AUTOMATIC CREATION OF DOMAIN-SPECIFIC RECONFIGURABLE CPLDS FOR SOC

Mark Holland, Scott Hauck

Department of Electrical Engineering
University of Washington
Seattle, WA 98195, USA
mholland@ee.washington.edu, hauck@ee.washington.edu

ABSTRACT

Many System-on-a-Chip devices would benefit from the inclusion of reprogrammable logic on the silicon die, as it can add general computing ability, provide run-time reconfigurability, or even be used for post-fabrication modifications. Also, by catering the logic to the SoC domain, additional area and delay gains can be achieved over current, more general reconfigurable fabrics. This paper presents tools that automate the creation of domain-specific CPLDs for SoC, including an Architecture Generator for finding appropriate architectures and a Layout Generator for creating efficient layouts. By tailoring CPLDs to the domains that they are supporting, we provide results that beat representative fixed architectures by 5.6x to 11.9x on average in terms of area-delay product.

1. INTRODUCTION

Reconfigurable logic fills a useful niche between the flexibility of a processor and the performance provided by custom hardware. This usefulness extends to the System-on-a-Chip (SoC) realm, where reconfigurable logic can provide cost-free upgradeability, coprocessing hardware, and uncommitted testing resources. This flexibility, however, causes area, delay, and power penalties. As such, it would be useful to tailor the reconfigurable logic to a user specified domain in order to reduce the unneeded flexibility, thereby reducing the performance penalties that it suffers. The dilemma then becomes creating domain-specific reconfigurable fabrics in a short enough time that they can be useful to SoC designers.

In the Totem project, we are automating the architecture generation process in order to reduce the amount of effort and time that goes into the process of designing domain-specific reconfigurable logic. This paper presents the creation of domain-specific CPLD architectures, a project termed Totem-CPLD. CPLDs are reconfigurable architectures that typically use PLAs or PALs connected through a central crossbar. Totem-CPLD tailors CPLDs to a specific domain by altering the sizes of the functional units (PLAs) in terms of inputs, product terms, and outputs. This process creates domain-specific CPLD architectures that perform significantly better than "typical" CPLDs.

2. BACKGROUND

Wilton et al. have explored the development of synthesizable programmable logic cores (PLCs) based on PLAs and LUTs [1], [2]. In their process, they develop and deliver an HDL description of the PLC to the SoC designer to be incorporated into the synthesis flow. Their soft cores provide easy integration into existing ASIC flows, but the standard cell hardware implementation of their cores will cause them to suffer performance penalties.

Before Totem-CPLD, we performed work in which we explored the feasibility of making domain-specific reconfigurable PLAs and PALs [3]. We found that we could increase performance by depopulating the AND- and OR-planes in the arrays. Depopulating the arrays in a PLA is very restrictive to future mappings, however, so we chose not to use PLA depopulation in Totem-CPLD.

In order to create CPLD architectures, we will be using a tool called PLAmap, which is currently the best academic technology-mapping algorithm for CPLDs [4]. PLAmap is a performance driven mapping algorithm whose goal is to minimize the delay/depth of the mapped circuit. It is run by providing a PLA size (inputs, product terms, outputs) and a circuit (in BLIF format) to be mapped, and it returns the number of PLAs required and depth of the mapping, along with the mapping itself.

3. APPROACH – TOOL FLOW

The tool flow for Totem-CPLD is as follows. To begin the process, the SoC designer provides us with a domain specification that contains the circuits that need to be supported. These circuits are fed into an Architecture Generator, which finds a CPLD architecture that provides good results for the selected domain, and outputs the architecture description and the area-delay product of the implementation. The architecture description is then sent to a Layout Generator which creates a full VLSI layout of the specified CPLD architecture. The layout is then returned to the designer as "IP" to be incorporated into the SoC device.

3.1. Architecture Generator

The Architecture Generator is responsible for reading in circuits and finding a CPLD architecture that supports the circuits efficiently. Search algorithms are used to make calls to PLAmap, after which the results are analyzed according to area and delay models that we have developed. The algorithms then make a decision to either make further calls to PLAmap, or to exit and use the best CPLD architecture that has been found. PLAmap assumes full connectivity between the PLAs, and the Architecture Generator accommodates this by connecting all the PLAs through a full crossbar.

