Totem: Custom Reconfigurable Array Generation

Katherine Compton
Northwestern University
Evanston, IL 60208
kati@ece.nwu.edu

Scott Hauck
University of Washington
Seattle, WA 98195
hauck@ee.washington.edu

Abstract

Reconfigurable hardware has been shown to provide an efficient compromise between the flexibility of software and the performance of hardware. However, even coarse-grained reconfigurable architectures target the general case, and miss optimization opportunities present if characteristics of the desired application set are known. We can therefore increase efficiency by restricting the structure to support a class or a specific set of algorithms, while still providing flexibility within that set. By generating a custom array for a given computation domain, we explore the design space between an ASIC and an FPGA. However, the manual creation of these customized reprogrammable architectures would be a labor-intensive process, leading to high design costs. Instead, we propose automatic reconfigurable architecture generation specialized to given application sets. The Totem custom reconfigurable array generator is our initial step in this direction.

Introduction

One of the primary difficulties of using FPGAs and reconfigurable systems for DSP, networking, and other applications is the fine-grained nature of many of these devices. Common operations such as multiplication and addition would greatly benefit from more efficient coarse-grained components. A number of reconfigurable systems have therefore been designed with a coarse-grained structure. These structures target the general case, attempting to fulfill the computation needs of any application that may be needed. However, because different application types have different requirements, this creates a large degree of wasted hardware (and silicon area) if the applications run on the system are constrained to a limited range of computations. While the flexibility of general-purpose hardware has its place for situations where the computational requirements are not known in advance, frequently specialized on-chip hardware is used to obtain greater performance for a specific set of compute-intensive calculations.

While general-purpose reconfigurable systems have exhibited their value in the applications mentioned above [Compton00], we believe that performance gains can be further improved within a smaller area if the algorithm types are known prior to fabrication. By generating a custom reconfigurable array for a computation domain, we can reduce the amount of "useless" hardware and programming points that would otherwise occupy valuable area or slow the computations. Architectures such as RaPiD [Ebeling96], PipeRench [Goldstein99], and Pleiades [Abnous96] have made progress in this direction by targeting multimedia and DSP domains. The RaPiD group has also proposed the synthesizing of custom RaPiD arrays for different application sets [Ebeling98, Cronquist99b]. In many ways this effort can be viewed as a first step in this direction.

We are working towards the automatic creation of custom reconfigurable architectures designed specifically for a given range of computations being performed. These application domains could include cryptography, DSP or a subdomain of DSP, specific scientific data analysis, or any other compute-intensive area. This concept is different from traditional ASICs in that we retain some level of hardware programmability. This programmability gives the custom architecture a measure of flexibility beyond what is available in an ASIC, as well as providing the benefits of run-time reconfigurability. Run-time reconfiguration can then be employed to allow for near ASIC-level performance with a much smaller area overhead due to the re-use of area-intensive hardware components. Essentially, depending on the needs of the algorithms and the stated parameters, this architecture generation could potentially provide a design anywhere within the range between ASICs and FPGAs. Very constrained computations would be primarily fixed ASIC logic, while more unconstrained domains would require near-FPGA functionality. This custom array would then be a computational unit within an ASIC fabricated for the tasks needed. Because the ASIC will be custom-designed, we can also optimize the array for the application domain.

Specialized reconfigurable architectures, 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 or sets desired. Unfortunately, this is contrary to one of the basic principals of FPGAs and reconfigurable hardware, which is quick time-to-market with low design costs. Therefore, we have started the Totem project – an endeavor to automatically generate custom reconfigurable architectures based on an input set of

applications, and therefore greatly decrease the cost of new architecture development. This paper presents our initial progress in this arena.

Background

We are designing the Totem generator to leverage the coarse-grained nature of many compute-intensive algorithms. This involves using large word-width computation structures such as adders and multipliers, as well as word-width routing structures. Because we are operating on word-sized data, a one-dimensional structure is not only less complex to generate, but also efficient. Essentially, the bit order of the data words does not change within the routing structure, leading naturally to a computational flow along one axis. Changing the direction of these buses would require additional routing area for the wire bends. Also, using available RAM units, 2D computations can be transformed into the 1D domain [Cronquist99a].

This leads us to consider the RaPiD architecture [Ebeling96] as a basis for our first efforts in the Totem project. RaPiD not only bears a strong similarity to the parameters mentioned above, but also has a compilation engine and a library of completed netlists for the architecture. It uses components such as multipliers, adders, and RAMs to allow for efficient computation of algorithms operating on word-sized data. These components are arranged along a one-dimensional axis, as shown in Figure 1. Signals travel horizontally along the routing, with vertical routing used only to provide connections between buses and computational components. The RaPiD compiler operates on application descriptions, converting them to netlists mapped to component types present in the RaPiD architecture. For more detail on the specifics of the RaPiD architecture and compiler, please refer to one of the papers on the subject [Ebeling96, Cronquist99a].

