Automatic Design of Area-Efficient Configurable ASIC Cores

Katherine Compton, *Member*, *IEEE*, and Scott Hauck, *Senior Member*, *IEEE*

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. Restricting the structure to support a class or a specific set of algorithms can increase efficiency 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 paper 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.

Index Terms—Reconfigurable architecture, logic design and synthesis.

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 can be more efficiently performed by coarse-grained components. A number of reconfigurable systems have therefore been designed with a coarse-grained structure, but these designs 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 when computational requirements are not known in advance, specialized hardware could be used to obtain greater performance for a specific set of compute-intensive calculations.

Efforts to reduce the amount of "useless" hardware and increase the efficiency of the computing device have focused on more customized structures. Architectures such as RaPiD [2], PipeRench [3], and Pleiades [4] target 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 specialization a step further, allowing the user to select the computation domain (such as signal processing, encryption, scientific data processing, or a subset of applications within a domain) by providing representative circuits to an architecture generator, a concept 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 domain-specific Systems-on-a-Chip (SoCs), where ASIC-style accelerators would otherwise be used. cASICs are not intended to replace entire ASIC-only chips. The cASIC hardware would accelerate the most compute-intense and most common applications for which the SoC is intended, acting as support hardware or coprocessor circuitry to a host microprocessor. The host would execute software code and compute-intensive sections would be off-loaded to one or more cASIC accelerators to increase efficiency. The cASIC design flow would be part of the design process for the SoC itself. Ideally, this process would be automated, with a complete tool flow to process application descriptions in high-level languages and output cASIC designs based on compute-intense commonly executed code. Although much work has been published in this general area of hardware/software codesign and hardware compilation, the most relevant addresses extraction of inner loops so as to create cASIC-style designs [13]. This paper, however, focuses on techniques to design the cASIC hardware after the circuit candidates are known.

- *K. Compton is with the Department of Electrical and Computer Engineering, University of Wisconsin-Madison, 1415 Engineering Drive, Madison, WI 53706. E-mail: kati@engr.wisc.edu.*
- *S. Hauck is with the Department of Electrical and Computer Engineering, University of Washington, Box 352500, Seattle, WA 98195. E-mail: hauck@ee.washington.edu.*

Manuscript received 14 June 2005; revised 5 Dec. 2005; accepted 11 Oct. 2006; published online 21 Mar. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0197-0605.
Digital Object Identifier no. XXX

1. The Totem Project is a multiperson project with several publications. The cited theses are the most comprehensive documents. All Totem publications can be found at the referenced Web site.

While the circuit set that cASICs can implement is fixed, they are different from traditional ASICs in that some hardware programmability is retained [8], [14]. This also separates cASICs from structured ASICs, where configurability is at the mask level and the design can only be configured once prior to the final fabrication steps. The eASIC FlexASIC [15] is somewhat of an exception—the routing is mask-programmable, but logic is in the form of SRAM-programmable LUTs. However, it does not appear that FlexASICs are intended to be runtime reconfigurable like cASICs. The Flexible ASIC (not related to FlexASIC) is a type of structured ASIC closer in flavor to cASICs. Automatic design tools create a base design suitable for a set of specific related applications and mask programming is used to complete the design for a single chosen application. The goal of this work is to minimize the number of interconnect layers needed for any one application and amortize design costs across all applications. Due to mask-level programmability, Flexible ASICs are not suitable for runtime reconfiguration.

cASICs use runtime reconfiguration to implement different circuits at different times. Because the set of circuits and number of reconfiguration points is limited, not only is cASIC performance expected to be close to an ASIC, but the circuit will be significantly smaller than an FPGA implementation. Furthermore, because we reuse hardware to implement multiple circuits, cASICs can be smaller than the sum of the ASIC areas of the individual circuits. The area benefit of cASICs is critical. For many high-performance applications, special hardware accelerators can become quite large. If each desired accelerator were implemented separately, this could result in an unreasonably large (and expensive) chip. Allowing the accelerators to share hardware makes their use more attractive to 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 have some ability to find resource sharing opportunities, but unfortunately 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. We verified this problem by attempting to synthesize a set of four multiply-accumulate (MAC) circuits, each with a different pipeline depth, designed as separate modules joined by an outer module that chooses between their outputs. Both Synopsys and the Xilinx tools synthesized four separate multiplier units, even on high effort and optimizing for area.

