin, each signal is assigned to its own wire. Next, a list of correlations between all compatible wire pairs (wires that are not both used in the same netlist) is created. The highest correlation value is selected at each iteration, and those two wires are merged. All other correlations related to either of the two wires that have been merged are updated according to the characteristics of the new shared wire. If any of the correlations now contain a conflict due to the new attributes of the merged wire (i.e., both wires in the correlation hold a signal from the same netlist), these correlations are deleted from the list, as they are no longer valid. This process continues until the correlation list is empty, and no further wires may be merged.

3.2.2. Bipartite

The merging of netlists into cASIC architectures is a form of matching problem. This might naturally lead one to consider the use of bipartite matching to solve the problem. One group has already used maximum weight bipartite matching to match netlist instances together to form the components [19]. However, there are two significant problems with this approach. The first is that this type of logic construction does not consider the physical locations of the instances or their components. The physical locations of components and mapped instances determines the length of wires needed to make the connections between units, and is therefore critical to effective logic construction. Furthermore, although bipartite matching was used to determine sharing of logic resources, the routing resources (wires) were not shared between netlists at all using their technique.

Second, the bipartite matching algorithm was used recursively, matching two netlists together, then matching a third netlist to the existing matching, and so on. While any individual matching can be guaranteed to have the maximum weight, the cumulative solution may not. The order in which the netlists are matched can affect the quality of the final solution. This is true even if bipartite matching is not used for the logic construction but only for routing construction.

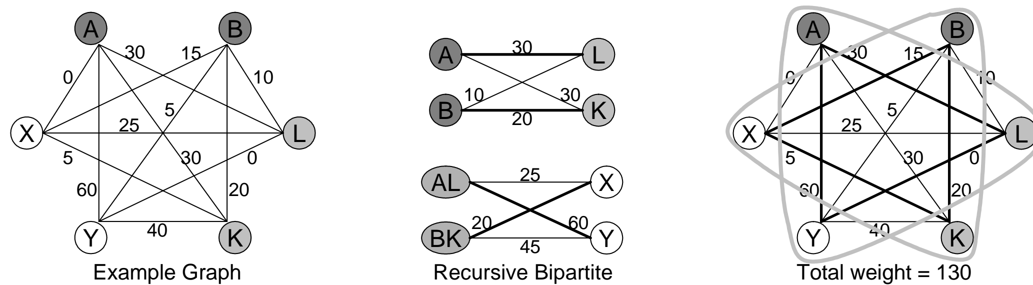


Figure 7: An example graph which does not produce the optimal solution when bipartite matching is used recursively. At left is the original graph, the middle shows the two recursion steps, and at right is the solution. The total weight is the weight of the edges completely contained by the grouping.

A cASIC generation algorithm was created which uses recursive maximum weight bipartite matching, to compare with the proposed Clique proposed discussed in the next section. Logic for these architectures is still constructed as discussed in section 3.1 because of the location issue mentioned previously, but the merging of signals into wires is performed using a bipartite matching algorithm.

3.2.3. Clique

The downside of the Greedy and Bipartite techniques is that they merge wires based on short-term local benefits, without considering the problem as a whole. There may be cases where merging the two most similar wires at one point prevents a more ideal merging later in the

algorithm. Therefore, the Clique algorithm has been implemented to globally address the routing creation problem.

Clique partitioning is a concept from graph algorithms whereby vertices are divided into completely connected groups. In our algorithm each wire is represented by a vertex, and the "groups", or cliques, represent physical wires. The algorithm uses a weighted-edge version of clique partitioning to group highly correlated signals together into wires, where the correlation value between signals is used as the edge weight. The cliques are then partitioned such that the weight of the edges connecting vertices within the same clique is maximized. Signals that cannot occupy the same wire (signals from the same netlist) carry an extremely large negative weight that will prevent them from being assigned to the same clique. Therefore, although signal A may be highly correlated with signal B, and signal B is highly correlated with signal C, they will not all be placed into the same wire (clique) if signal A conflicts with signal C, due to the large negative weight between those vertices. Figure 8 shows the clique partitioning solution to the weighted-edge graph from the example of Figure 7.