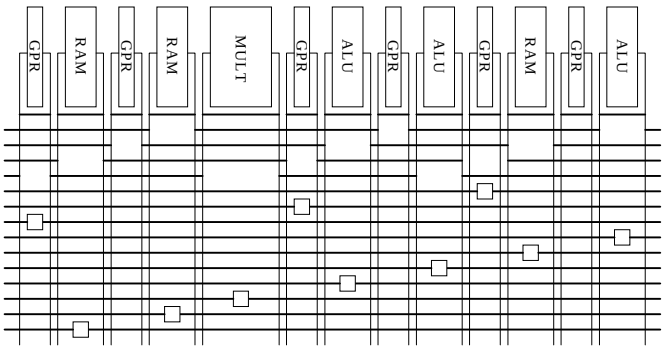

Figure 1: A block diagram of a single cell from the RaPiD architecture [Ebeling96, Scott01]. 16 cells are tiled along the horizontal axis to form the full architecture.

Using the RaPiD netlists and architecture style as a starting point, we have created an architecture generator which reads in the compiled RaPiD netlists, performs profiling, and creates a custom one-dimensional datapath capable of executing any of the input netlists. Because this hardware is optimized to the particular application set, it should have a smaller area and delay cost than a generic architecture implementing the same applications. However, by providing a reconfigurable interconnect and programmable logic units many of the benefits of reconfigurability are retained.

Architecture Generator

We have created an initial Totem architecture generator for compiled RaPiD netlists (in standard RaPiD netlist format) that creates a computational structure and routing fabric optimized for the given group of netlists. Like RaPiD, the architecture we generate is coarse-grained, consisting of components such as multipliers and adders, with a one-dimensional routing structure. Again, the motivating factor here is to automatically create architectures for input application sets that leverage the benefits of both ASICs and reconfigurable implementations. These architectures aim to have a higher performance and smaller area than possible in a general-purpose reconfigurable architecture implementation. The custom architectures also use the reconfigurable aspect to retain a measure of flexibility within the architecture and the area advantages of hardware re-use.

The architecture generation occurs in two distinct phases. In the placement stage of the generation we determine the computation needs of the algorithms, create the computational components (ALUs, RAMs, multipliers, registers, etc), and order the physical elements along the one-dimensional datapath. The individual instances of component use within the netlists must be assigned to the physical components. This binding also occurs in the placement stage. In the routing stage, we create the actual wires and muxing/demuxing needed to interconnect the different components, including the I/Os. These phases are described in depth in the next sections.

To determine the quality of our automatically generated structures, we measure the area of these specialized structures, and compare it to that of a basic, generic RaPiD architecture (pictured in Figure 1) implementing the same netlists. In order to better analyze the results, we will consider a number of methods of routing structure generation. We then compare the areas of the architectures we generate to a calculated lower bound value for each set of netlists.

Placement

The ordering of the physical elements within our generated structure is determined via simulated annealing. This algorithm operates by taking a random initial placement of physical 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. In order to avoid settling into a local minima in the placement space, non-improving moves are also sometimes accepted. The probability of accepting a "bad" move is governed by the current temperature. This temperature is initially high,

