Automatic Design of Reconfigurable Domain-Specific Flexible Cores

Katherine Compton
Dept. of ECE
University of Wisconsin-Madison
kati@engr.wisc.edu

Scott Hauck
Dept. of EE
University of Washington
hauck@ee.washington.edu

Abstract

Reconfigurable hardware is ideal for use in Systems-on-a-Chip, as it provides both hardware-level performance and post-fabrication flexibility. However, any one architecture is rarely equally optimized for all applications. SoCs targeting a specific set of applications can greatly benefit from incorporating customized reconfigurable logic instead of generic FPGA logic. Unfortunately, manually designing a domain-specific architecture for every SoC would require significant design time. Instead, this article discusses our initial efforts towards creating a reconfigurable hardware generator capable of automatically creating flexible, yet domain-specific, designs. Our tests indicate that our generated architectures are more than 5x smaller than equivalent FPGA implementations, and nearly as area-efficient as standard cell designs. We also use a novel technique employing synthetic circuit generation to demonstrate the flexibility of our architecture generation techniques.

1 Introduction

Reconfigurable hardware shows great potential for systems-on-a-chip (SoCs), as it provides speeds similar to hardware execution but maintains a level of flexibility not available with more traditional custom circuitry [1]. This flexibility is the key to allowing both hardware reuse and post-fabrication modification. A widely available form of reconfigurable hardware is the field-programmable gate array (FPGA), and the structures within an SoC could be patterned after these designs. However, because of their highly flexible nature, FPGAs can incur significant area and speed penalties. If the SoC itself will be custom-fabricated, this presents the opportunity to customize the reconfigurable hardware to the target application domain. Domain-specific reconfigurable hardware provides flexibility within a domain, but with the extra unnecessary flexibility optimized away. This leads to reduced area and increased performance compared to a generic FPGA-style architecture. Unfortunately, manual design of a new reconfigurable architecture for each new domain would be disadvantageous in terms of design time and expertise required. In the Totem Project, we instead focus on the automatic creation of customized reconfigurable architectures, including high-level design [2][3][4], VLSI layout [5][6][7], and custom place and route tools [8][9].

This article focuses on the automatic creation of 1D datapath-style reconfigurable architectures that are flexible enough to handle changes in a domain's circuits, such as upgrades or bug-fixes, and even the introduction of entirely new circuits. We present a number of different tactics to generate these hardware designs, then test both the area-efficiency and flexibility of the generated architectures to verify the benefit of domain-specific hardware.

2 Background

The presented architecture generator creates designs in the style of the RaPiD architecture [10][11], shown in Figure 1. RaPiD is composed of coarse-grained word-sized computation units such as ALUs, Multipliers, and RAMs, arranged along a 1D axis. Routing is in the form of word-sized busses arranged in tracks running parallel to the axis. Each component contains multiplexers on each of its inputs which choose between the signals of each routing track, as well as demultiplexers on each of the outputs that allow the unit to directly output to any of the routing tracks. The two primary motivations for choosing the RaPiD system as a starting point, apart from its successes in the digital signal processing (DSP) area, are the 1D organization and the existing compiler. As with RaPiD, a 1D design is effective for the datapath-style computations that we initially target. More importantly, a 1D structure simplifies our task significantly, though in future efforts we will examine 2D designs. The existing compiler is important because this allows us to create benchmark applications.

As previously stated, RaPiD has proven itself to be a very good architecture for DSP applications. However, this architecture was manually designed, and does not have enough routing capability for a number of the benchmarks we use. Furthermore, RaPiD was designed to be suitable for executing a wide variety of circuits within the DSP

domain. On the other hand, our goal is to customize an architecture for a given application set, with some extra resources if desired for future flexibility, and to generate this architecture automatically.

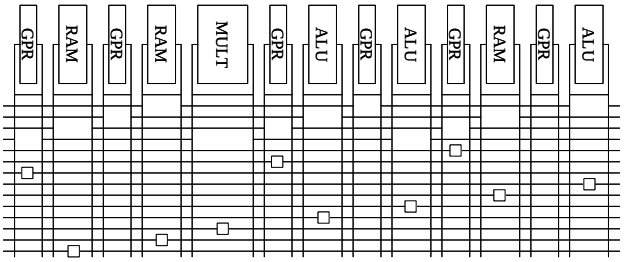

Figure 1: A single RaPiD cell [10][11]. Multiple cells tile horizontally to make a full architecture.

3 Architecture Generation

While RaPiD has a fixed logic mix and set of routing tracks, the Totem architecture generation algorithms can vary these particular features to achieve a more customized design. Essentially, RaPiD becomes one architectural instance that a flexible architecture generator could create. However, the generators should also be able to create architectures both larger and smaller, depending on the specified domain. For example, RaPiD was designed specifically for DSP operations. If a user only needs a small set of FIR filters, an architecture generator could automatically create an architecture more optimized for FIR filters than RaPiD. Likewise, if the tool is presented with a set of netlists which require more resources than the RaPiD architecture provides, the generation tool will still be able to create an architecture able to implement those netlists. In this section we briefly describe how we create the logic structure, then follow with an in-depth discussion of our routing architecture generation techniques.

3.1 Logic Generation

The logic generation step for flexible architecture generation is a slightly modified version of the logic generation technique from our previous work on configurable ASIC designs [2][12]. We profile the input circuit set that is the specification for the architecture, and determine the minimum number of each type of unit required to implement all of the given circuits (one at a time). These values are used to create the physical logic components for the flexible architecture. The user can specify if he or she wishes to add more. Simulated annealing [13] is used to simultaneously find a physical placement of the units and a mapping of the netlists to those physical components. Figure 2 shows two example netlists, and Figure 3 shows a possible logic architecture generated for those netlists. At this point, we have not yet created the routing structure. The physical placement and netlist mapping are needed to determine the connectivity requirements of the circuits for the later routing generation stage.