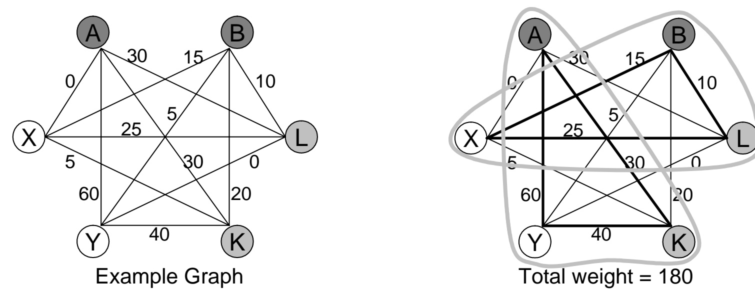


Figure 8: An improved solution to the graph of Figure 7 found using clique partitioning. The nodes in this graph represent the signals, and each clique (grouping) represents a wire shared by the vertices (signals) in the cliques. The total weight is the sum of all edge weights completely contained within a clique.

Given that weighted clique partitioning of a graph with both negative and positive edge weights is NP-Complete, an ejection chain heuristic based on tabu search [20] is used. Vertices are initially assigned to random cliques (where the number of cliques equals the number of vertices). Some cliques are allowed to be empty, but all vertices must be assigned to a clique. The algorithm then iteratively moves each vertex from its current clique to a different one. This is done by each time selecting a non-tabu vertex and a new clique for that vertex that will produce the maximum overall (not necessarily positive) gain in total weight for the graph. Once a vertex is moved, it is marked tabu until the next iteration. After all the vertices have been moved in an iteration, the list of cumulative solutions after each move is examined, and the one with the highest total weight is chosen. This solution is then used as the base for the next iteration of moves, and all vertices are marked non-tabu. This loop continues until none of the cumulative solutions in an iteration produces a total weight greater than the base solution for that iteration.

4. Results

Eight different applications (each composed of two or more netlists) were used to compare the area results of the Totem architectures to a number of existing implementation techniques, including standard cell, FPGA, and RaPiD techniques. These applications, along with their member netlists, are listed in Table 1. Five of these are real applications used for radar, OFDM, digital camera, speech recognition, and image processing. The remaining three applications are sets of related netlists, such as a collection of different FIR filters. The netlists were compiled from RaPiD-C by the RaPiD compiler.

Table 1: Eight applications used to test Totem architectures, each containing two or more distinct netlists. FIR, Matrix, and Sort are collections of similar netlists, while the others are actual applications.

Application	Member Netlists
Radar	decnsr, fft16_2nd, psd
OFDM	sync, fft64
Camera	color_interp, img_filt, med_filt
Speech	log32, fft32, 1d_dct40
FIR	firsm, firsm2, firsm3, firsymeven, firtm_1st, firtm_2nd
Matrix	matmult, matmult4, matmult_bit, limited, limited2
Sort	sort_g, sort_rb, sort_2d_g, sort_2d_rb
Image	med_filt, matmult, firtm_2nd, fft16_2nd, 1d_dct40

4.1. Reference Implementations

The applications listed in Table 1 were implemented using standard cells, an FPGA, and RaPiD. These three implementation types are used to provide comparative results used in evaluation of the cASIC architectures. The following paragraphs describe the techniques used for these three comparative implementation methods.

The standard cell layouts of the netlists (converted automatically from RaPiD netlist format to structural Verilog) were created using Cadence in a TSMC 0.18μm process with 6 metal layers. Generally, the total area for an application set is the sum of the areas required for the netlists. However, for the application sets which are a collection of very similar netlists (FIR, Matrix, and Sort from Table 1), this assumption is likely to be incorrect. Therefore, to err on the side of caution for these particular cases, the maximum area required by any one member netlist is instead used, under the assumption that a small amount of additional control circuitry may allow all member netlists to use the same hardware. I/O area is not included, as I/O area is also not measured for the Totem architectures.