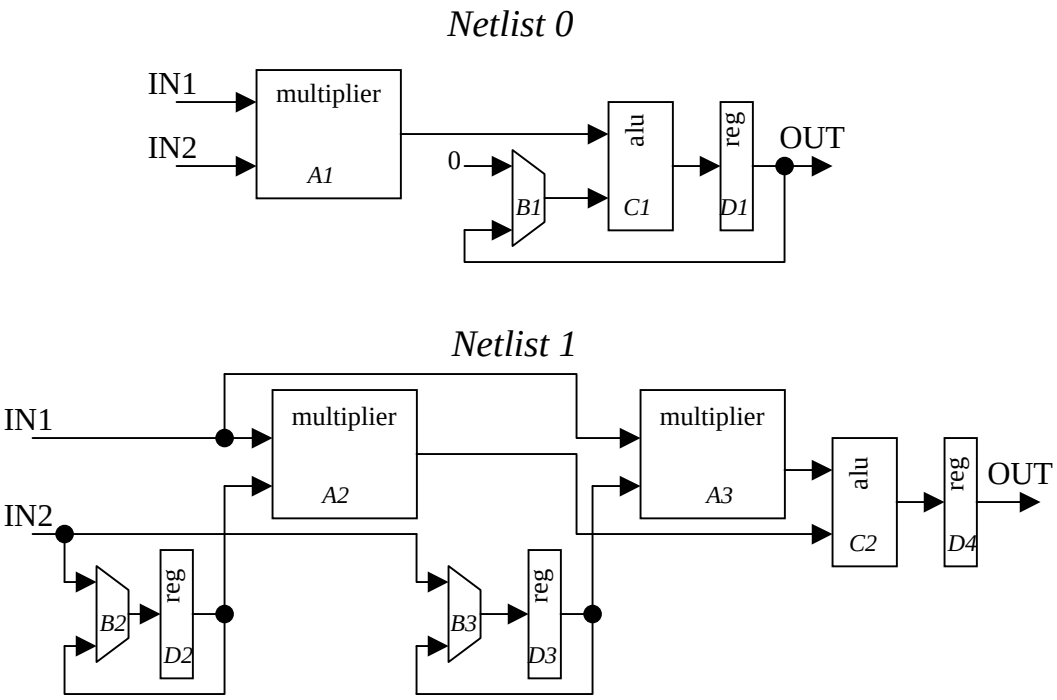

Figure 2: Two different netlists that could be implemented together on a custom architecture. Netlist 0 is a multiply-accumulate circuit, while netlist 1 is a simple 2-tap FIR filter, using constants stored in the registers.

causing a larger number of bad moves to be accepted, and is gradually decreased until no bad moves will be accepted. A large number of moves are attempted at each temperature.

While the general practice of simulated annealing is relatively straightforward, there is an additional consideration for our architecture generator. We would like to base our cost estimate of the quality of the placement on the area and delay of the completed circuit. However, knowledge of the routing structure, which has not yet been created, is then required. Instead we base our calculations on the individual signals within the netlists. But the extents and locations of these signals are not known until the instances of each netlist are assigned (bound) to physical components. The best binding depends on the given placement, and the best placement depends on the given binding.

This is different from typical placement problems where binding is either not applicable (ASIC) or is determined post-fabrication (FPGA). Our binding problem is also distinct from traditional binding of components within an FPGA because here we have the flexibility to perform physical movements (changing the location of specific hardware resources), whereas the architecture in standard FPGAs is traditionally fixed by the manufacturer. Additionally, we know in advance at least a subset of our target netlists, and can use this information to judge the quality of the layout. For our application, the best component binding and the best physical placement are interdependent, and therefore we perform these two operations simultaneously.

Consequently, we extend the simulated annealing framework to solve not only the physical placement problem, but the binding problem. The instances of each netlist are arbitrarily assigned initial bindings to corresponding physical components. Then, we create an additional type of "move" to be attempted within the simulated annealing algorithm – rebinding an instance of a circuit component of a single netlist to a different physical component. The probability of attempting a re-binding versus a physical component movement is equal to the fraction of total components that are instances instead of physical structures. Figure 2 shows two netlists that could be used to generate a custom architecture, and Figure 3 shows a reasonable placement. Note the difference between instances and physical components. For example, instances D1 and D4 should likely be in the same physical register because they share an input and an output. But which physical register is immaterial, provided it is near the ALU.

Our current Totem implementation uses the maximum number of each resource in any of the netlists to determine the number of that type of physical component to instantiate. This represents the minimum number of these components necessary to execute the netlists. Future Totem implementations will provide the potential to use a larger number of components than strictly required by the netlists if it will improve area (particularly routing and multiplexing) or delay results. For now, however, using the minimum number of each type of component allows us to examine one end of the design space, and provides a simple method to determine the computational requirements of the circuit. Once these

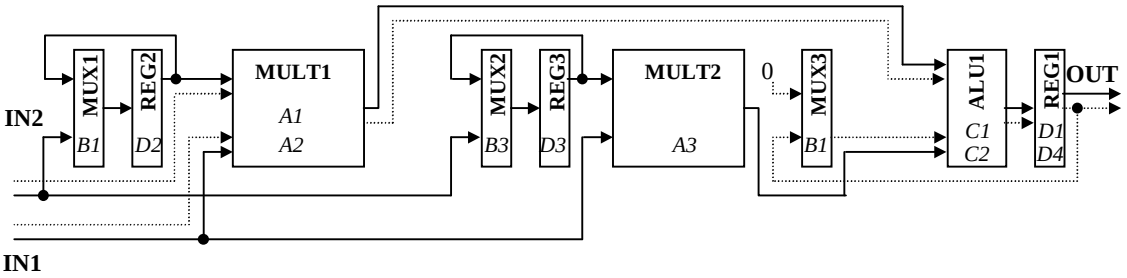

Figure 3: A reasonable placement for a custom architecture with both netlists from Figure 2. Netlist 0's signals are shown as dotted, and Netlist 1's signals are solid. The bold I/O lines connect to both netlists. Wires have not yet been created. The instance names from Figure 2 are shown inside the boxes in italics, while the bold capital labels on components refer to the component name. The muxes and demuxes to provide configurable routing have not yet been added.

components have been created, we can use simulated annealing to order them along the one-dimensional datapath.