Flattening the circuit can help, but will dramatically increase synthesis time and many sharing opportunities may still be overlooked by tools optimized for sharing within smaller areas of a single circuit. In our experiments, automatic flattening did not allow multiplier sharing in either Synopsis or the Xilinx tools for our MAC test case. However, by rewriting the code as one HDL module, Synopsys was able to eliminate two multipliers and the Xilinx tools eliminated one. Unfortunately, requiring all code to be contained within a single HDL module for large designs is counter to good design practices. More research into automated cASIC sharing is therefore essential 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 paper describes our work toward a cASIC generator including experiments and data beyond the initial preliminary results [16]. 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 buses. 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 paper, the generated cASIC architectures are significantly smaller than required by a traditional FPGA implementation, in some cases using less than half the area required for a set of separate standard cell implementations with the same functionality.

2 Background

Current efforts in the Totem Project focus on coarse-grained architectures for compute-intensive application domains such as digital signal processing, compression, and encryption. The RaPiD architecture [2], [17] 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 [18] 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.

The RaPiD architecture is composed of a set of repeating cells (Fig. 1) tiled horizontally. The logic units within the cells operate on full words of data and include 16-bit ALUs, 16×16 multipliers, 16-bit wide RAM units, and 16-bit registers. Each component contains a multiplexer on each of its inputs that chooses between the signals of each routing track. Each component also has a demultiplexer on each of the outputs that allows the unit to directly output to any of the routing tracks. Inputs are on the left side of a unit, while the outputs are on the right side of the unit. Global inputs

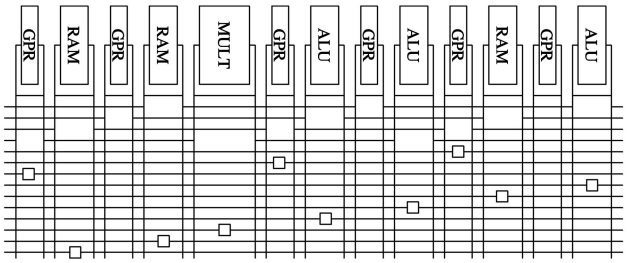


Fig. 1. A single cell of RaPiD [17], [19]. A full architecture is composed of multiple cells laid end-to-end.

also reside on the 1D datapath at the ends of the architecture.

The routing architecture is one-dimensional and segmented, with each track word-width. The top routing tracks are local, containing short wires for fast short-distance communication. The bottom 10 tracks provide longer-distance routing, allowing wires to be connected to form longer routes. The bus connectors (small squares in the figure) also 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 paper 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], [17], 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 routing network—the only wires and multiplexers available are those required by the netlists. 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 is unlikely to be implementable in the generated hardware. This hardware is optimized for 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 significantly fewer present than in an FPGA, where each LUT and all flexible routing must be configured. In cASICs, configuration bits control any multiplexers needed on the inputs of logic units, as well as ALU modes. These configuration bits allow for hardware reuse among the specification netlists. Intelligent sharing of logic and routing resources keeps the quantity of configuration bits required extremely small. Each netlist in the specification is implemented in turn by programming these bits appropriately. Multicontexting could be used to ensure single-cycle configuration of the hardware [20], though the scarcity of configuration bits would result in relatively fast reconfiguration times even without multicontexting.

Ideally, the design flow for cASICs would be entirely automatic. Applications could be written at a high level and compute-intensive sections of code would be selected for hardware implementation. A wide variety of research is ongoing, related to compiling high-level languages to hardware or mixed hardware and software.

cASIC architecture generation occurs in two phases. The logic phase determines the computation needs of the application netlists, creates the computational components (ALUs, RAMs, multipliers, registers, etc.), and orders the physical elements along the one-dimensional datapath. Also, the netlist instances must be bound to the physical components. The routing phase creates wires and multiplexers to connect the logic and I/O components.

3.1 Logic Generation

cASIC logic generation 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 enforced. 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 four 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 [21].