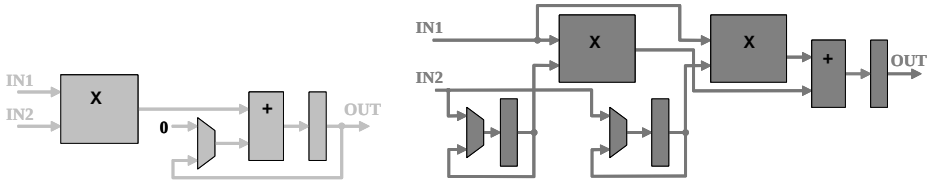

Figure 2: Two different example netlists, a multiply accumulate (light) and a 2-tap FIR filter (dark)

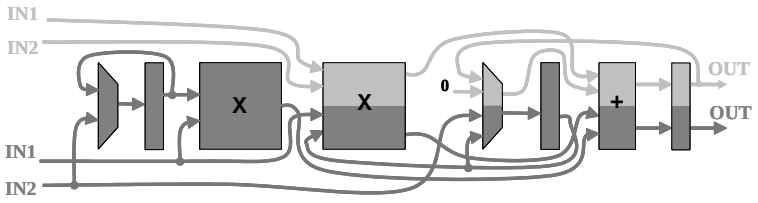

Figure 3: The logic structure created for the example netlists of Figure 1. The shading indicates the logic units used by each netlist. Only one netlist is active at any given time. Lines indicate communication needs.

3.2 Routing Generation

After the logic is generated and placed, a routing architecture generation algorithm heuristically creates the configurable routing structure based on the signal locations and lengths within the application netlists [3]. We have developed several different algorithms to do this. However, before discussing the actual routing generation algorithms, we first discuss common features and issues of the architecture generation algorithms.

3.2.1 Common Features

Architectural Style

The generated architectures are track-based architectures similar in style to RaPiD. Local tracks (the upper routing tracks in RaPiD, also shown in Figure 4a), are used for short connections. A special track, the topmost shown in Figure 1 and Figure 4a, containing "feedback" wires, as they only route from a unit's outputs to that unit's inputs. Distance routing tracks (the bottom tracks in RaPiD, also shown in Figure 4b) have bus connectors that connect or disconnect track wire segments, depending on how they are configured. These connectors are programmed independently, and provide an optional pipeline delay. Distance tracks allow for a great deal of routing flexibility, but bus connectors can add delay as a signal passes through them as well as a not-insignificant area penalty.

Each track has an associated *length* and *offset*. The length is based on the number of units spanned by the wires in the track, and can be determined by subtracting the indices of the furthest units a wire can reach. For example, a wire that spans from the outputs of unit 1 to the inputs of unit 3 would be considered a length-2 wire. The offset indicates the left-right shifting of wires within the track, and corresponds to the index position of the first break or bus connector between wires. Since all wires in a track have the same length, the offset of a track is always less than the length of the track. Figure 4 shows a variety of tracks labeled with their lengths and offsets.

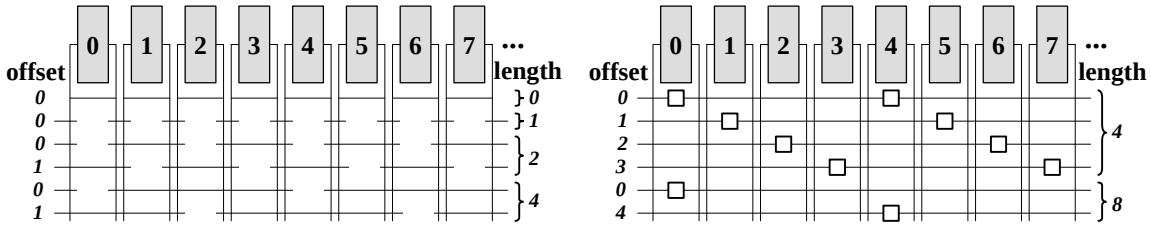

Figure 4: Examples of the different types of routing tracks, (a) local tracks, and (b) distance tracks with bus connectors (represented by the squares on the tracks). The “wire length” of each track is given to the right of the track.

For this study, we restrict allowable wire lengths based on experience with the RaPiD architecture. Local tracks can be no longer than length-8, as they are intended for short, fast connections. Distance wires must have lengths between 8 and 16 (inclusive). The minimum length restriction avoids adding too many bus connectors (and their associated area/delay penalty). The maximum length is because extremely long wires become the critical path. Our experiments indicated extremely few if any tracks were created longer than 16, so we chose 16 as the maximum.

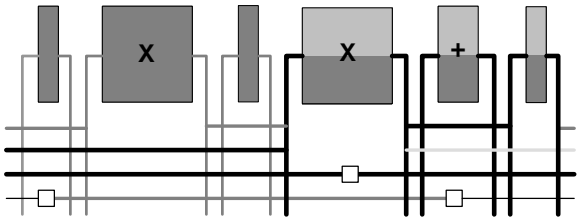

Figure 5: Flexible routing architecture created from the placement of Figure 3. Vertical lines represent the multiplexers and demultiplexers on every port of each unit. Black wires are used by both netlists of Figure 2, while the other colors indicate wires used only by the corresponding netlist.

The high-level operation of the architecture generation algorithms we present is similar—each iteratively adds routing tracks until all signals from all the netlists in the problem can be routed onto the architecture (though not necessarily all at the same time). An example routing architecture for the placement of Figure 3 is shown in Figure

5. After all the tracks have been added, the user is given the opportunity to increase the number of routing tracks beyond the generated architecture for optional additional flexibility.