For the initial temperature calculation, number of moves per temperature, and cooling schedule, we used the guidelines presented for VPR [Betz97]. The cost metric used in our implementation of the simulated annealing algorithm attempts to create shorter wires in order to minimize the area and delay of the final placement. Since the computation structures are fixed by the netlists at this point, area is directly related to the number of wires passing through a given cross-section of our architecture. Delay is dependent on the length of the wires, and long wires will tend to cause larger cross-sections. Therefore, minimizing the cross-section routing width will aid in reducing both area and delay.

In our cost metric, we relate signals, which are a desired interconnection within a given netlist, to wires, which are physical routing resources fabricated into the array. As stated previously, we have not yet generated a routing structure. We instead use the signals of the netlists to approximate the routing using the following method. First we determine how many signals from each netlist pass through a given cross-section. For our purposes, we consider each physical element location to be a cross-section to examine. Then we take the maximum width across all netlists at each of these cross-sections as an estimate of the routing width required at that location. In order to increasingly discourage wide overall cross-sections, we square the total widths at each component location. To get the final cost value, we add up the squared overall cross-section widths from each cross-section location. This allows us to strongly penalize very wide areas, and lightly penalize more narrow areas.

At the completion of the placement phase, we have obtained an ordering of the physical components along the one-dimensional datapath, as well as a binding of each of the instances of the various netlists to those physical components. The next step is to then generate the routing structure needed to connect the physical components in order to best accommodate the interconnection requirements.

Routing

A custom routing structure, like a custom computational structure, will increase the efficiency for a reconfigurable structure used for a given class of algorithms. By bounding the computation domain to a set of applications, we decrease the amount of extraneous hardware and routing, leading to a more area and delay efficient architecture. The step of creating the custom routing structure is thus essential to our custom architecture generator.

The routing structure of the custom generated architecture will depend on the results obtained in the placement phase. At this point, the physical locations of the components will be fixed, as well as the bindings of the netlist instances to those components. From the netlist information, we have the list of signals, with their respective source and sink instances (the signal's ports). These instances have been bound in the placement stage, so we know the physical location of the ports of the signals. We must create wires and connections in order to allow each netlist to execute individually on our custom hardware.

This also may involve generating multiplexers on the inputs and demultiplexers on the outputs of components to accommodate the different requirements of the various netlists. For example, if netlist A needs the output of the adder to route to a register, but netlist B sends the adder's output to a multiplier, then a demultiplexer is instantiated on the output of the adder to allow for the signals to be directed properly for each netlist. Additionally, if netlist A receives an adder input from a register, while netlist B receives the same input from another adder, a multiplexer is instantiated to choose between these two sources based on which netlist is currently active in the architecture. Figure 4 continues the example from the placement section, showing the generated routing structure for the given placement. Because several of the wires here contain more than one signal, it is evident that the routing cross-section width is less in Figure 4 than if each line from Figure 3 was made into a wire.

The object of the routing generation phase is to minimize area by sharing wires between netlists while keeping the number of

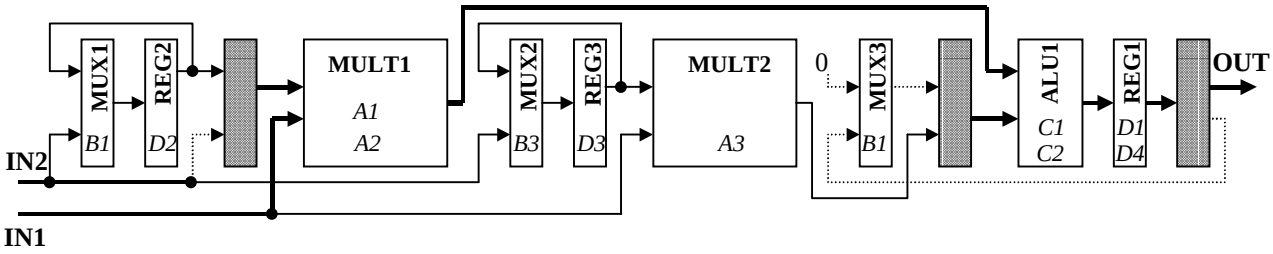

Figure 4: The desired routing architecture for the example of Figure 2 and Figure 3. Each line is a wire. The bold lines indicate a wire that is shared by both netlists. Wires used by only one netlist are shown dotted for Netlist 0 and solid for Netlist 1. The shaded components are routing muxes and demuxes added to allow sharing of components between netlists.