PLAs are specified by their number of inputs (IN), product terms (PT), and outputs (OUT), so the search space for the Architecture Generator is three-dimensional. Searching the entire 3-D space is not viable, as PLAmap can take on the order of hours for larger circuits, and our ultimate goal is to find a suitable CPLD architecture in a matter of hours or days. Clearly, minimizing the number of PLAmap calls is important to our runtime.

In order to gain some intuition about the search space, we ran five random LGSynth93 circuits through PLAmap and acquired a coarse representation of the 3-D space for each circuit. The first thing that we noticed by looking at these results was that the three PLA variables are related, as can be expected. In general, a ratio of 1 to 2 to .5 for the IN, PT, and OUT variables respectively was found to consistently provide good results, especially in the area of 10-20-5 sized PLAs.

We also observed that the 3-D search space is generally well shaped, meaning that results tend to get better as you approach the optimal point. This observation led us to the concept of breaking the 3-D space into three 1-D spaces, which can be searched sequentially and in much less time. Specifically, our algorithms start by searching for a good input size (while keeping a 1x-2x-.5x IN-PT-OUT relationship), next search for a good output size, and finish by searching for a good product term size.

Architectures are evaluated using the metric of area-delay product. When reported for a domain, the area-delay product consists of the worst-case area implementation in the domain (since the reconfigurable CPLD must be large enough to hold each of the circuits), multiplied by the average delay of the domain. The area model for this calculation is derived from the actual sizing of the VLSI layout components that we created, and the delay model was acquired by performing an hspice static timing analysis of the components.

3.1.1. Search Algorithms

We developed four different Architecture Generation algorithms in order to find good CPLD architectures: Hill Descent, Successive Refinement, Choose N Regions, and Run M Points. All algorithms break up the 3-D search space into 1-D steps by searching for good input, output, and product term sizes, in that order. Additionally, the input step always uses PLAs with a 1x-2x-.5x IN-PT-OUT ratio, while the output and product term steps always alter ONLY the output and product term values respectively. Each variable is explored only in a range that provided reasonable results in preliminary testing: inputs between 4 and 28, product terms between 10 and 90, and outputs between 1 and 25.

3.1.1.1. Hill Descent

The Hill Descent algorithm starts by running PLAmap on architectures with 10-20-5 and 12-24-6 PLAs. Whichever result is better, we continue to take results in that direction (i.e. smaller or larger PLAs), keeping the 1x-2x-.5x ratio intact and performing steps of IN = +/-2. We continue until a local optimum is reached, as determined by the first result that does not improve upon the last result. We then explore the PLAs with IN = +/-1 of the current local optimum. The best result is noted, and the input value is permanently locked at this value, thus ending the input step.

The output optimization step occurs next. The first data point in this step is the local optimum from the input step, and the second data point is acquired by running PLAmap on a PLA with one more output than the current optimum (IN and PT do not change). Again, we descend the hill by altering OUT by +/-1 until the first result that does not improve upon the previous result. At this point we lock the output value and proceed to the product term optimization step. The product term optimization step repeats the process from the previous two steps, varying the PT value by +/-2 until the descent stops. At this point, the PT values +/-1 of the optimum are taken, and the best overall result seen is the output of the algorithm.

The Hill Descent algorithm has no method for avoiding local minima, as any minima will stop the current descent. Therefore it is somewhat difficult for this algorithm to find architectures that vary much in size from the 10-20-5 PLA starting point, but reasonable results are still obtained due to the fact that the 10-20-5 starting point is a relatively good point in the 3-D search space.

3.1.1.2. Successive Refinement

The successive refinement algorithm is intended to slowly disregard the most unsuitable PLA architectures, thereby ultimately deciding upon a good architecture by process of elimination. In the input optimization step (Figure 1), data points are initially taken for PLAs with input counts ranging from 4 (lower bound) to 28 (upper bound) with a step size of 8. So initially, 4-8-2, 12-24-6, 20-40-10, and 28-56-14 PLAs are run (part a in Figure 1). The left and right edges are then examined, regions that do not contain local/global minima are trimmed from consideration (shaded region of part a), and the bounds are adjusted accordingly. The step size is then halved, and the above process is repeated (part b). This occurs until we have performed an exploration with a step size of 1.