Cost Function

While area and delay are important cost considerations, they are difficult to use as a cost function for routing architecture construction. For example, for local tracks, different track lengths or offsets do not have different area costs. However, these choices can affect the overall area of the final architecture if one length and offset combination is overall more *useful* for the signals that need to be routed. Therefore, instead of directly measuring the area or delay cost of a given track during the iterative routing generation process, we instead consider the cross-section of signals that are as of yet unroutable on the incomplete routing architecture. This value is calculated by finding the maximum unroutable signal cross-section (UCS) at any physical unit index for each individual netlist, then for each index, choosing the maximum value across all of the netlists. The maximum of the new values is the total UCS value that represents the "cost" during routing generation. Figure 6 illustrates the initial UCS calculation for the placement of Figure 3. This value is also the lower bound of the number of tracks that must still be added to the architecture at any given point to implement the source netlists. The initial UCS is the lower bound of the total number of tracks. This lower bound is, however, a very improbable solution due to the extremely high number of bus connectors that would almost certainly be required in order to have that few tracks.

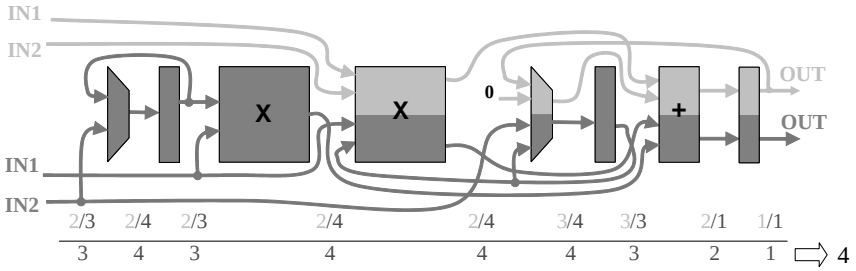

Figure 6: Calculation of the unroutable cross-section (UCS) for the placement of Figure 3. The cross-section is given for each netlist (light grey netlist / dark grey netlist) at each index. The maximum for the netlists appears below, and the overall UCS at the far right.

Fast Internal Router

In order to calculate the UCS, the generator must determine which signals are and are not routable. Because this is a frequent operation performed within nested loops, a very fast router is needed. Initially, a left-edge algorithm [14] was considered. However, this algorithm does not consider the suitability of a given wire segment for a given signal, only whether or not the signal fits in a given wire. While the algorithm is appropriate to test routing success or failure for complete architectures with uniformly-sized wire segments, it is not indicate how close an incomplete routing architecture is to implementing a given netlist. This could result in misguided track choices during iterative routing construction. For example, a circuit may have two signals: one relatively short, and the other relatively long. The short signal's leftmost connection may be one position to the left of the long signal's leftmost connection. With a left-edge routing algorithm, the short signal will be routed first. If a long-segmented track is created to implement the long signal, but that track is also able to implement the short signal, the short signal would be routed onto the track instead of the long one. This may deceive the generation algorithm into adding another long track for the still unroutable long signal instead of a more efficient short track for the short signal.

Therefore, we modify the left edge algorithm to greedily consider length-appropriateness during routing. Each time a signal/wire pair is considered, all other unrouted signals that could use the wire are also considered. The signal with the closest "fit" is routed onto the wire. If the original signal from the pair was not chosen, that signal is reconsidered on the next iteration. For the routing operation, each netlist is considered a separate problem—signals from different netlists can be assigned to the same wire.

3.2.2 Routing Generation Algorithms

Each of our three flexible reconfigurable routing generation algorithms represents a different point in the area/flexibility design space. The goal is to provide an SoC designer with a wider range of choices for flexible reconfigurable cores. If area is the primary consideration, and any additional netlists implemented on the

architecture are expected to be very similar to those in the specification set, the designer may choose a more area-efficient but less flexible design. However, if the future uses of the SoC are not completely known at design time, the designer may choose a more flexible reconfigurable core to provide future robustness. This section discusses all three generation algorithms, and presents pseudocode describing their operation.

Greedy Histogram

The Greedy Histogram (GH) routing generation algorithm attempts to customize a RaPiD-style routing architecture as much as possible to the input netlist set. Therefore, this algorithm does not restrict the offsets of the created tracks as the next two algorithms do. While this can negatively impact the architecture's flexibility to implement netlists beyond the input netlist set [4][15], the goal of the algorithm is to customize the routing architecture significantly for the given applications within the limitations of the previously described architectural style. The pseudocode for this algorithm is shown in Figure 7. In this algorithm, tracks are added one at a time within an infinite loop. The loop is broken when all of the netlists can be fully routed onto the architecture using the integrated fast router. The algorithm chooses the wire length of a "new" track based on a histogram of the lengths of the unroutable signals. The actual track creation method depends in part upon the wire length chosen, as indicated by the pseudocode in Figure 7. Note that the comparative "benefit" of one track over another is based on the relative reduction in the UCS, or the relative reduction in the number of unroutable signals if the UCS values are the same.

```
Greedy_Histogram()
    Let S = the set of all signals
    Let U = the set of unroutable signals (initially all signals)
    Let T = the set of tracks (initially empty)

    While U not empty
        Let H = histogram of U (unroutable signals) by signal length
        Let length = highest index of H that contains the max value in H

        If (length == 0)
            Add a feedback track to T
        Else if (length < MIN_DISTANCE_LENGTH)
            Try all possible offsets for a local track of the given length
            Add a local track of the given length to T with the offset providing the greatest benefit
        Else if (length > MAX_LOCAL_LENGTH)
            Try all distance routing lengths, and all possible offsets for each length
            Add a distance track to T with the (length, offset) combination providing the greatest benefit
        Else
            Find the best local track and distance track possibilities using the above techniques
            Add the track to T that provided the greatest benefit, choosing the local track if tied
        Route signals S onto tracks T, and update U accordingly
```

Figure 7: Pseudocode for the Greedy Histogram routing generation algorithm.