muxes and demuxes required low. We have developed a number of routing algorithms that explore different parts of this design space. Two of the algorithms, greedy and clique partitioning, use heuristics to group similar signals from different netlists into wires. For comparison, the maximum sharing algorithm gives the results for the generated placement if the wires are shared as much as possible (minimizing routing cross-section width). Finally, a lower-bound calculation for area is presented in the results section in order to provide a reference against which we may measure the performance of the other algorithms.

In order to understand the motivations for the algorithms presented below, we must first discuss the routing problem itself. As in the placement problem, creating the routing is two problems combined into one. First, how should the wires be created, and second, which signals should be mapped to which wires. In many current FPGA architectures, wire lengths can be adjusted for each netlist by taking advantage of programmable connections between lengths of wire, potentially forming a single long wire out of a few or many short wires. However, for the current Totem architecture generator we only provide for this type of wire segmentation in the maximum sharing algorithm. While future versions of our application will use segmentation as a means to provide more flexibility within a given area, we begin with non-segmented wires for simplicity.

Unless signals from different netlists share the exact same source and sinks, we 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 minimum sharing algorithm. As stated above, the maximum sharing algorithm uses segmented wires to allow for the maximum amount of wire sharing between netlists, and therefore minimum routing area. The remaining two algorithms, greedy and clique partitioning, use heuristics to determine how the wires should be shared between signals. The heuristics operate by placing signals with a high degree of similarity between ports (source and sinks) together into the same wire. This not only reduces area by sharing the wire, but also reduces the size of the muxes and demuxes off of the shared ports. The exact methods used by each algorithm to determine wire sharing are described below.

Greedy

The concept of correlation between wires or signals is critical to both the greedy and clique partitioning algorithms, and is also used to some extent in the low-area algorithm. The idea is that we wish to merge the different netlists' signals onto physical wires so as to minimize the number and size of multiplexers on the inputs to functional units, and demultiplexers on the outputs.

We use a modified version of the weight equation for a clique-partitioning heuristic [Dorndorf94]. The original equation is: correlation = 2 * <# of identical attributes> - <total # attributes>. Here, items that are completely dissimilar will have a correlation of -<total # attributes>. In our program, an "attribute" is a port, and a "shared attribute" is when two wires share an input or output location. Rather than use the total number of ports in the architecture or the total number of ports for any one wire (which would both create many negative edge weights, penalizing wires with few ports), we instead use the union of the ports of the two signals being correlated. Therefore, the above equation will yield a positive value if more than half of the total number of ports among two signals are shared by those signals.

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

Clique Partitioning

Although the greedy method will place highly correlated signals together into a wire, this is not necessarily the best solution. A higher degree of sharing may be possible, or perhaps the delay would be much better (and the area not

much worse) if a signal was not shared at all. The clique partitioning heuristic uses a more sophisticated algorithm to address these issues.

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. We use a weighted-edge version of clique partitioning [Dorndorf94], as we wish to group highly correlated signals together into wires, where the correlation value between signals is used as the edge weight. The cliques are then partitioned such that the weight of the edges connecting vertices within the same clique is maximized. Signals that cannot occupy the same wire (signals from the same netlist) carry an extremely large negative weight that will prevent them from being assigned to the same clique. Therefore, although signal A may be highly correlated with signal B, and signal B is highly correlated with signal C, they will not all be placed into the same wire (clique) if signal A conflicts with signal C, due to the large negative weight between those vertices.

Given that weighted clique partitioning of a graph with both negative and positive edge weights is NP-Complete, we use an ejection chain heuristic [Dorndorf94] based on tabu search. We start with a random assigning of vertices to cliques (where the number of cliques equals the number of vertices). We allow cliques to be empty, but all vertices must be assigned. The algorithm then uses multiple iterations in which each vertex is moved from its current clique to a different one. This is done by each time selecting a non-tabu vertex and a new clique for that vertex that will produce the maximum overall (not necessarily positive) gain in total weight for the graph. Once a vertex is moved, it is marked tabu until the next iteration.

At the end of the iteration after all vertices have been moved, the list of cumulative solutions after each move is examined, and the one with the highest total weight is chosen. The moves leading to this solution are kept, and the remainder discarded. This solution is then used as the base for the next iteration of moves, and all vertices are marked non-tabu. This loop continues until none of the cumulative solutions in an iteration produces a total weight greater than the base solution for that iteration.