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

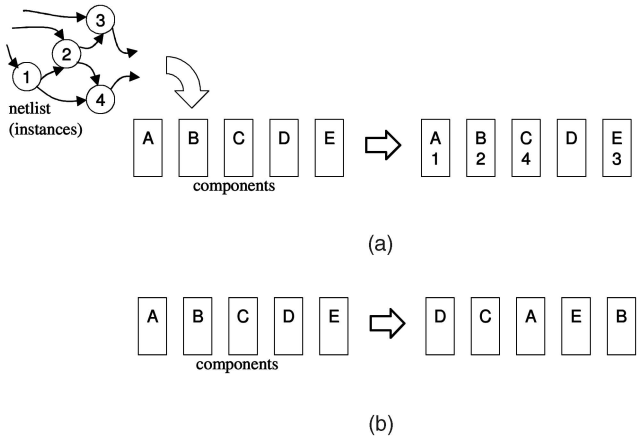


Fig. 2. Binding versus physical moves. (a) *Binding* assigns instances of a netlist to physical components. (b) *Physical moves* reposition the physical components themselves.

multiplier instance followed by adder and register instances. These instances must be implemented by the appropriate type of physical components in the hardware.

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. Fig. 2 illustrates the difference between binding and placement. Using this terminology, traditional synthesis for FPGAs requires only bindings, whereas placement for standard cells involves only physical moves.

Reconfigurable architecture generation is a unique situation in which both binding and physical moves must be considered. Locations of 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 interrelated, 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 [22], 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 attempting 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 nonimproving 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.

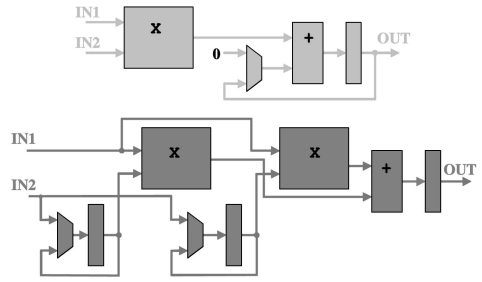


Fig. 3. Two example netlists for architecture generation. The light netlist performs a multiply-accumulate (MAC), while the dark netlist is a 2-tap FIR filter. These two netlists are used in the example placement process given in the next few figures.

In 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 a physical component's position (ordering) along the 1D axis.

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 Fig. 3 appears in Fig. 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.

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

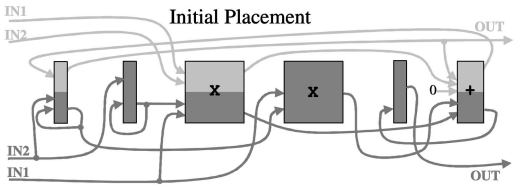


Fig. 4. An initial physical placement and binding for an architecture for the Fig. 3 netlists. Shading shows component use by netlist, but only one can be active at a time.

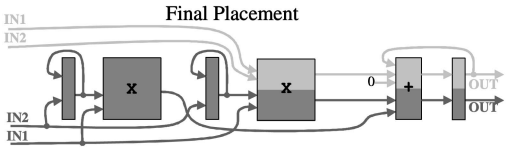


Fig. 5. Final placement of the architecture created from the initial placement in Fig. 4. The signal cross-section has been greatly reduced during the placement process.

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 [23], 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 specialized communication structure. The only routing resources are those which are explicitly required by one or more of the netlists. This section discusses cASIC routing generation techniques. After the logic structure is created using the techniques of Section 3.1, the physical locations of the components are fixed, as are the bindings of netlist instances to the 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. We then create wires to implement these signals, allowing each netlist to execute individually on the custom hardware. We may also create multiplexers and demultiplexers on component ports to accommodate the different needs of the specification netlists. For example, if netlist A has a register receiving 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.

Fig. 6 continues our example, showing the generated routing structure for the placement in Fig. 5. Note that several wires implement signals from both netlists. Like logic resources, wires are only used by one netlist at a time —whichever is currently programmed on the architecture. "Sharing" routing resources between netlists reduces area, as the routing architecture can become extremely large if each signal is implemented by a dedicated wire.

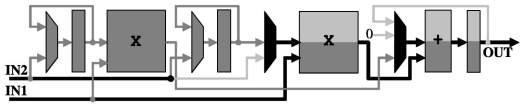


Fig. 6. cASIC routing architecture created for the example from Section 3.1. Shaded wires are used only by the indicated netlist, while black wires are used by both.