The next two algorithms focus on creating regular patterns in the routing architecture. Although we already require that all wires in a given track are the same length, the offsets chosen for different tracks may cause bus connectors or breaks between local routing wires to be unevenly distributed across the architecture. By contrast, the next two algorithms require even distribution of the bus connectors, breaks, and the different logic unit types. The necessary track offsets to accomplish this are determined automatically [4]. These algorithms also restrict wire lengths to powers of two. This restriction is common in FPGA architectures, and is intended to further generalize the architectures. Given the lack of length-8 local routing in the GH algorithm, we also remove this possibility. Thus, possible local track lengths are 0, 2, and 4, and possible distance track lengths are 8 and 16. Finally, logic resource types are also distributed evenly throughout the architectures.

Add Max Once

The Add Max Once (AMO) algorithm attempts to add as many of the "cheapest" tracks as possible to reduce the unroutable cross-section. Tracks are added from the shortest local length (fastest tracks) to the longest local length, then distance tracks. Tracks of the each type are added until no further reductions are possible with more tracks of

that type. One problem with this technique is that only one length of distance track will be considered, since with enough distance routing tracks of any length all signals can be routed by using the bus connectors to form longer wires when necessary. Length-8 distance routing tracks were chosen as the distance track length for this algorithm because experiments indicated that more length-8 tracks were generally created than length-16 using the other generation techniques. Also, restricting distance tracks to length-16 could result in the creation of too many tracks. The pseudocode for Add Max Once appears in Figure 8.

```
Add_Max_Once()
    Let S = the set of all signals
    Let U = the set of unroutable signals (initially all signals)
    Let T = the set of tracks (initially empty)

    For each local track length in increasing order (0,2,4)
        Add local tracks of the given length until the UCS could not be further reduced no matter how
            many more tracks of that length were added at this point
        Route signals S onto tracks T, and update U accordingly
    Add distance tracks of length 8 until all signals can be routed
```

Figure 8: Pseudocode for the Add Max Once regular routing architecture generation algorithm.

Add Min Loop

AMO tends to weight towards the use of distance routing tracks because it only considers each wire length and type combination once. However, it is possible that once a distance track is added, using additional local tracks will once again reduce the UCS. Therefore, the Add Min Loop (AML) algorithm has been created in an effort to promote local tracks over distance tracks. Pseudocode for this algorithm is given in Figure 9. AML also examines track types and lengths from "cheapest" to most "expensive", but after each addition the AML algorithm revisits cheaper track types and lengths to determine if any would now be useful. This algorithm is able to cope with different distance track lengths, and the order of tracks considered is from shortest local to longest local, then longest distance to shortest distance. Bus connectors consume area (priority is given to local tracks, followed by distance tracks with fewer connectors), and short wires are faster than long ones (priority within local tracks is given to shorter lengths).

```
Add_Min_Loop()
    Let S = the set of all signals
    Let U = the set of unroutable signals (initially all signals)
    Let T = the set of tracks (initially empty)

    While U not empty
        Route signals S onto tracks T, and update U accordingly
        For each local track length in increasing order (0,2,4)
            Add local tracks of the given length until the UCS could not be further reduced even with
                more tracks of that length, capping the number added at the length of the track
            If any tracks were added
                Remove all distance tracks and longer local tracks
                Go to the beginning of outer While loop
        For each distance track length in decreasing order (16,8)
            Add one distance track of the given length if it would reduce the UCS
            If a track was added
                Remove all longer distance tracks
                Go to beginning of outer While loop
        Since none of the attempts reduced the UCS, add one track of the same type and length as the
            attempt that most reduced the count of unroutable signals
```

Figure 9: The pseudocode for the Add Min Loop regular routing architecture generation algorithm.

AML considers a given track types and length to see if it will reduce the UCS. To further emphasize local routing, multiple local tracks may be added at the same time—up to as many as the length of the local track (providing the full range of possible offsets for that length). The algorithm adds as few tracks as possible up to that length value to

get the greatest UCS reduction. If adding the maximum number of local tracks of the given length does not reduce the UCS, the next (more expensive) track type/length is considered. Distance routing tracks are considered more expensive than local ones, so the algorithm only allows one distance routing track to be created per iteration. If none of the possibilities reduces the UCS, the algorithm breaks the stalemate by determining which of the attempts reduced the *count* of unroutable signals the most. A single track of this type and length is created, and a new iteration begins. Note that in the case of a tie, cheaper tracks are preferred over more expensive ones.

At times, the new combination of tracks may reduce the overall benefit of a previously-added expensive track if the combination of cheaper tracks in the architecture is able to provide the same UCS reduction. To compensate for this possibility, any tracks more expensive than the new tracks are removed whenever tracks are added to the architecture, After the track counts are modified accordingly, execution begins again from the top of the loop, and all track types are again considered in the same order as before.

4 Results

The next few sections present comparative results of flexible routing architecture generation algorithms. First we present information on the input netlists used by our algorithms. Then the area results of the algorithms are compared to each other as well as to other implementations of the input circuits. Finally, we compare the flexibility of the generated algorithms using a novel flexibility measurement technique [15].

4.1 Input Netlists

Eight different applications (each composed of two or more netlists) were used to evaluate the presented architecture generation algorithms. 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 applications were designed for the RaPiD architecture.

Table 1: Eight applications used to test generated architectures, each with 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

For comparison, we implemented these applications using several different techniques: standard cells, FPGA, RaPiD, and cASIC. Details of these implementations follow. For the standard cell flow, Verilog netlists were synthesized in Cadence to a TSMC 0.18µm process with 6 metal layers. Application areas are the sum of the areas required for each of the member netlists, except for the "collection" applications (FIR, Matrix, and Sort), which are given area estimates equal to the largest of the members. Verilog netlists were also synthesized to a Xilinx Virtex-II FPGA [16]. The XC2V1000 device has a slices:multipliers:RAMs resource ratio of 128:1:1, which we use as an atomic unit (tile) of FPGA area. The area of an individual tile (approximately 25K system "gates") was computed based on the die size and a die photograph [17], and then scaled to a 0.18µm process for a final tile size of 1.643mm^2. The total area required by an application is the maximum of the tile areas of its member netlists.