The FPGA solution is based on the Xilinx Virtex-II FPGA, which uses a 0.15μm 8-metal-layer process, with transistors at 0.12μm [21]. In particular, the die area was obtained for an XC2V1000 device [22]. This FPGA contains not only LUT-based logic ("slices"), but also embedded RAM and multiplier units, in a proportion of (128 slices : 1 multiplier : 1 RAM). This proportion of resources is used as a tileable atomic unit when determining required FPGA area for the designs, as manually-designed FPGA cores for SoCs are unlikely to be very customizable except in terms of the quantity of total tileable resources. The area of an individual tile, which corresponds to approximately 25K system "gates" of logic, was computed (using a photograph of the die) to be 1.141mm^2. This area was then scaled to a 0.18μm process by multiplying by $(.15/.18)^2$ to yield a final tile size of 1.643mm^2 to compare all solutions using the same fabrication process. The Verilog files created from individual netlists were placed and routed onto a Virtex-II chip, and the number of tiles required for the applications were measured. In this case, the total area required by an application is the maximum of the areas required by its member netlists, as the hardware resources are reusable.

The area required to implement the applications on a static RaPiD architecture [2][14] was also calculated. The RaPiD results represent a partially-customized FPGA solution. The RaPiD reconfigurable architecture was designed for the types of netlists used in this testing, and contains specialized coarse-grained computational units used by those netlists. The number of RaPiD cells can be varied, but the resource mix and routing structures within the cell is fixed.

To find the area for each application, the minimum number of RaPiD cells needed to meet the logic requirements of the application was calculated. The application's netlists were then placed and routed onto the architecture to verify that enough routing resources were present. If not, and the routing failed, the number of cells was increased by one until either all of the

application's netlists could successfully place and route, or place and route still failed with 20% more cells than the application logic required.

Manual layouts of each of the units and routing structures were created in a TSMC 0.18µm process with 5 metal layers. The logic area is simply the sum of the areas of the logic units in the architecture. Routing area is the sum of the areas of the multiplexers, demultiplexers, and bus connectors (segmentation points) in the architecture. Routing tracks are directly over the logic units in a higher layer of metal, and are therefore not counted as contributing to the area. In some cases, the RaPiD architecture did not have sufficient routing resources to implement a circuit. The RaPiD cell would have to be manually redesigned to fit these netlists. This illustrates one of the primary benefits of an automatic architecture generator – provided enough die area is allocated, a solution can always be created.

4.2. cASIC Implementations

For the cASIC architecture generation methods, the areas are computed based on the manual layouts used for the RaPiD area calculation. Logic area is computed using the same method, but the routing area is a more complex computation. Area used by multiplexers and demultiplexers are again computed according to manual layouts. But unlike RaPiD, wire area can add to the total area of the architecture. A wire cross-section of up to 24 can be routed directly over the logic units, so as with RaPiD, this routing area is considered "free". However, when the routing cross-section is greater than 24, the additional cross-section adds to the height of the architecture.

Table 2: The areas of the routing structures created by the Bipartite generation methods using both the ports and the overlap methods. All possible orderings of the netlists were considered, and the minimum, average, and maximum areas are given. The percent difference between the minimum and maximum areas is also given.

	Radar	OFDM	Camera	Speech	FIR	Matrix	Sort	Image
# Netlists	3	2	3	2	6	5	4	5
Ports Min	0.075	0.731	0.958	0.476	0.520	0.093	0.183	0.573
Ports Avg	0.076	0.731	0.959	0.476	0.582	0.094	0.185	0.582
Ports Max	0.077	0.731	0.959	0.476	0.629	0.097	0.187	0.589
% Diff	2.273	0.000	0.089	0.000	20.914	4.587	2.337	2.780
Overlap Min	0.093	0.429	0.301	0.371	0.247	0.131	0.248	0.573
Overlap Avg	0.094	0.429	0.305	0.373	0.263	0.140	0.251	0.582
Overlap Max	0.094	0.429	0.307	0.374	0.273	0.145	0.255	0.589
% Diff	0.917	0.000	1.983	0.690	10.345	10.390	2.749	2.780