Maximum Sharing

In the current version of Totem, the previous two algorithms do not allow more than one signal from a given netlist to share the same wire through the use of segmented routing. This results in a larger routing area than strictly necessary. In order to provide some sort of measure of what the routing area might be if segmented wires were to be used, the maximum sharing algorithm is used. In this algorithm we actually assign signals to tracks, with a modified version of the standard left-edge algorithm. The signals are sorted based on their leftmost endpoint. Signals are taken in order from the list, leftmost first, and assigned to a track. The track with empty space furthest to the left for the current signal's netlist is selected. If more than one track fits this description, the track with the highest correlation to the current signal is chosen. In this manner, we aim to pack the signals into as few tracks as possible, while still considering some amount of correlation to reduce the number of input multiplexers and output demultiplexers.

This algorithm only represents an attempt at a minimum routing area, not a minimum total area. The difficulty here is that each optional connection in a track requires the instantiation of a bus connector component within the datapath. In the maximum sharing algorithm we do not consider bus connector placement or minimization, and therefore although the routing area may be significantly reduced, the logic area is increased.

Results

In order to form an initial evaluation of our Totem architecture generator, we compare architectures it generates within the DSP domain to RaPiD, a reconfigurable architecture designed specifically for DSP [Ebeling96]. We have obtained a number of compiled RaPiD netlists for testing our architecture generator. These netlists were designed before Totem was written, and hence are not targeted to our generator. The names of these netlist files, along with a short description are listed in

. The table also lists the datapath area required to implement each netlist on the standard RaPiD architecture, as it appears in [Cronquist99a]. It should be noted, however, that these areas are based on the computational requirements of the netlists. These applications were not routed onto the RaPiD architecture, only placed. Several of the applications, such as filter_img and filter_med, will not be able to be routed onto the RaPiD architecture due to heavy routing needs. Therefore, this comparison is somewhat biased in favor of RaPiD (and thus is a conservative estimate of Totem's benefits).

Benchmark	RaPiD area		Description
	# Cells	Area	
filter_img	21	644.07	Image Filter
filter_med	15	460.05	Median Filter
firsm	9	276.03	FIR Filter
firsymeven	17	521.39	FIR Filter
matmult	6	184.02	Matrix Multiplier
sort2	9	276.03	Sorter

Table 1: The RaPiD netlists used for our benchmarking. The number of RaPiD cells required to implement each of these netlists on a traditional RaPiD architecture is given, followed by the resulting area in $M\lambda^2$.

Benchmark Netlists	RaPiD area		Totem Method	Totem Area			Factor Improved	Lower Bound Area	Factor Off Bound
	# Cells	Area		Logic Area	Route Area	Total Area			
filter_img filter_med	21	644.07	Clique	243.85	58.77	302.62	2.13	193.77	1.56
			Greedy	238.75	57.34	296.09	2.18		1.53
			Max Share	583.72	1.38	585.10	1.10		3.02
filter_img filter_med firsymeven	21	644.07	Clique	322.04	84.90	406.94	1.58	255.04	1.60
			Greedy	315.38	81.98	397.36	1.62		1.56
			Max Share	699.10	1.39	700.49	0.92		2.75
filter_img firsm	21	644.07	Clique	250.40	55.19	305.59	2.11	198.93	1.54
			Greedy	248.64	54.62	303.26	2.12		1.52
			Max Share	576.21	2.23	578.44	1.11		2.91
filter_med firsm	15	460.05	Clique	158.19	25.34	183.53	2.51	115.81	1.58
			Greedy	156.80	24.88	181.68	2.53		1.57
			Max Share	380.77	1.41	382.18	1.20		3.30
firsm firsymeven	17	521.39	Clique	168.79	20.97	189.76	2.75	145.88	1.30
			Greedy	167.58	20.72	188.30	2.77		1.29
			Max Share	385.64	1.26	386.90	1.35		2.65
firsm firsymeven matmult	17	521.39	Clique	216.09	35.15	251.24	2.08	179.01	1.40
			Greedy	214.54	34.48	249.02	2.09		1.39
			Max Share	473.39	2.01	475.40	1.10		2.66
firsm matmult sort2	9	276.03	Clique	150.50	21.16	171.66	1.61	110.92	1.55
			Greedy	147.10	20.48	167.58	1.65		1.51
			Max Share	346.13	0.19	346.32	0.80		3.12
ALL	21	644.07	Clique	357.10	99.35	456.45	1.41	255.04	1.79
			Greedy	348.05	93.59	441.64	1.46		1.73
			Max Share	744.01	3.85	747.86	0.86		2.93

Table 2: The results of executing different sets of RaPiD netlists on Totem, along with comparison RaPiD areas and calculated lower bound areas. All areas listed are in terms of $M\lambda^2$.