RaPiD [10][11] represents a partially-customized FPGA solution. The minimum number of RaPiD cells needed to implement each netlist was determined, and like the FPGA solution, application areas are the maximum areas across the member netlists. For the actual cell areas, we use areas from manual layouts of each of the logic and routing structures created in a TSMC 0.18µm process with 5 metal layers. Logic area is the sum of the logic unit areas, and routing area is the sum of the multiplexer, demultiplexers, and bus connector areas. Routing tracks are directly over the logic units in a higher layer of metal, and thus do not add area.

Finally, the netlists were also implemented using a "configurable ASIC" (cASIC) technique [2][12], which uses the same logic generation techniques presented in section 3.1. The cASIC technique uses clique partitioning to create very customized routing structures. In this process, the netlist signals are represented by nodes in a graph. The nodes

are then grouped into cliques based on similarities in their locations and terminals, and no more than one signal from each netlist can be in a given clique. A physical wire is created for each clique to implement the relevant signals. cASICs can then implement any netlist from their specification set, but likely no others. The benefit of cASICs is that they represent highly area-efficient design points, capturing the area savings of hardware reuse through run-time reconfiguration, while minimizing the overhead of programmability. Area estimates for cASIC implementations are based on the same manual layouts used for RaPiD, and are given for comparison.

4.2 Track Count Analysis

Table 2 lists the number of routing tracks created by each flexible routing generation algorithm for each application, along with lower bounds represented by the UCS values of the original placements before routing generation. These bounds are most likely infeasible due to the large number of bus connectors required for the increased wire sharing between signals from different netlists. The Add Max Once (AMO) and Add Min Loop (AML) algorithms have the same lower bound, as they use the same placement techniques. The Greedy Histogram (GH) method generally has a lower cross-section because it achieves a "better" placement by not restricting the locations of logic units. Each algorithm results in a track count within a factor of 1.5 to 2 of the lower bound. The more regular algorithms tended to come closer to their lower bounds. We expect this is due in part to the sometimes higher lower bound of these algorithms, and part due to the unexpected GH behavior discussed in the next section.

Table 2: The number of routing tracks created for each application by each routing generation algorithm. Lower bounds are based on the original unroutable signal cross-sections of the placements.

		Radar	OFDM	Camera	Speech	FIR	Matrix	Sort	Image	Average Factor
GH	Actual	17	34	24	25	18	24	21	21	1.86
	Bound	11	18	13	13	11	10	11	12	
	Factor	1.55	1.89	1.85	1.92	1.64	2.40	1.91	1.75	
AMO	Actual	18	28	25	24	17	21	21	17	1.55
	Bound	13	21	16	17	11	10	13	12	
	Factor	1.38	1.33	1.56	1.41	1.55	2.10	1.62	1.42	
AML	Actual	18	32	26	27	18	18	21	20	1.61
	Bound	13	21	16	17	11	10	13	12	
	Factor	1.38	1.52	1.63	1.59	1.64	1.80	1.62	1.67	

4.3 Area Analysis

The areas of the flexible architectures are computed using the same methods as the cASIC area computations. Wire cross-sections of greater than 24 can increase the height of the architecture beyond the logic height, and therefore increase the total area of the architecture. Figure 10 graphs the areas of the generated flexible architectures for eight different application domains. While the areas of the architectures created by the three different algorithms are close, there are some notable differences. GH was intended to create architectures less flexible, but more optimized (smaller), than the two regular routing algorithms. In most cases, this is true. However, there are a number of applications for which GH produces architectures that are *larger* than those created by the other algorithms. One likely explanation is that GH is too greedy in track creation. For example, if the histogram indicates that the most common signal length is 11, and the second-most common signal length is 12, the algorithm will create a length-11 track even though a length-12 track may be a better choice, as it can also implement length-11 signals. This issue could lead to the creation of more tracks than necessary.

AMO generally results in architectures with fewer tracks but more bus connectors than AML, as demonstrated by correlating the results in Table 2 with those in Figure 10. This is logical, as AML places a greater emphasis on local track creation, but may create more overall tracks as a result. For the Radar and Sort applications, AMO and AML create architectures with the same number of tracks. However, the areas of the AMO architectures are slightly higher due to more bus connectors. AML creates smaller architectures for the Speech and Camera applications as well, even though it creates more tracks. In other cases where the AML area is greater than the AMO area, the AML algorithm created a significantly greater number of tracks, increasing the width of multiplexers and demultiplexers, and sometimes the height of the architecture. This indicates that track minimization, while not the only important goal, is still relevant. We expect that an updated algorithm balancing the bus connectors and track count could create smaller architectures than AMO and AML in most if not all cases.