For the output optimization step, the IN and PT values are locked at the best result found in the input step. The

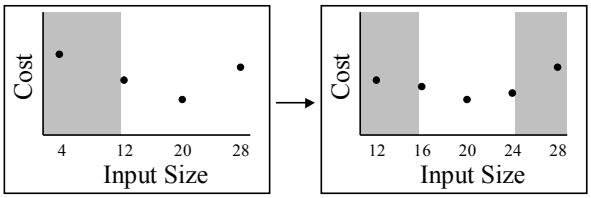

Fig. 1. A step in the input optimization of the Successive Refinement algorithm. At each iteration, shaded regions are trimmed (keeping the data point at their edge) and the step size halved. This continues to a step size of 1.

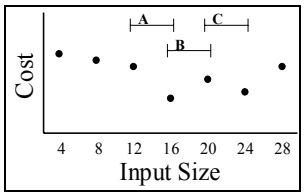

Fig. 2. Choose N Regions algorithm. Region B is the best, because it has the best primary point (along with A) and the best secondary point. Region A is 2nd best, region C is 3rd best.

output values are varied according to the above refinement algorithm, using an initial lower bound of 1, upper bound of 25, and step size of 8. The recursion again continues until the results for a step size of 1 have been taken, at which point we lock the IN and OUT values. The product term optimization step next repeats this process for PT values between 2 and 90 and a step size of 8, after which the best result is returned as the best architecture found.

The Successive Refinement algorithm does not trim sub-optimal regions from the middle, and can therefore require more PLAmap runs than is necessary. However, several local optima are explored at maximum granularity, providing a good survey of the areas around the minima.

3.1.1.3. Choose N Regions

The Choose N Regions algorithm makes a wide sweep of each 1-D space, and then uses the results to choose N regions to explore at a finer granularity. A region consists of the space between two data points.

Like the Successive Refinement algorithm, the input optimization step of the Choose N Regions algorithm is initiated by taking data points for PLAs with inputs ranging from 4 to 28, but now with a step size of 4. N regions are then chosen for further exploration (N=2 was experimentally found to be a good value). A region consists of a data point on the left side, a data point on the right side, and the unexplored space between them. The N best regions are the N regions with the best primary result, using the secondary result to break ties (see Figure 2). These N regions are retained, the step size is halved, and we iterate, repeating until we've explored with a step size of 1.

For the output optimization step, we lock the input and product term values from the result found in the input step. The output value ranges from 1 to 25, with a step size of 4, and the process is repeated. For the product term step, the input and output values from the best result are locked, and the PT values are ranged from 2 to 90 with a step size of 8. After the product term step has completed its step size of 1, the best overall result is returned.

The Choose N Regions algorithm has the advantage of retaining multiple regions of consideration for N>2, and for N=2 it fully explores to the left and right of the best available point. It also can disregard old minima that get replaced by new, better results.

3.1.1.4. Run M Points

The Run M Points algorithm is very similar to the Run N Regions algorithm. Each step is set up in the exact same manner (limits and step sizes), but the best regions are explored differently. Instead of choosing the best N regions and exploring deeper, the Run M Points algorithm always chooses the best data point and explores around it. It does this until it has explored exactly M points for a 1-D step, and then it proceeds to the next step.

While the Choose N Regions algorithm explores N possible optima in parallel (akin to breadth first search), the Run M Points algorithm can be seen as exploring the optima one at a time (depth first). It will explore the best optimum until it runs out of granularity, then will turn to the second best optimum, and so on. In this way it also considers multiple possible optima, as determined by the value chosen for M. Experiments showed that a value of M=15 works well for this algorithm.

3.1.2. Algorithm Add-Ons

The above algorithms comprise the bulk of the Architecture Generator, but some additional routines have been deemed necessary in order to obtain more robust results.