First, the Bipartite technique was examined to determine the effect of the order that netlists are merged into the cumulative solution. Table 2 lists, for each application, the minimum, average, and maximum areas across the solutions for each ordering of the netlists. The percent difference between the minimum and maximum areas is also given. When there are only two netlists, there is only one possible ordering, and the minimum and maximum values are identical. However, these results indicate that for any cases with more than two netlists, the ordering can affect the final area. For the circuit sets examined here with more than two netlists, there is on average approximately a 4.5% difference in routing area between the best and the worst orderings. However, in one case, the routing area varies by as much as 20%. Therefore, this technique may not be appropriate for cases with more than two netlists.

Next we generated architectures using Greedy, Clique, and No Sharing techniques. The No Sharing algorithm creates a separate wire for every signal—a completely different set of wires is used depending on which netlist is in operation. This method is included to demonstrate the importance of sharing routing resources between netlists. An area comparison of the tested cASIC methods is given in Figure 9, which has been normalized to the area result of Clique

Overlap (which on average produces the smallest architectures). Areas are listed for Greedy, the average Bipartite case, and Clique, each with two categories: Ports, and Overlap. As stated previously, Ports indicates that the similarity between signals is computed according to the number of sources and sinks shared by those signals. Overlap indicates that the similarity is computed according to common location and length of the signals.

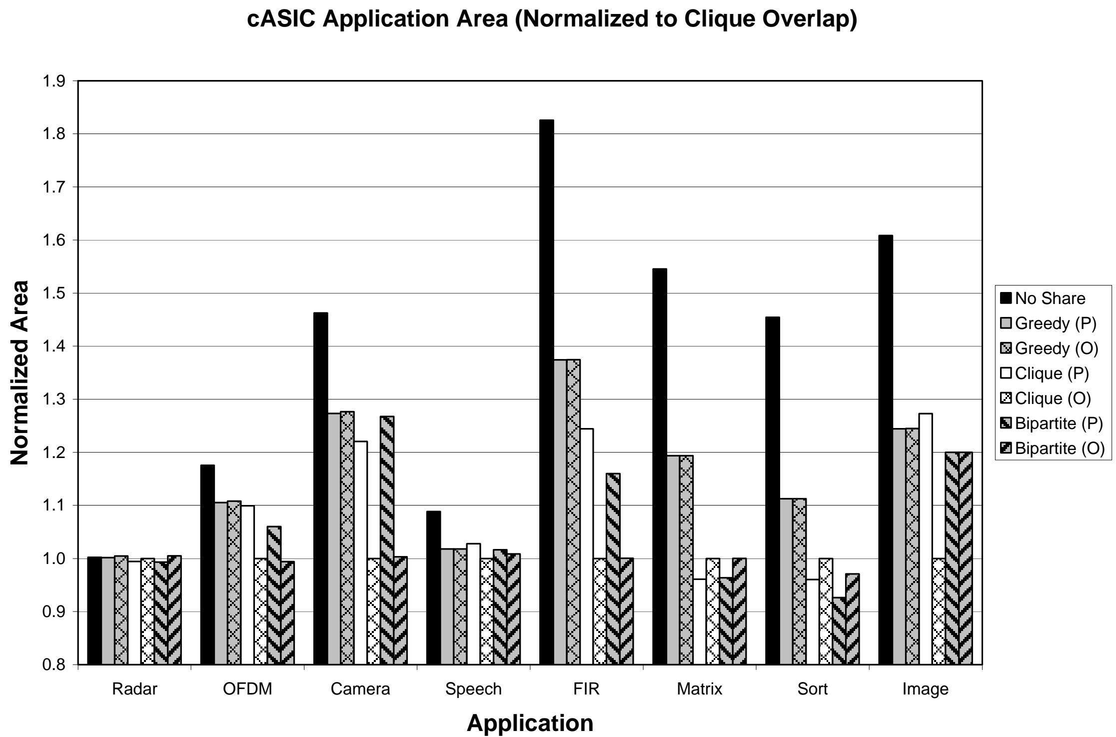


Figure 9: Comparative area results of the different cASIC routing generation algorithms, normalized to the Clique Overlap result for each application. The Bipartite results given are the average across orderings.