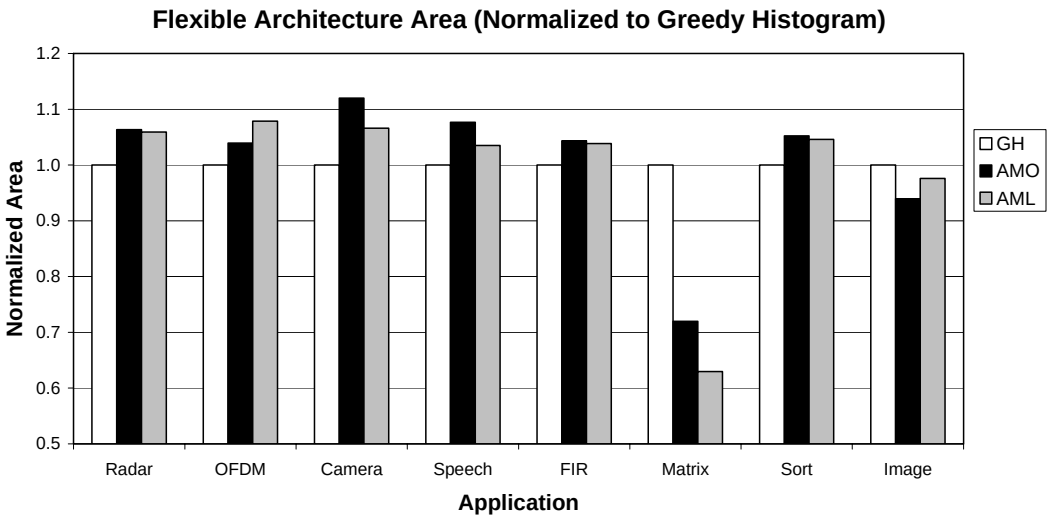

Figure 10: Comparative area results of the flexible routing generation algorithms, Greedy Histogram (GH), Add Max Once (AMO), and Add Min Loop (AML). The areas have been normalized to the GH results.

Table 3 lists the areas of the architectures created using the different flexible routing generation heuristics, with the corresponding standard cell, FPGA, RaPiD, and cASIC areas listed for comparison. These results are summarized in Table 4. As expected, customized flexible architectures are smaller than FPGA implementations—from a 5.3x to a 5.5x area improvement. These results highlight the area benefits of optimized reconfigurable computation and routing structures. Note that the FPGA architectures are only just over a factor of 7x larger than standard cell implementations. This is likely due to advances in FPGA design such as embedded multipliers.

Table 3: Areas, in mm^2, of the eight applications from Table 1 as implemented using the flexible routing generation algorithms and the comparative techniques discussed in section 4.1.

		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	---
cASIC	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.082	0.658	0.570	0.486	0.502	0.115	0.265	0.466
	Total	1.515	4.776	2.747	13.235	2.243	1.239	1.458	2.083
GH	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	1.271	15.896	5.651	22.202	1.737	2.138	2.110	2.302
	Total	2.705	20.014	7.829	34.950	3.478	3.261	3.303	3.919
AMO	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	1.443	16.682	6.591	24.886	1.888	1.224	2.283	2.065
	Total	2.877	20.800	8.768	37.635	3.630	2.347	3.476	3.681
AML	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	1.431	17.470	6.168	23.430	1.870	0.930	2.262	2.208
	Total	2.865	21.588	8.346	36.179	3.612	2.053	3.455	3.825

For applications where a RaPiD implementation was possible, the flexible architectures ranged from 56%-58% of the required RaPiD area. When RaPiD was not able to implement an application, the flexible generation algorithms were still able to create an architecture. Again, this is a key feature of automatic architecture generation—the logic and routing resource mix can be customized to the needs of the specification, and are not limited by a static design. Surprisingly, the flexible architectures were also on average close in area to the corresponding standard cell layouts. This highlights the value of reusing expensive hardware resources across netlists.

The added flexibility of flexible routing structures does significantly increase the areas over those of cASIC designs, but the results indicate this area increase does not overwhelm the area savings of hardware reuse. More importantly,

the flexible architectures have the ability to implement netlists beyond the implementation set, while the cASIC implementations do not. The next section discusses the flexibility of these architectures in further depth.

Table 4: A summary of the results from Table 3. Area improvements were first calculated for each application, then averaged across all applications.

Improvement Over Std Cells	
Method	**Area**
FPGA	0.20
RaPiD	0.48
cASIC	2.04
GH	0.90
AMO	0.91
AML	0.92

Improvement Over RaPiD	
Method	**Area**
Std Cells	2.34
FPGA	0.38
cASIC	3.68
GH	1.72
AMO	1.71
AML	1.77

Improvement Over FPGA	
Method	**Area**
Std Cells	7.20
RaPiD	4.01
cASIC	11.86
GH	5.25
AMO	5.37
AML	5.53

Improvement Over AMO	
Method	**Area**
Std Cells	1.37
FPGA	0.24
cASIC	0.60
Clique Avg	2.49
GH	1.01
AML	1.02

4.4 Flexibility Analysis

When designing a reconfigurable architecture with the goal of implementing multiple circuits, flexibility is a key concern. Unfortunately, there is not yet an accepted method for measuring flexibility. Commercial FPGA devices are frequently measured by "gate count", but this only measures logic capabilities, not routing flexibility. Furthermore, many consider the method used to count the gates as suspect. Gate count is even less appropriate for domain-specific reconfigurable architectures, where flexibility is not a simple sum of the number of configuration points or generic logic structures, but instead the ability of a design to implement a particular type of circuit. If an architecture needs to implement a circuit with three ALUs, but contains twenty multipliers and no ALUs, that architecture should not be considered "flexible" enough for its purpose despite having much logic.

For architectural exploration and analysis of reconfigurable structures, flexibility is a key metric. If the architecture cannot implement the circuits currently needed or those predicted as necessary in the future, that architecture is no more useful than one that violates area or timing constraints. This section discusses the problem of flexibility analysis. First we present a very simplistic and straightforward method to test architectural flexibility, and present flexibility results for the generated architectures. We then propose a new, more thorough, flexibility testing technique based on synthetic circuit generation, and apply it to the generated architectures as well.

4.4.1 Simple Flexibility Tests

A quick (but not completely effective) flexibility test is to attempt to place and route all 26 of our netlists onto each generated architecture. The reason this test is inadequate is that in many cases, the circuits attempted on each architecture are actually outside the domain the architecture targets. A failure to implement these circuits may not indicate a lack of flexibility within a domain. However, the test does provide some insight into the general flexibility of the architectures. The results for these tests are shown in Table 5. If a netlist failed placement and/or routing, it was attempted on a larger architecture, where the quantity of logic resources was increased by 10 or 20 percent, as indicated in the table. An architecture with a greater number of logic resources can sometimes allow for an improved placement, which in turn can result in an easier routing operation.