3.1.2.1. Radial Search

The 3-D search space for this problem is relatively well shaped, but there are many local optima that might prevent the above algorithms from finding the global optimum. One way to look outside of these local optima is to search the 3-D space within some radius of the current optimum. So for a radius R search around an X-Y-Z architecture, we would vary IN from X-R to X+R, PT from Y-R to Y+R, and OUT from Z-R to Z+R, testing all architectures in this 3-D subspace. We chose to run radial searches of R = 3 at the end of each basic algorithm in order to look for results that our algorithms missed. Because of large run times, however, radial searches are more for algorithm evaluation than for use in a production system.

3.1.2.2. Algorithm Iteration

The Architecture Generator algorithms all assume that the PLAs should be in a 1x-2x-.5x relationship in terms of inputs, product terms, and outputs. This is just a rough guideline, however, and is very rarely the optimal ratio for a given domain. Thus, an interesting idea is to run the basic algorithms and then look at the resulting PLA to obtain a

new IN-PT-OUT relationship. A second iteration of the algorithm can be run with this new IN-PT-OUT relationship, exploring the 3-D search space using a relationship that the domain has already been shown to prefer. For example, if the first iteration chose a 10-30-8 architecture, then the IN-PT-OUT relationship for the next iteration would be 1x-3x-.8x. A second iteration has been carried out for all of the algorithms on each domain.

3.1.2.3. Small PLA Inflexibility

The initial step of each algorithm locks the input value at a value that it deems to be appropriate by testing a wide range of PLA sizes. During the course of algorithm development, we found that domains that migrate to small input values during the input step (i.e. a 4-8-2 PLA) are left with very little flexibility for the corresponding output and product term steps. The PLAs become strictly input limited, and very few ranges of outputs or product terms will result in reasonable results. When this occurs, the final result of the algorithm tends to be very poor.

To alleviate this, we have added a modification to all of the algorithms. Now, if the input step chooses a PLA with 4 or fewer inputs, the output step will be run both with the PLA found in the input step (4-8-2 or smaller) and with a 10-20-5 PLA. Both of these branches are propagated to the product term step, and the best overall result of the two branches is taken. We found that this process alleviated the problem of being trapped in small PLA sizes, and it provided better results in all applicable cases.

3.2. Layout Generator

The Layout Generator takes the architectures description from the Architecture Generator and turns it into a full VLSI layout. It does this by tiling pre-made, highly optimized layout cells into a full CPLD layout. The Layout Generator runs in Cadence's layoutPlus environment, and uses a SKILL routine that was written by Shawn Phillips [5]. The layouts are designed in the TSMC .18µ process.

Figure 3 displays a small CPLD that was created using the Layout Generator. For clarity's sake, the encoding logic required for programming the RAM bits is not shown, but would appear along the left and bottom of the laid out CPLD. Pre-made cells exist for every part of the PLA and crossbar, as well as the logic used for programming the RAM cells. The Layout Generator simply puts together the pre-made layout pieces as specified by the architecture description that the Architecture Generator provides. The PLAs are implemented in pseudo-nMOS in order to provide a compact layout at the cost of power dissipation.

4. METHODOLOGY

PLAmap restricts us to the use of 2-bounded BLIF format circuits. Many BLIF circuits were obtained from the LGSynth93 benchmark suite, but other circuits were obtained in HDL formats. The HDLs were loaded into Altera's Quartus 2 program, which (thanks to a mod provided by Altera) is able to dump the designs into a BLIF format. SIS was then used to 2-bound the BLIFs.

Table 1. The domains used in our work.

Domain	Circuits	Inputs	Outputs	Gates
Combinational	21	5-178	1-123	8-2350
Sequential	13	4-35	1-23	77-552
Floating Point	12	22-67	22-57	24-9895
Arithmetic	10	28-34	16-33	302-4392
Encryption	6	261-452	132-387	4876-23637

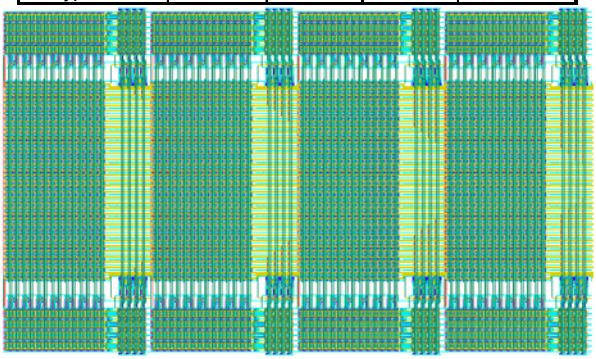

Fig. 3. A small CPLD with eight 7-8-4 PLAs (top and bottom) and a crossbar.