As expected, all three heuristic techniques of both similarity types perform better than the No Share algorithm for all applications. Generally, Clique performs better than the other methods, with Clique Overlap on average 2% smaller than Bipartite Overlap, 6% smaller than Bipartite Ports, and 13% smaller than Greedy Ports or Greedy Overlap. However, there is clearly room for improvement of the weighting calculation used by the Clique Partitioning

method, as both Greedy and the average Bipartite produce a smaller area in some situations. Additionally, Clique sometimes performs better using Ports, and other times using Overlap. Neither is consistently better than the other. An improved similarity (weight) calculation for the Clique method should therefore incorporate both Ports and Overlap similarity techniques.

Table 3: The areas, in mm^2, of the eight different applications from Table 1, as implemented in standard cells, a Virtex-II, RaPiD, and the various cASIC techniques. A summary of these results appears in Table 4. Bipartite results are given for the minimum, maximum, and average over all netlist orderings.

		Radar	OFDM	Camera	Speech	FIR	Matrix	Sort	Image
Std. Cell	Total	4.101	9.168	7.268	26.523	2.846	1.785	1.541	6.843
FPGA	Total	19.719	59.157	23.006	78.877	26.292	19.719	26.292	19.719
RaPiD	Logic	2.838	---	---	45.401	3.783	1.892	2.838	---
	Routing	2.158	---	---	34.536	2.878	1.439	2.158	---
	Total	4.996	---	---	79.937	6.661	3.331	4.996	---
No Share	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.090	1.259	1.441	1.413	1.917	0.829	0.970	1.331
	Total	1.523	5.377	3.619	14.161	3.658	1.952	2.163	2.947
Greedy Ports	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.089	0.938	0.973	0.493	1.011	0.385	0.462	0.664
	Total	1.522	5.055	3.151	13.242	2.753	1.508	1.655	2.280
Greedy Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.093	0.950	0.981	0.493	1.012	0.385	0.462	0.664
	Total	1.526	5.068	3.159	13.242	2.754	1.508	1.655	2.281
Clique Ports	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.078	0.909	0.842	0.622	0.752	0.090	0.236	0.716
	Total	1.511	5.027	3.020	13.370	2.493	1.214	1.428	2.333
Clique Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.086	0.456	0.297	0.261	0.262	0.140	0.294	0.216
	Total	1.520	4.574	2.475	13.010	2.004	1.264	1.487	1.833
Bipartite Min Ports	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.075	0.731	0.958	0.476	0.520	0.093	0.183	0.573
	Total	1.509	4.849	3.136	13.224	2.262	1.217	1.376	2.190
Bipartite Avg Ports	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.076	0.731	0.959	0.476	0.582	0.094	0.185	0.582
	Total	1.509	4.849	3.136	13.224	2.324	1.218	1.378	2.199
Bipartite Max Ports	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.077	0.731	0.959	0.476	0.629	0.097	0.187	0.589
	Total	1.510	4.849	3.137	13.224	2.370	1.221	1.380	2.206
Bipartite Min Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.093	0.429	0.301	0.371	0.247	0.131	0.248	0.573
	Total	1.526	4.547	2.479	13.120	1.989	1.255	1.441	2.190
Bipartite Avg Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.094	0.429	0.305	0.373	0.263	0.140	0.251	0.582
	Total	1.527	4.547	2.483	13.121	2.005	1.264	1.444	2.199
Bipartite Max Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.094	0.429	0.307	0.374	0.273	0.145	0.255	0.589
	Total	1.527	4.547	2.485	13.122	2.015	1.269	1.448	2.206

Table 4: Area improvements calculated over the reference architectures, then averaged across all applications. The Bipartite results are the average across netlist orderings.