The object of routing generation is to minimize area by sharing wires between netlists while adding as few multiplexers/demultiplexers as necessary. Heuristics group signals with similar connections from different netlists into wires. To clarify 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. The raw number of shared ports is the similarity value in this case. *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. The range of component indices overlapped is the similarity value in this case. 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. A correlation contains references to a pair of wires and the similarity value between them. The correlation with the highest similarity 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

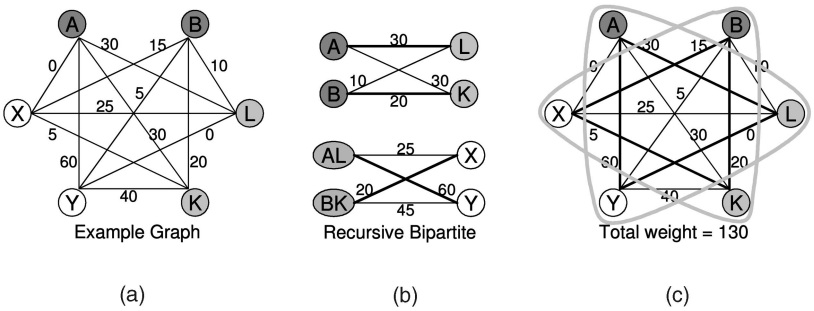


Fig. 7. Recursive bipartite matching for these three netlists results in a suboptimal solution. Nodes are signals (shades indicate netlist), circled groups are wires, and the cost is the sum of the edge weights in the groups. (a) Graph, (b) recursion steps, and (c) solution.

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 components [14]. However, there are two fundamental 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 determine the length of wires needed to make the connections between units and are 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.

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.

We created a cASIC generation algorithm that uses recursive maximum weight bipartite matching to compare against the Clique approach proposed in the next section. Logic for these architectures is constructed as discussed in Section 3.1 because of the location issue mentioned previously, but the routing generation is done via recursive bipartite matching. In this algorithm, signals are represented by nodes in the bipartite graph, edge weights are similarity values between signals, and a matched pair (and later group) becomes a wire.

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 ramifications. There may be cases where merging the two most similar wires at one point prevents a more ideal merging later in the algorithm. Clique partitioning more accurately models the routing creation problem, operating more globally than greedy and avoiding the netlist ordering problems of bipartite matching.

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 signals with high similarity together into wires, where the similarity 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 have a high similarity value with signal B and signal B may have a high similarity value 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. Fig. 8 shows the clique partitioning solution to the weighted-edge graph from the example of Fig. 7.

Given that the weighted clique partitioning of a graph with both negative and positive edge weights is NP-Complete, we use an ejection chain heuristic based on tabu search [24]. Vertices are initially assigned to random cliques (where the number of cliques equals the number of vertices). Not all cliques must contain vertices, 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 selecting a nontabu vertex each time 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

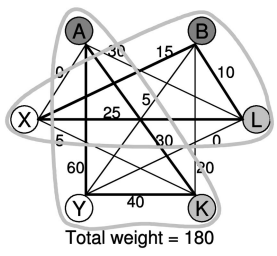


Fig. 8. An improved solution to the graph of Fig. 7 found using clique partitioning.

TABLE 1
Eight Applications, Each Containing Two or More Distinct Netlists

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

FIR, Matrix, and Sort are collections of similar netlists, while the others are actual applications.

weight is chosen. This solution is then used as the base for the next iteration of moves and all vertices are marked nontabu. 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 the 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, composed of multipliers, ALUs, RAMs, and register nodes, were compiled from RaPiD-C by the RaPiD compiler and were pipelined and retimed by that tool flow.

4.1 Reference Implementations

The applications listed in Table 1 were implemented using standard cells, an FPGA, and RaPiD to provide comparative results for evaluation of 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 $0.18\mu\mathrm{m}$ process with six metal layers. Generally, the total area for an application set is the sum of the areas required for the netlists. This is an appropriate comparison for two reasons. First, the target systems execute hybrid applications, with compute-intense sections implemented in hardware. The netlists may come from different applications or different parts of the same applications. ASIC accelerators for these applications may not all be encapsulated in a single hardware description for the synthesizer to find sharing opportunities. In fact, the designer (or the tool chain, if processing software code) may import existing cores to implement the compute-intensive sections, in which case the cores would be designed separately. Second, the ability of current commercial synthesizers to fully exploit potential resource sharing is still limited, as previously discussed.