We created five domains of circuits for our major testing. The combinational and sequential domains consist of files gathered from LGSynth93, and are simply grouped for their combinational or sequential characteristics. The remaining three domains consist of floating point, arithmetic, and encryption files respectively. These were all HDL files, accumulated from OpenCores.org, Altera software developers, Quartus 2 megafunctions, and floating point libraries.

The floating point domain consists floating point multipliers, adders, and dividers, as well as an LNS divider, an LNS multiplier, LNS and floating point square root calculators, and a floating point to fixed-point format converter. These files were all obtained from [6] and [7].

The arithmetic domain consists of different multiplier and divider implementations, as well as a square root calculator and an adder/subtractor. The encryption domain consists of the Cast, Crypton05, Magenta, Mars, Rijndael, and Twofish encryption algorithms (all without memories), competitors from the advanced encryption standard competition [8]. Table 1 shows the number of circuits and ranges of inputs, outputs, and gates for each domain.

The domain-specific CPLD architectures are compared to results obtained by implementing the domains in fixed CPLD architectures. We have chosen three different fixed architectures to which to compare our results, all of which will use a full crossbar to connect the PLA units in order to conform to our area and delay models.

A 1991 analysis of PLA sizing in reprogrammable architectures by Kouloheris and El Gamal [9] showed that CPLDs should have PLAs with 8-10 inputs, 12-13 product terms, and 3-4 outputs. To model this, the first architecture to which we will compare uses 10-12-4 PLAs.

Secondly, our own initial analysis using several LGSynth93 circuits showed that 10-20-5 PLAs tended to show good performance. We will use this as our second fixed architecture. Third, we will compare against a Xilinx CoolRunner-like architecture. The CoolRunner uses 36-48-16 PLAs, so we will compare our domain-specific results to a fixed CPLD architecture that uses 36-48-16 PLAs.

Note that we are NOT making a direct comparison to Xilinx's CPLDs or any other existing CPLD architecture. By implementing everything using our own physical layouts, we intend to remove the designer from the cost equation and simply show the advantages obtained by making domain-specific architectures.

5. RESULTS

The Choose N Regions and Run M Points algorithms both have a user-supplied variable. In the Choose N Regions algorithm we must choose how many regions get explored each iteration, while in the Run M Points algorithm we need to determine how many overall PLAmap runs get executed in each of the three steps. By running the algorithms on simple "dummy" domains compiled from LGSynth93, we found that no significant gains are achieved above values of N=2 and M=15 for the Choose N Regions and Run M Points algorithms respectively. As such, N=2 and M=15 will be used for all results.

Next, we took our five main domains and ran each of our algorithms on each domain. Additionally, we mapped each domain to the fixed architectures described above. These results are shown in Table 2. All area*delay results are normalized to the values obtained for the Successive Refinement algorithm, and the columns labeled "Runs" show how many architectures each algorithm tested for each domain. The bottom row shows the geometric mean for area*delay, and the average for runs.

From Table 2 it is apparent that domain-specific CPLD architectures are a win over fixed architectures. For each of the five domains that we considered, the algorithms that we developed always came up with a better CPLD architecture than any of the fixed architectures. On average, the fixed architectures perform 5.1x to 10.9x worse than the domain-specific architectures found by the Successive Refinement algorithm in terms of area*delay.

The Successive Refinement, Choose N Regions, and Run M Points algorithms tend to choose the same architectures. The simple Hill Descent algorithm, however, only matched these results for one of the five domains. With respect to runtime, the Hill Descent algorithm took 3.5x to 5.1x fewer runs than the other algorithms.

The Successive Refinement, Choose N Regions, and Run M Points algorithms all chose 4-8-2 PLAs for the floating point and arithmetic domains in the first step, causing them to be stuck in small PLA architectures. The "Small PLA Inflexibility" algorithm add-on was applied to these instances to remove them from their sub optimal areas. This caused a slight increase in the number of runs that were needed for these algorithms, most notably in the Choose N Regions and Run M Points results.