Improvement Over Std. Cells	
Method	**Area**
FPGA	0.20
RaPiD	0.48
No Share	1.63
Greedy (P)	1.87
Greedy (O)	1.87
Clique (P)	1.94
Clique (O)	2.16
Bipartite (P)	1.98
Bipartite (O)	2.08

Improvement Over RaPiD	
Method	**Area**
Std Cells	2.34
FPGA	0.38
No Share	2.95
Greedy (P)	3.39
Greedy (O)	3.39
Clique (P)	3.64
Clique (O)	3.75
Bipartite (P)	3.72
Bipartite (O)	3.76

Improvement Over FPGA	
Method	**Area**
Std Cells	7.20
RaPiD	4.01
No Share	9.00
Greedy (P)	10.63
Greedy (O)	10.62
Clique (P)	11.50
Clique (O)	12.30
Bipartite (P)	11.77
Bipartite (O)	12.14

Table 3 gives the areas found by the different cASIC routing generation algorithms, with the corresponding standard cell, FPGA, and RaPiD areas listed for comparison. These results are summarized in Table 4. As these tables indicate, cASIC architectures are significantly smaller than the corresponding FPGA area for the same netlists. The heuristics range on average from a 10.6x improvement to a 12.3x improvement in area, while even the No Sharing algorithm results in a 9x improvement. FPGAs without custom embedded multipliers and RAMs would be expected to require even more area than the Virtex-II for these applications. The Virtex-II here is only a factor of 7.2x larger than standard cells, whereas with older homogenous FPGAs, implementations were generally assumed to be 1-2 orders of magnitude larger.

Comparisons of cASIC techniques to RaPiD also yield favorable results, with area improvements of 3.4x to 3.8x for the cASIC heuristics. These applications were created for RaPiD, and RaPiD has been hand-optimized for DSP-type operations, which make it more efficient (2.8x smaller) than a generic FPGA for these applications.

Finally, the cASIC heuristic methods also created architectures on average half the size of standard cell implementations of the applications. One of the reasons the cASIC architectures

are able to achieve such small areas is because the tools use full-custom layouts for the computation blocks. The FIR, Matrix, and Sort architectures demonstrate the value of the full-custom units. In these applications, the standard cell area is estimated to be the size of the largest member netlist (as explained in section 4.1) to give standard cell design the benefit of the doubt. Even with the overhead of adding reconfigurability, these cASIC area results are close to or slightly better than the standard cell implementation. Using a library of coarse units in conjunction with a standard cell synthesis tool would, of course, improve the standard cell results.

However, the largest benefits occur in the cases where an application has several differently-structured netlists, and a separate circuit must be created for each member netlist in a standard cell implementation. By reusing components for different netlists, the cASIC architectures achieve areas on the order of a full-custom implementation (generally assumed 2-3x smaller than standard cells). While the use of library components in these cases would decrease the standard cell area to some extent, it would not solve the problem of hardware reuse.

The cASIC method of architecture creation therefore has significant area benefits for situations in which standard cells are generally considered instead of FPGAs for efficiency reasons. Finally, a full-custom manual layout could be created for these applications that might be smaller than the cASIC architectures. However, this would require considerably more design time, which can be quite expensive, and may not always be possible due to production deadlines.

5. Future Directions

As mentioned previously, the current cASIC tools are limited to customized 1D RaPiD-style datapaths. However, this limitation is an implementation detail, not a necessity for the idea or techniques themselves. Expanding this process to 2D would require augmenting the placement process to use a 2D grid, an easy modification. Routing would become slightly more difficult, but could be accomplished using maze routing techniques. It is possible that empty spaces would need to be inserted to accommodate bends in wires, but we would expect this to cause at most a minor increase in area given that the routing structure is very limited in cASICs.

The techniques described could also target different netlist types and logic types beyond what is supported by the RaPiD hardware and netlists. This would require creating manual layouts of the new logic units. The majority of the cASIC tool would remain unchanged, as it was written in a very parameterized manner. The only potential difficulty would be if the logic units were not of a uniform height. In this case, macro-cell placement techniques would be employed to achieve an efficient layout.