That said, for application sets which are collections of similar netlists (FIR, Matrix, and Sort from Table 1), this assumption is likely to be incorrect and unfair. Therefore, to err on the side of caution, the maximum area required by any one member netlist is used for these cases as a small amount of additional control circuitry may allow all member netlists to use the same hardware. I/O area is not included since 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 $0.15\mu\mathrm{m}$ 8-metal-layer process, with transistors at $0.12\mu\mathrm{m}$ [25]. In particular, the die area was obtained for an XC2V1000 device [26]. This FPGA contains not only LUT-based logic ("slices," where there are two 4-LUTs per slice), but also embedded RAM and multiplier units in a proportion of 128 slices:1 multiplier:1 RAM. We use this proportion of resources as a tileable atomic unit when determining the required FPGA area for the designs. Mmanually 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 $1.141\ \mathrm{mm}^2$. This area was then scaled to a $0.18\mu\mathrm{m}$ process by multiplying by $(.15/.18)^2$ to yield a final tile size of $1.643\ \mathrm{mm}^2$ 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 was 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], [17] 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 are 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 percent more cells than the application logic required.

Manual layouts of each of the units and routing structures were created in a TSMC $0.18\mu\mathrm{m}$ process with five metal layers. The logic area is simply the sum of the areas of the logic units in the architecture. The 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

TABLE 2
Areas of the Routing Structures Created by Bipartite Matching Using Both the Ports and the Overlap Methods

	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

All netlist orderings were tested. The minimum, average, and maximum areas are given, as is the percent difference between the min and max.

architecture did not have sufficient routing tracks 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

Areas of cASIC architectures are computed based on the manual layouts used for the RaPiD area calculation. The logic area is computed using the same method, but the routing area is a more complex computation. The area used by multiplexers and demultiplexers (including the related configuration bits) are again computed according to manual layouts. Unlike RaPiD, wire area can contribute to an architecture's total area. A 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 larger, the additional cross-section adds to the architecture height.

First, the Bipartite technique was examined to determine the effect of the order in which 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 percent difference in routing area between the best and the worst orderings. However, in one case, the routing area varies by as much as 20 percent. Therefore, this technique may not be appropriate for cases with more than two netlists.

Next, we generated architectures using the 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 Fig. 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.

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 percent smaller than Bipartite Overlap, 6 percent smaller than Bipartite Ports, and 13 percent smaller than Greedy Ports or Greedy Overlap. There is clearly room for improvement in the similarity calculation and, potentially,

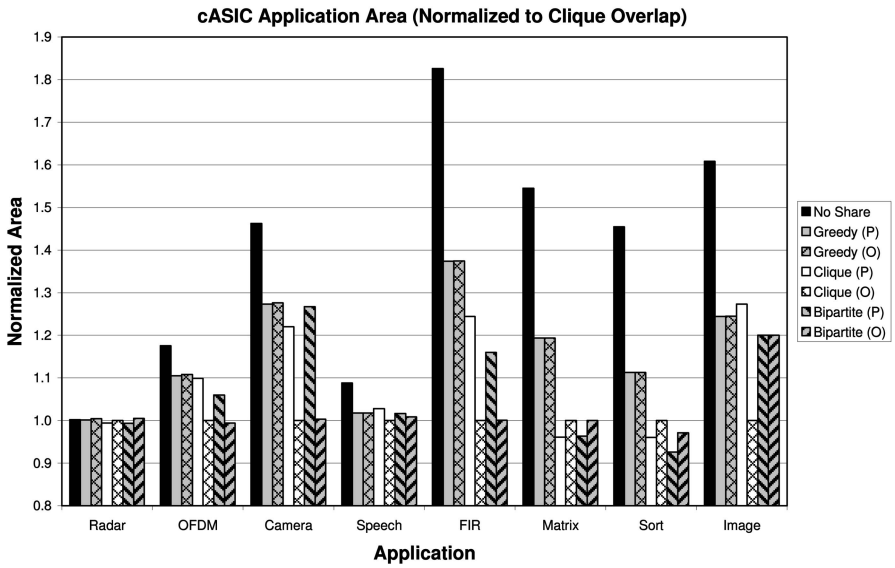


Fig. 9. Comparative area results of the different cASIC routing generation algorithms, normalized to the Clique Overlap result for each application. The Bipartite results given are the average across orderings.