Each base algorithm was also run with the second iteration and radial search add-ons described above. Table 3 displays the best architectures found by the base algorithms compared to the best results found using these add-ons. If multiple algorithms found the same result, the algorithm that used the fewest runs is reported.

As Table 3 shows, running a second iteration of the algorithms was able to improve the area-delay product by up to .11x, with a mean area*delay gain of .04x and a mean runtime cost of 2.0x. The R=3 radial search add-on was able to reduce the area-delay product by up to .18x, with a .11x mean improvement. The runtime cost for the radius=3 add-on is about 8.5x when compared to the base algorithms.

Table 3 shows that running a second iteration can be as effective as running a radial search, and it requires much less time. Also note that our base algorithms are performing reasonably well, as in all cases they are within .18x of the best results we can easily find.

Table 3 shows that multiple algorithms provide high quality results. Considering the 2nd iteration column, the Run M Points algorithm actually matches the Choose N Regions algorithm for all but one of the data points (the other is only .7% worse) with only a few more runs required for each domain. We have chosen the Run M Points algorithm with a 2nd iteration as our best algorithm.

5.1. Benefits of Domain-Specific Devices

Table 2 shows that our base algorithms find domain-specific architectures that outperform representative fixed architectures by 5.1x to 10.9x. Using our best algorithm, this rises to 5.6x to 11.9x. These are not necessarily the best fixed architectures for our set of domains, though. In fact, we have found the architectures that each domain prefers (in Table 3), so it makes sense that these architectures might work well as fixed architectures. Table 4 shows the area-delay performance of each domain mapped to the architectures found using our best algorithm. Results are normalized to the domain-specific architecture results. The new fixed architectures still perform 1.8x to 2.5x worse than the domain-specific architectures, even though we have hand selected them to go well with our domains. This shows that even if you pick the best possible fixed architecture, there is a bound as to how close you can come to domain-specific results – in this case, domain-specific beats fixed by at least 1.8x.

Table 2. Architecture results for domain-specific and fixed architectures. Results are normalized to the Choose N Regions algorithm. Geometric mean is used for area-delay results.

Domain	Algorithms												Fixed Architectures		
	Hill Descent			Succ. Refinement			Choose N Regions			Run M Points			10-12-4	10-20-5	36-48-16
	Arch	A*D	Runs	Arch	A*D	Runs	Arch	A*D	Runs	Arch	A*D	Runs	A*D	A*D	A*D
Combinational	12-25-4	2.16	13	12-71-4	1.00	90	12-71-4	1.00	40	12-70-4	1.01	52	9.52	4.13	9.46
Sequential	14-27-5	1.18	14	14-38-5	1.00	78	14-38-5	1.00	39	14-38-5	1.00	50	2.65	2.34	2.26
Floating Point	9-26-4	4.50	15	10-24-2	1.00	82	10-24-2	1.00	67	10-24-2	1.00	85	13.02	8.04	28.21
Arithmetic	10-22-2	1.00	14	10-22-2	1.00	76	10-22-2	1.00	67	10-22-2	1.00	85	46.71	18.73	46.49
Encryption	10-20-2	1.47	13	16-67-4	1.00	38	10-25-2	1.47	39	10-25-2	1.47	51	3.10	2.33	5.38
Geo. Mean		**1.76**	**13.8**		**1.00**	**69.8**		**1.08**	**48.7**		**1.08**	**62.6**	**8.62**	**5.08**	**10.86**

Table 3. Best base algorithm results compared to best results after a 2nd iteration and radius=3 search. (HD=Hill Descent, SR=Successive Refinement, CN=Choose N Regions, RM=Run M Points).