Finally, the architectures could be generated with a little additional flexibility in an effort to allow for minor changes to the specified circuit set. This could be accomplished by increasing the size of the multiplexers on the logic unit inputs by a set percent, or to a set size. Routing tracks could then be added to allow routing flexibility. In this case, a flexible router such as Independence [23] could be used to map new circuits to the hardware. However, at this point we are no longer performing the task targeted by this article. The design would then approach a domain-flexible architecture in style, and we suggest that other architecture generation techniques designed for flexible domain-specific architectures would be more appropriate [17].

6. Conclusions

This article described the cASIC style of architecture, and presented three different heuristics to create these designs. The first uses a greedy approach, the second uses recursive maximum weight bipartite matching, while the third uses a more sophisticated graph-based algorithm called clique partitioning to merge groups of similar signals into wires. Two different methods to measure this signal similarity were discussed, one based on the common ports of the signals, and the other based on the common span (overlap). The results indicate that a better similarity measurement would be a combination of the two, incorporating both ports and signal overlap.

The area comparison also demonstrates the inefficiencies introduced by the flexibility of FPGAs. While the generic structure is critical for implementing as wide a variety of circuits as possible, it is that flexibility that causes it to require 12x more area than a cASIC architecture. The Virtex-II FPGA does, however, perform much better than earlier FPGA designs, at least in part due to the use of coarse-grained multiplier and RAM units. The RaPiD architecture extends the use of coarse-grained units to the entire architecture, but is customized to DSP as a whole. If the application set is only a subset of DSP, further optimization opportunities exist, with cASIC techniques achieving up to 3.8x area improvements over the RaPiD solution.

This article also demonstrated another key benefit cASIC generation has over the use of a static architecture such as RaPiD. In cASIC generation, if enough area is allotted on the SoC die, an architecture can be created for any set of netlists. On the other hand, the RaPiD resource mix is fixed. For some applications, this structure may not have the correct logic mix for the application, leading to copious wasted area. Alternately, a static structure may not provide a rich

enough routing fabric, as was demonstrated by the failure of some applications to place and route onto a RaPiD architecture. Finally, cASIC architectures have been created that are under half the size of standard cell implementations of the desired application set. These area results indicate that cASIC architecture design is not only an excellent alternative to FPGA structures when the target circuits are known, but also a viable alternative to standard cell implementations.

7. References

[1] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", *ACM Computing Surveys*, Vol. 34, No. 2. pp. 171-210, June 2002.

[2] 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.

[3] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler", *IEEE Computer*, Vol. 33, No. 4, pp. 70-77, 2000.

[4] A. Abnous, J. Rabaey, "Ultra-Low-Power Domain-Specific Multimedia Processors", *Proceedings of the IEEE VLSI Signal Processing Workshop*, October 1996.

[5] Morpho Technologies, http://www.morphotech.com/

[6] Stretch, Inc., http://www.stretchinc.com/

[7] The Totem Project, http://www.ee.washington.edu/people/faculty/hauck/Totem/

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

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

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] M. Scott, “The RaPiD Cell Structure”, *Personal Communications*, 2001.

[17] 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.

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

[19] Z. Huang, S. Malik, “Managing Dynamic Reconfiguration Overhead in Systems-on-a-Chip Design Using Reconfigurable Datapaths and Optimized Interconnection Networks”, *Conference of Design Automation and Test in Europe (DATE)*, 2001.

[20] U. Dorndorf, E. Pesch, "Fast Clustering Algorithms", *ORSA Journal on Computing*, Vol. 6, No. 2, pp. 141-152, 1994.

[21] Xilinx, Inc., *Virtex™-II Platform FPGAs: Detailed Description*. Xilinx, Inc., San Jose, CA, 2002.

[22] Chipworks, Inc., “Xilinx XC2V1000 Die Size And Photograph”, Chipworks, Inc., Ottawa, Canada, 2002.

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