Given that each RaPiD cell is identical, a combination of RaPiD netlists can be implemented on the maximum number of cells required by any one of those netlists. This is the area calculation used to compare to the results of our generated architectures.

The area of a generated architecture can be considered in terms of logic area and routing area. Logic area includes all computational components, as well as any muxes or demuxes that we may have generated for netlist sharing. For the maximum sharing algorithm, logic area also includes the area required for any bus connectors used within the generated tracks. In all generated architectures, the size of the generated muxes and demuxes, as well as the size of any muxes that the netlist compiler may have instantiated, are reduced to the smallest number of inputs or outputs required to route the needed signals from/to the appropriate location. In other words, if there are 5 signals entering a multiplexer, but 2 of them share the same wire, then the multiplexer has 3 <shouldn't this be 4?> physical inputs, and is sized accordingly. The equations for the mux and demux sizes were obtained through the analysis of the layouts of the RaPiD multiplexer and driver objects. In the traditional RaPiD architecture, all multiplexers have 14 inputs, frequently resulting in needlessly large multiplexers. The routing area of the Totem architectures is somewhat more complex to compute. The RaPiD architecture has 14 buses that run over the computational components. Therefore, we assume that any wire cross-section width up to 14 does not add any height to the layout (it is included in the logic area). However, some areas of the circuit may contain a larger width, and this must be accounted for. In order to compute the additional area beyond the logic area required for the routing, we examine the wire cross-section at each physical component. If this cross section is larger than 14, then the additional routing is translated to the height in lambda required for the extra buses. This height is multiplied by the width in lambda of the corresponding computational unit. The total routing area is obtained by summing this value at each component location.

The different benchmark combinations, along with RaPiD area requirements and Totem area requirements, are listed in Table 2. The results of the three different routing methods are given for each benchmark combination in terms of logic area, routing area, and total area.

For a lower bound on the area requirements of any generated circuit for the input netlists, we consider only the minimum number of circuit elements required to implement all of the

desired circuits. We assume that no additional multiplexing is required on the inputs or outputs to a component due to the presence of multiple netlists. We also assume that all routing will fit within the 14 tracks that are routed over the computational components. In most situations this represents an unachievable lower bound. This lower bound is also given in Table 2.

There are a number of interesting conclusions we can draw from Table 2. First, in both the clique and greedy algorithms, we obtain some area improvement for all netlist combinations, sometimes above a factor of 2. In light of the primitive nature of our first Totem implementation, these results are quite promising. By comparing the routing areas given by greedy and clique to the routing area of maximum sharing, we can infer that the clique and greedy algorithms could be improved a great deal by the introduction of segmented routing. This feature is currently only present in the maximum sharing algorithm, which has led directly to its lower routing area.

Second, we see that sharing wires as much as possible (maximum sharing) may yield the lowest routing area, but does not give the lowest overall area. In fact, for each of our benchmark combinations, maximum sharing resulted in the worst total area of the three algorithms tested. This result is unsurprising, as by requiring maximum sharing we are introducing a large number of bus connectors to form segmented routing. Therefore, when we are adding the ability to create segmented routing to the Totem generator, it will be important to weigh the reduction of wire area against the increase of bus connector area in order to obtain a lower overall area.

Third, the architecture generator has the best performance when the netlists used are highly similar. The two combinations using only FIR filters and the matrix multiplier most closely approach the lower bound area. Intuitively, this makes sense, as similar netlists will use many of the same computational units in much the same order. The placement and routing phases take advantage of this similarity. Note that when all of the netlists are combined to generate one architecture, we have the poorest performance (though still better than the traditional RaPiD architecture for greedy and clique).

Conclusions

FPGAs provide effective solutions for many coarse-grained applications, such as digital signal processing, encryption, scientific data processing, and others. However, commercial FPGAs are fine-grained, and miss many optimization opportunities. Coarse-grained reconfigurable architectures have been designed to improve efficiency for these types of applications, but they still target a broad spectrum of coarse-grained computations. By creating reconfigurable architectures specialized for the application domain(s) being used, we can reap the benefits of custom computational and routing resources, similar to an ASIC, while still leveraging the assets of reconfigurable computing. These custom reconfigurable architectures can then be embedded into an SOC environment, yielding a high-performance computing solution.