TABLE 3
Areas, in mm^2, of the Eight Different Applications from Table 1 Implemented in Standard Cells, a Virtex-II, RaPiD, and the cASIC Techniques

		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

A summary of these results appears in Table 4. Bipartite results are given for the min, max, and average over all orderings.

the heuristic techniques for Clique, as both Greedy and the average Bipartite produce a smaller area in some situations, despite the flaws of these algorithms. Additionally, Clique sometimes performs better using Ports and other times using Overlap. Neither is consistently better than the other. An improved similarity (weight) calculation would consider both Ports and Overlap.

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 one to two orders of magnitude larger.

Comparisons of cASIC techniques to RaPiD also yield favorable results, with area improvements of 3.4x to 3.8x for

TABLE 4
Area Improvements Calculated over the Reference Architectures, then Averaged Across All Applications

Improvement Over Std. Cells		Improvement Over RaPiD		Improvement Over FPGA	
Method	**Area**	**Method**	**Area**	**Method**	**Area**
FPGA	0.20	Std Cells	2.34	Std Cells	7.20
RaPiD	0.48	FPGA	0.38	RaPiD	4.01
No Share	1.63	No Share	2.95	No Share	9.00
Greedy (P)	1.87	Greedy (P)	3.39	Greedy (P)	10.63
Greedy (O)	1.87	Greedy (O)	3.39	Greedy (O)	10.62
Clique (P)	1.94	Clique (P)	3.64	Clique (P)	11.50
Clique (O)	2.16	Clique (O)	3.75	Clique (O)	12.30
Bipartite (P)	1.98	Bipartite (P)	3.72	Bipartite (P)	11.77
Bipartite (O)	2.08	Bipartite (O)	3.76	Bipartite (O)	12.14

The Bipartite results are the average across netlist orderings.

the cASIC heuristics. Because they do not need to be flexible beyond the specification netlist sets, cASICs also devote significantly less area to routing resources as a proportion of the total area. These applications were created for RaPiD and RaPiD has been hand-optimized for DSP-type operations, which makes 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 to be 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. 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. A full delay and power analysis of cASIC architectures has not yet been performed, but based on relative flexibility (where flexibility is what introduces area, delay, and power overhead), we would expect results to be between ASIC and FPGA implementations.

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 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, which could increase area. But, we would expect this increase to be minor 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.

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 [27] could be used to map new circuits to the hardware. However, at this point, we are no longer performing the task targeted by this work. 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 [21].

Finally, the tool flow for cASIC design must be solidified. Until this point, we have been concerned only with the process of creating the cASICs themselves and have compiled our application netlists using the RaPiD compiler. A more compelling case could be made for cASIC design if an entire tool flow could be provided. This flow would profile a set of applications specified in a high level language to find compute-intense sections, identify similarity across applications, and synthesize circuits into netlists used for cASIC creation. This process would include retiming and automatic time-multiplexing based on area and speed constraints. The cASIC design techniques presented here, modified with the improvements listed above, would then create cASIC structures for the target SoC. cASIC mappings and placements would be verified against the original netlists. Furthermore, the created configuration bitstreams would be analyzed to ensure that physical wires would only be driven by at most one source. The final cASIC design would be sent to a layout generator [11] to create a core for the SoC.

6 Conclusions

This paper 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). Results indicated 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 paper 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.

REFERENCES

[1] K. Compton and 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, and P. Franklin, "RaPiD–Reconfigurable Pipelined Datapath," *Lecture Notes in Computer Science 1142—Field-Programmable Logic: Smart Applications, New Paradigms and Compilers,* R.W. Hartenstein and M. Glesner, eds., pp. 126-135, Springer-Verlag, 1996.

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

[4] A. Abnous and J. Rabaey, "Ultra-Low-Power Domain-Specific Multimedia Processors," *Proc. IEEE VLSI Signal Processing Workshop,* Oct. 1996.

[5] Morpho Technologies, http://www.morphotech.com/, 2007.

[6] Stretch, Inc., http://www.stretchinc.com/, 2007.

[7] The Totem Project, http://www.ee.washington.edu/people/faculty/hauck/Totem/, 2007.