In many cases, netlists can be placed and routed onto architectures that have sufficient physical logic. One interesting data point is the one FIR filter netlist that will place and route on the AMO and AML architectures, but requires 10% more logic for the GH architecture. The regular design of AMO and AML architectures may contribute to their ability to implement that particular FIR filter without an increase in logic. Naturally, larger netlists tend to result in architectures capable of implementing the other benchmarks, as they naturally require more logic and routing resources. For example, the Camera and Image applications were able to implement far more netlists than the Matrix or Sort applications. The Sort application architectures are also crippled in this experiment by a complete lack of multiplier units, as they were unnecessary for the member netlists. Therefore even with a large % increase in architecture logic, any netlist with a multiplier will fail to place and route. Again, this failure does not necessarily reflect domain flexibility, as additional sorting netlists are unlikely to require multipliers.

Table 5: Simple flexibility test of the generated architectures. All 26 netlists were tested on all architectures, which were increased in size on a % basis if necessary. Table rows indicate: the # of source netlists for the architectures (SRC), the # that place and route without additional logic, the # that require a 10% or 20% logic increase, and the # that fail even with 20% more logic.

	Radar			Camera			OFDM			Image			DCT/FFT			FIR			Matrix			All Sort		
	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML
SRC	3	3	3	3	3	3	2	2	2	5	5	5	3	3	3	6	6	6	5	5	5	4	4	4
0	8	8	8	18	18	18	16	16	16	11	11	11	9	9	9	5	5	5	3	4	4	1	1	1
10	0	0	0	2	2	2	1	1	1	4	4	4	0	0	0	7	7	7	1	0	0	0	0	0
20	0	0	0	1	1	1	0	0	0	0	0	0	3	3	3	0	0	0	0	0	0	0	0	0
Fail	15	15	15	2	2	2	7	7	7	6	6	6	11	11	11	8	8	8	17	17	17	21	21	21

4.4.2 Detailed Flexibility Analysis

As stated earlier, there are no accepted existing techniques to measure flexibility of reconfigurable architectures. However, in order to truly evaluate and compare reconfigurable architectures, flexibility must be considered. For domain-specific reconfigurable architectures, flexibility can be defined as the ability to implement circuits from that domain. The more (and varied) circuits within the domain that can be implemented, the more flexible the architecture. Determining "flexible enough" must be done the same way as "small enough" and "fast enough"—based on the design specification and the anticipated future needs of the hardware. However, one must still measure what the flexibility is before determining if it is sufficient.

This measurement can be based on creating an architecture from a domain, and testing whether or not other circuits from that domain will fit. In reality, any circuits that are known from the domain would be used in the specification of the architecture. It is the *unknown* circuits in the domain that we wish to test on the architecture, but are thwarted by the fact that they are, in fact, unknown. We can, however, emulate this process through the use of a synthetic circuit generator. The known circuits of the domain can be profiled to determine a range of characteristics for the domain. Then, synthetic circuits can be created with characteristics selected randomly from within the range. To this end, existing circuit generation techniques [18][19] have been modified for the heterogeneity, coarse granularity, and structure of RaPiD netlists. The synthetic RaPiD netlists can then be used to provide a greater sample set of netlists from the domain for flexibility testing [15]. This section presents the results of flexibility tests of the architecture generation algorithms using this technique.

Single Circuit Flexibility

We first tested the flexibility of the architecture generation methods for single circuit generation. Ten synthetic netlists were created based on each of our 26 netlists. Each architecture generation algorithm was used to create an architecture for each synthetic netlist, for a total of 780 architectures. Generated netlists were forced to have the same logic resources as the parent netlist so only routing flexibility is tested. We attempted to place and route the original parent netlists on each of their corresponding 30 architectures. The results, presented in Table 6, highlight the flexibility differences of the architecture generation methods.

GH only results in architectures sufficiently flexible to implement the original circuit under 20% of the time. On the other hand, a significant percentage of the AMO and AML architectures created from synthetic netlists can successfully implement the original parent netlists—AMO has a 94% success rate, while AML has an 89% success rate. In a few cases, the original parent circuit could not be placed and routed onto architectures created from itself using GH or AML, as shown by the "Orig" column of Table 6. The overall results from this experiment indicate that AMO creates architectures inherently more flexible than either of the two other methods, likely from a combination of regular structure and the greater use of bus connectors.

Table 6: Success rates (%) of routing the original parent netlist on architectures created from synthetic circuits. The "Orig" column indicates if the parent netlist can be implemented on an architecture created directly from itself using the given technique.

	GH		AMO		AML	
	Orig	%	Orig	%	Orig	%
1d_dct40	Y	40	Y	80	N	60
color_interp	Y	30	Y	100	Y	100
decnsr	N	10	Y	90	N	30
fft16_2nd	Y	10	Y	100	Y	100
fft32	Y	10	Y	100	Y	100
fft64	Y	10	Y	100	Y	100
firsm	Y	10	Y	100	Y	100
firsm2	Y	10	Y	100	Y	100
firsm3	Y	0	Y	100	Y	100
firsymeven	Y	0	Y	100	Y	100
firtm_1st	Y	0	Y	100	Y	90
firtm_2nd	Y	0	Y	80	Y	90
img_filt	Y	20	Y	100	Y	100
limited	Y	0	Y	100	Y	100
limited2	Y	30	Y	60	Y	60
log32	Y	90	Y	100	Y	100
matmult	Y	0	Y	90	Y	100
matmult4	N	0	Y	100	Y	100
matmult_bit	Y	0	Y	100	Y	100
med_filt	Y	40	Y	100	Y	90
psd	Y	100	Y	100	Y	100
sort_g	Y	10	Y	100	Y	100
sort_rb	Y	40	Y	100	Y	100
sort_2d_g	Y	10	Y	100	Y	90
sort_2d_rb	Y	10	Y	100	Y	100
sync	Y	0	Y	40	Y	10
AVERAGE	18.5		93.8		89.2	

Domain Flexibility

Next we expanded the flexibility testing to look at architectures generated from multiple synthetic circuits, which were in turn generated from profiling our different application domains from Table 1. In this case, the characteristics for each generated circuit fall within the ranges for that characteristic observed during profiling. Again, to ensure sufficient logic is available and test only routing flexibility, the minimum logic requirements of each real netlist in the domain was also profiled, and these quantities also provided to the architecture generation algorithms. When a specification set of synthetic circuits did not meet a minimum logic requirement, the architecture generation algorithms added sufficient logic of the correct type to guarantee sufficient logic for the original netlists. Those original domain netlists were then placed and routed onto the architectures. This process was repeated ten times for each domain, and the results are given in Table 7.