Because it is unreasonable to expect a custom FPGA layout for each set of applications needed we have begun work on Totem, a custom reconfigurable architecture generator. This architecture generator will provide the ability to generate the specialized reconfigurable hardware in a fraction of the time required to create an architecture by hand.

We have shown area comparisons between the circuit areas of the architectures generated by the first Totem implementation and the traditional RaPiD architecture areas required for the same combinations of netlists. All but one of our generating algorithms provided a measure of circuit area reduction, even with the initial simplified implementations described in this paper. More than half of the test showed more than a factor of 2 improvement for those algorithms. The results also hint at guidelines for future algorithm development, such as the reduction of segment points in addition to wire cross-section width for segmented routing.

Finally, the improvement of the custom architecture area over the traditional RaPiD area requirements, coupled with the fact that the custom architectures are on average only 1.5 times larger than the absolute lower bound, indicates that custom FPGA architecture generation has a great deal of potential.

Future Work

Currently, we target only area reduction in the creation of our custom reconfigurable architectures. We plan to expand our algorithms to also provide delay optimization. This will allow users of Totem to generate an architecture that will be customized not only for their computation and area needs, but that also meets their delay constraints.

The Totem Custom Reconfigurable Array Generator has a great deal of expansion potential. Because this is the first version, we have made simplifications to the problem in order to obtain our initial results. We can therefore increase the power of this generator in a large number of ways.

First, the cost function for the simulated annealing, while effective, can be further expanded. A more powerful cost function would encourage mappings using the same component type that share a source or destination to be bound to the same resource. This has the benefit of increasing correlations between wires prior to the actual routing structure creation, decreasing the size and/or amount of muxes generated.

Second, at the moment we only create the minimum number of computational units required by the union of the input netlists. Future versions of the Totem generator should provide the flexibility to include resources in excess of the

minimum. This could potentially improve delay values by shortening wires and reducing muxing. The routing method we have initially taken is also on the simplistic side. We provide only point-to-point connections. Future work will entail the creation of segmented routing structures. This will greatly reduce routing area compared to our current implementation, where wires cannot be split.

Additionally, we would like to provide some ability for new netlists with similar characteristics to also be able to be executed on our custom architectures. Currently, the generated architecture is optimized only for the input application netlists, and would likely not perform as well if additional netlists were used post-fabrication. Future versions of the generator will allow users to control where their custom architecture will lie along the spectrum between ASIC and FPGA in order to provide for post-fabrication flexibility of the design.

Finally, to bring our Totem generator to a finished, commercially viable state, we are also working on providing automatic VLSI layout creation of our generated architecture descriptions. Using the output of the architecture generator, this will provide a layout ready for detailed simulation and fabrication. We are also creating automatic methods for generating placement and routing tools for these custom-generated FPGA architectures, enabling Totem to be a complete start-to-finish tool for custom array generation.

Acknowledgments

This research was supported in part by Motorola, Inc., and DARPA. Katherine Compton was supported by an NSF Fellowship. Scott Hauck was supported by an NSF CAREER award.

Thanks to the RaPiD team at the CS department of the University of Washington, especially Chris Fisher and Mike Scott, for fielding questions on the architecture and the compiler. Thanks also to University of Washington EE graduate students Akshay Sharma for the measure of RaPiD cells required to implement each netlist, and Shawn Phillips for help with RaPiD layout structures.

References

[Abnous96] A. Abnous, J. Rabaey, "Ultra-Low-Power Domain-Specific Multimedia Processors", *Proceedings of the IEEE VLSI Signal Processing Workshop*, October 1996.

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

[Compton00] K. Compton, S. Hauck, "Configurable Computing: A Survey of Systems and Software" *submitted to ACM Computing Surveys*, 2000.

[Cronquist99a] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999.

[Cronquist99b] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Presentation at Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999.

[Dorndorf94] U. Dorndorf, E. Pesch, "Fast Clustering Algorithms", *ORSA Journal on Computing*, Vol. 6, No. 2, pp. 141-152, 1994.

[Ebeling96] C. Ebeling, D. C. Cronquist, P. Franklin, "RaPiD – Reconfigurable Pipelined Datapath", *6th Annual Workshop on Field-Programmable Logic and Applications*, 1996.

[Ebeling98] C. Ebeling, "Rapid: A Configurable Architecture for Compute-Intensive Applications", *http://www.cs.washington.edu/research/projects/lis/www/rapid/overview*, 1998.

[Goldstein99] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer, "PipeRench: a Coprocessor for Streaming Multimedia Acceleration", *ISCA*, 1999.

[Scott01] M. Scott, "The RaPiD Cell Structure", *Personal Communications*, 2001.