Domain	Best Base Algorithm				Best with 2nd Iteration				Best with Rad.=3 Search			
	Alg	Arch	A*D	Runs	Alg	Arch	A*D	Runs	Alg	Arch	A*D	Runs
Combinational	CN	12-71-4	1.00	40	CN	12-71-4	1.00	60	CN	9-72-4	0.85	377
Sequential	CN	14-38-5	1.00	39	CN	14-38-4	0.99	77	CN	14-37-6	0.96	377
Floating Point	CN	10-24-2	1.00	67	CN	8-18-2	0.89	105	CN	8-21-2	0.90	307
Arithmetic	HD	10-22-2	1.00	14	CN	10-22-2	1.00	87	HD	7-20-2	0.82	250
Encryption	SR	16-67-4	1.00	38	SR	15-68-4	0.92	75	SR	15-68-4	0.92	375
G Mean / Avg			**1.00**	**39.6**			**0.96**	**80.8**			**0.89**	**337.2**

Table 4. Results of running each domain on the best domain-specific architectures found.

Domain	Best	12-70-4	14-38-4	8-18-2	10-22-2	8-32-2
Combinational	1.00	1.00	2.74	3.61	4.38	2.83
Sequential	1.00	2.22	1.00	3.61	2.85	4.13
Floating Point	1.00	3.90	4.24	1.00	1.14	1.02
Arithmetic	1.00	6.21	6.80	1.82	1.00	1.80
Encryption	1.00	1.20	1.28	1.01	1.39	1.00
Geo. Mean	**1.00**	**2.30**	**2.52**	**1.89**	**1.82**	**1.85**

6. CONCLUSION

We have presented a tool flow for creating domain-specific CPLDs for SoC, including an Architecture Generator which finds domain-specific CPLD architectures by using any of four search algorithms. Analysis of the different algorithms and possible add-ons displayed that the Run M Points algorithm with a second iteration is our best algorithm.

Using this algorithm, we created domain-specific architectures that outperform representative fixed architectures by 5.6x to 11.9x in area*delay. Even choosing the best fixed architectures available, our domains specific architectures were still 1.8x to 2.5x better.

This paper also presented a Layout Generator which takes pre-made layout units and tiles them to make full VLSI CPLD layouts in the TSMC .18µ process.

7. ACKNOWLEDGEMENTS

Mike Hutton and Swati Pathak at Altera were essential to this work, providing the BLIF dumper for Quartus 2 and many useful circuits. Ken Eguro provided the encryption circuits, and Steve Wilton provided a vital VQM to BLIF converter. Deming Chen provided assistance with PLAmap.

Mark Holland was supported by an NSF Fellowship, and Scott Hauck by a Sloan Fellowship. This research was supported by a grant from NSF.

8. REFERENCES

[1] A. Yan, S. Wilton, "Product Term Embedded Synthesizable Logic Cores", IEEE International Conference on Field-Programmable Technology, 2003, pp. 162-169.

[2] N. Kafafi, K. Bozman, S.J.E. Wilton, "Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores", ACM/SIGDA 11th International Symposium on Field-Programmable Gate Arrays, 2003, pp. 3-11.

[3] M. Holland, S. Hauck, "Automatic Creation of Reconfigurable PALs/PLAs for SoC", 14th International Conference on Field-Programmable Logic and Applications, 2004, pp. 536-545.

[4] D. Chen, J. Cong, M. Ercegovac, Z. Huang, "Performance-Driven Mapping for CPLD Architectures", ACM/SIGDA 9th International Symposium on Field-Programmable Gate Arrays, 2001, pp. 39-47.

[5] S. Phillips, "Automating Layout of Reconfigurable Subsystems for Systems-on-a-Chip", PhD Thesis, University of Washington, Dept. of EE, 2004.

[6] ENS Lyon. A VHDL Library of Parametrisable Floating-Point and LNS Operators for FPGA. September 4, 2003. <http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/>

[7] M. Leeser. Variable Precision Floating Point Modules. <http://www.ece.neu.edu/groups/rpl/projects/floatingpoint/>

[8] U.S. Department of Commerce. National Institute of Standards and Technology. FIPS PUB 197, Advanced Encryption Standard (AES), November 2001.

[9] J. Kouloheris, A. El Gamal, "FPGA Performance vs. Cell Granularity", IEEE Proceedings of the Custom Integrated Circuits Conference, 1991, pp. 6.2/1-6.2/4.

[10] M. Holland, "Automatic Creation of Product-Term Based Reconfigurable Architectures for System-on-a-Chip", PhD Thesis, University of Washington, Dept. of EE, 2005.