This set of domain tests also demonstrates the flexibility differences of the flexible architecture generation algorithms. GH has the lowest flexibility, successfully routing 91.6% of netlists onto the architecture created for their domain. AML had a mid-level flexibility, with a success rate of 99.4%. AMO has the highest flexibility, with 99.7% of the netlists successfully routed. These results mirror the expected flexibility of the three algorithms. GH is inherently less flexible, as it attempts to customize the routing architecture as much as possible to the specified netlists. AMO has the highest flexibility, as it emphasizes the use of distance (segmented) routing tracks, which inherently permit a wider variety of choices for a routing signals. AML creates a regular routing architecture, but attempts to use less area than AMO by focusing on local (non-segmented) routing whenever possible, and thus is somewhat less flexible.

Table 7: Success rates of routing original domain netlists on architectures created by the routing architecture generation algorithms from a set of synthetic benchmarks created by profiling that domain.

Application	Netlists	GH	AMO	AML
Radar	decnsr	100.0	100.0	100.0
	fft16_2nd	10.0	100.0	100.0
	psd	100.0	100.0	100.0
OFDM	sync	100.0	100.0	100.0
	fft64	60.0	100.0	90.0
Camera	color_interp	100.0	100.0	100.0
	img_filt	100.0	100.0	100.0
	med_filt	100.0	100.0	100.0
Speech	log32	80.0	100.0	100.0
	fft32	100.0	100.0	100.0
	1d_dct40	100.0	100.0	100.0
FIR	firsm	100.0	100.0	100.0
	firsm2	100.0	100.0	100.0
	firsm3	100.0	100.0	100.0
	firsymeven	50.0	100.0	100.0
	firtm_1st	100.0	100.0	100.0
	firtm_2nd	100.0	100.0	100.0
Matrix	matmult	90.0	100.0	100.0
	matmult4	90.0	100.0	100.0
	matmult_bit	90.0	100.0	100.0
	limited	100.0	100.0	100.0
	limited2	100.0	100.0	100.0
Sort	sort_g	100.0	100.0	100.0
	sort_rb	100.0	100.0	100.0
	sort_2d_g	100.0	100.0	100.0
	sort_2d_rb	100.0	100.0	100.0
Image	med_filt	70.0	90.0	90.0
	matmult	100.0	100.0	100.0
	firtm_2nd	100.0	100.0	100.0
	fft16_2nd	100.0	100.0	100.0
	1d_dct40	100.0	100.0	100.0
AVERAGE		91.6	99.7	99.4

5 Conclusions

We have presented three different algorithms to generate domain-specific reconfigurable architectures for SoCs, ranging in optimization and flexibility. The area comparisons presented here demonstrate the area benefits of customization. The flexible domain-specific architectures were on average 5.3-5.5x smaller than the equivalent FPGA implementations. While FPGAs are important for situations in which one cannot reliably predict how the hardware will be used, they are comparatively inefficient for circumstances when some or all characteristics of the target applications are known. The RaPiD project addresses this issue to some degree by employing coarse-grained computational units, but this architecture has only limited customization through varying cell count. Our algorithms were able to create architectures nearly half the area required by RaPiD. Furthermore, these algorithms are able to create architectures large enough to implement applications that do not fit in the current RaPiD design.

We have also demonstrated a new technique to measure the flexibility of domain-specific reconfigurable architectures and architecture generation algorithms. Our flexibility results confirmed our expectations that the higher the specialization, the lower the flexibility. Furthermore, the comparative differences in flexibility between AMO and AML architectures highlighted the value of bus connectors, a value not truly measurable using existing architecture comparison techniques. Therefore, flexibility testing is essential if SoC designers are to choose reconfigurable cores not only on an area, power, and performance basis, but also with an eye for possible future changes or additions to the netlists implemented on the architecture.

Acknowledgments

This research was made possible by grants from NSF, NASA, and Motorola. Katherine Compton was also supported by a Cabell Fellowship and a UPR grant from Motorola. Scott Hauck was also supported by a Sloan Research Fellowship and an NSF CAREER award. Thanks to Ken Eguro and Kim Motonaga for their help gathering the comparative implementation data, Shawn Phillips for updating the RaPiD unit layouts, and Akshay Sharma for the Totem/RaPiD place and route tool. Thanks also to the RaPiD group, particularly Carl Ebeling, Darren Cronquist, and Chris Fisher, for the use of the RaPiD compiler, original logic unit layouts, and general assistance.

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

[3] 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.

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

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

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

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

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

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

[10] 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.

[11] 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.

[12] K. Compton, S. Hauck, "Automatic Design of Area-Efficient Configurable ASIC Cores", *submitted to IEEE Transactions on Computers*, 2005.

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

[14] A. Hashimoto, J. Stevens, "Wire Routing by Optimizing Channel Assignment within Large Apertures", *Proceedings of the 8th ACM Design Automation Workshop*, pp. 115-169, 1971.

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

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

[17] Chipworks, Inc., "Xilinx XC2V1000 Die Size And Photograph", Chipworks, Inc., Ottawa, Canada, 2002.

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

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