[8] K. Compton, "Architecture Generation of Customized Reconfigurable Hardware," PhD thesis, Dept. of Electrical and Computer Eng., Northwestern Univ., 2003.

[9] K. Eguro, "RaPiD-AES: Developing an Encryption-Specific FPGA Architecture," master's thesis, Dept. of Electrical Eng., Univ. of Washington, 2002.

[10] M. Holland, "Automatic Creation of Product-Term Based Reconfigurable Architectures for System-on-a-Chip," PhD thesis, Dept. of Electrical Eng., Univ. of Washington, 2005.

[11] S. Phillips, "Automating Layout of Reconfigurable Subsystems for Systems-on-a-Chip," PhD thesis, Dept. of Electrical Eng., Univ. of Washington, 2004.

[12] A. Sharma, "Place and Route Techniques for FPGA Architecture Advancement," PhD thesis, Dept. of Electrical Eng., Univ. of Washington, 2005.

[13] Z. Huang and S. Malik, "Exploiting Operation Level Parallelism through Dynamically Reconfigurable Datapaths," *Proc. Design Automation Conf.,* 2002.

[14] Z. Huang and S. Malik, "Managing Dynamic Reconfiguration Overhead in Systems-on-a-Chip Design Using Reconfigurable Datapaths and Optimized Interconnection Networks," *Proc. Design Automation and Test in Europe Conf. (DATE),* 2001.

[15] eASIC Corp., http://www.easic.com/, 2007.

[16] K. Compton and S. Hauck, "Totem: Custom Reconfigurable Array Generation," *Proc. IEEE Symp. FPGAs for Custom Computing Machines,* 2001.

[17] D.C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths," *Proc. Advanced Research in VLSI Conf.,* 1999.

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

[19] M. Scott, "The RaPiD Cell Structure," *Personal Comm.,* 2001.

[20] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A Time-Multiplexed FPGA," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines,* 1997.

[21] K. Compton, A. Sharma, S. Phillips, and S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems," *Proc. Int'l Conf. Field Programmable Logic and Applications,* pp. 59-68, 2002.

[22] S. Kirkpatrick, D. Gelatt Jr., and M.P. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, no. 4598, pp. 671-680, May 1983.

[23] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," *Proc. Int'l Workshop Field Programmable Logic and Applications,* pp. 213-222, 1997.

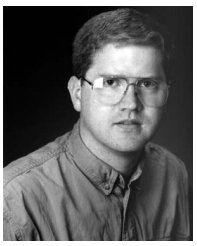
[24] U. Dorndorf and E. Pesch, "Fast Clustering Algorithms," *ORSA J. Computing,* vol. 6, no. 2, pp. 141-152, 1994.

[25] Xilinx, Inc., *Virtex-II Platform FPGAs: Detailed Description,* Xilinx, Inc., San Jose, Calif., 2002.

[26] Chipworks, Inc., *Xilinx XC2V1000 Die Size and Photograph,* Chipworks, Inc., Ottawa, Canada, 2002.

[27] A. Sharma, C. Ebeling, and S. Hauck, "Architecture-Adaptive Routability-Driven Placement for FPGAs," *Proc. Int'l Symp. Field-Programmable Logic and Applications,* 2005.

Katherine Compton received the BS, MS, and PhD degrees from Northwestern University in 1998, 2000, and 2003, respectively. Since January of 2004, she has been an assistant professor at the University of Wisconsin-Madison in the Department of Electrical and Computer Engineering. She and her graduate students are investigating new architectures, logic structures, integration techniques, and systems software techniques for reconfigurable computing. Dr. Compton serves on a number of program committees for FPGA and reconfigurable computing conferences and symposia. She is also a member of both the IEEE and ACM.



Scott Hauck received the BS degree in computer science from the University of California, Berkeley, in 1990, and the MS and PhD degrees from the Department of Computer Science, University of Washington, Seattle, in 1992 and 1995, respectively. He is an associate professor of electrical engineering at the University of Washington. From 1995 to 1999, he was an assistant professor at Northwestern University. His research concentrates on FPGAs, including architectures, applications, and CAD tools, reconfigurable computing, and FPGA-based encryption, image compression, and medical imaging. He has received a US National Science Foundation (NSF) Career Award, a Sloan Fellowship, and a TVLSI Best Paper Award. He is a senior member of the IEEE.