Automatic Creation of Domain-Specific Reconfigurable CPLDs for SoC

Mark Holland, Scott Hauck

***Abstract*—This paper presents tools that automate the creation of domain-specific CPLDs, targeted for SoC. By tailoring full-crossbar based CPLDs to the domains that they support, we provide results that beat fixed reconfigurable architectures by 5.5x to 11.8x on average in terms of area-delay product. We also create sparse-crossbar based CPLD architectures, using a novel switch smoothing algorithm that makes the crossbars amenable to layout. This algorithm reduced the wire jog pitch of our largest layout from 48 to just 3, allowing for a compact VLSI layout. These sparse-crossbar based CPLDs require just .37x the area and .30x the delay of our full-crossbar based CPLDs. We also address the question of how best to add resources to a CPLD in order to support future, unknown circuits, concluding that the best strategy is to add 5% to the crossbar switch density and to provide additional PLAs of the same size found in the base architecture.**

***Index Terms*— Automation, CPLD, Domain-Specific, System-on-a-Chip**

I. Introduction

System-on-a-Chip (SoC) designs are becoming more and more commonplace in the VLSI community, and one question that remains unanswered is how well reconfigurable logic will fit into this new design paradigm. SoC allows the integration of several distinct components on a single piece of silicon, but does so at the cost of increased design complexity. This complexity is usually alleviated through the use of hardware description language (HDL) designs which are synthesized to standard cells, or through the use of pre-made and pre-tested intellectual property (IP) cores.

Reconfigurable logic is composed of hardware resources whose function can be modified through the setting of configuration bits. As such, it merges much of the flexibility provided by general-purpose processors with the performance provided by custom hardware. Reconfigurable logic is useful in the SoC realm because it can provide cost-free upgradability, coprocessing hardware, conformity to varying protocols, or even uncommitted testing resources to an SoC design. Reconfigurable architectures can also be personalized to fit the needs of the SoC, as the “domain” of the chip will be known in advance, and the paradigm of IP reuse makes it particularly easy to incorporate reconfigurable logic into SoC as pre-made IP.

We feel that the integration of reconfigurable logic into SoC designs will depend largely on three factors: the performance, cost, and development time of the reconfigurable architectures. The Totem Project at the University of Washington is our attempt to address all these factors by automating the creation of domain-specific reconfigurable architectures. Automation allows us to provide quick turn-around time, creating domain-specific architectures shrinks the performance gap between our reconfigurable logic and fixed logic, and we will be able to retain the low cost and high-flexibility that reconfigurable devices are renowned for.

This paper deals with the creation of domain-specific CPLD architectures, a project termed Totem-CPLD. CPLDs are relatively small reconfigurable architectures that typically use PLAs or PALs as their functional units, and which connect the units using a single, central interconnect structure. We will be creating domain-specific PLA-based CPLDs by optimizing the sizes of their PLAs in terms of inputs, product terms, and outputs, and by reducing the connectivity of their interconnect structures.

II. Background

Of existing research, the most applicable to this work is “Product-Term Based Synthesizable Embedded Programmable Logic Cores” by A. Yan and S. Wilton [1]. In this paper they explore the development of synthesizable programmable logic cores based on PLAs, which are implemented in standard cells. Their methodology provides easy integration into an existing HDL-based flow, and while they provide some performance gains by tailoring to specific applications, the eventual standard cell implementation negates much of these gains.

In another related work, highly regular reconfigurable “Glacier” PLA (GPLA) structures are proposed [2]. The architecture stacks multiple PLAs in a uni-directional structure using river routing to connect them together, resulting in a structure that benefits from both high circuit regularity and

Manuscript received March 14, 2006. Mark Holland was supported by an NSF Fellowship, and Scott Hauck by a Sloan Fellowship.

M. Holland was with the University of Washington, Seattle, WA 98195 USA. He is now with Annapolis Micro Systems in Annapolis, MD 21401 USA (email mholland@gmail.com).

S. Hauck is with the University of Washington, Seattle WA 98195 USA (email shauck@ee.washington.edu).

predictable area and delay formulation. GPLAs are similar to our work in that they are hard programmable cores that can be integrated into SoC, but their interconnect is overly sparse, and is restricted by the need for directionality.

Finally, as a precursor to Totem-CPLD we performed work in which we explored the feasibility of making domain-specific reconfigurable PLAs and PALs [3]. In that work we wrote an architecture generation tool that mapped domains of circuits to either a PLA or a PAL in such a way that it could remove some of the unneeded programmable connections in the arrays. By doing this intelligently, we were able to remove 60%-70% of the programmable connections in the arrays, which provided delay gains of 15% to 30%. 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 this paper we will first introduce a method for creating domain-specific full-crossbar based CPLD architectures, targeted for SoC. We will then show how performance gains can be achieved by utilizing sparse-crossbars in the CPLD architecture. Last, we will touch on the question of how to add resources to our CPLDs in order to maximize the likelihood that future circuits will be supported by the architectures.

III. Full-Crossbar Based CPLDs

The tool flow for creating domain-specific full-crossbar based CPLDs is shown in Fig. 1. To begin the process, the customer 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 efficiently supports the selected domain, outputting the description of this architecture. The architecture description is then sent to a Layout Generator which creates a full VLSI layout of the specified CPLD architecture. The circuits and the architecture description are also fed into a Place & Route Tool, which is responsible for creating the configuration bitstreams that implement the circuits on the domain-specific architecture.

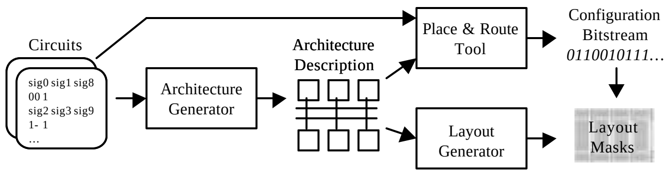

Fig. 1. The High Level Totem-CPLD Tool Flow

A. Architecture Generator

The Architecture Generator is responsible for reading in multiple circuits and finding a CPLD architecture that supports the circuits efficiently. We use a search algorithm to make calls to a tech-mapper named PLAmap, after which the results are analyzed according to area and delay models that we have developed. The algorithm then makes 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.

The Architecture Generator is responsible for finding a PLA size that leads to an efficient CPLD architecture for the given domain. 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. In order to keep the runtime or our tool reasonable, we needed to develop an algorithm that provides good results while only sampling a reasonable amount of this large search space.

Preliminary testing of LGSynth93 [4] circuits provided results that helped guide our algorithm development. We found that a ratio of 1 to 2 to .5 for the IN, PT, and OUT variables respectively provided effective architectures, especially in the area of 10-20-5 sized PLAs. Additionally, we observed that the 3-D search space is generally well shaped, meaning that results tend to get better as you approach more optimal points. This latter 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.

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.

1) *Search Algorithm*

We developed several algorithms in order to find effective CPLD architectures, and the most effective was our "Run M Points" algorithm. This algorithms break up the 3-D search space into 1-D steps by searching for good IN, OUT, and PT sizes, in that order.

The algorithm starts by varying IN from 4 to 28, with a step size of 4, and looks at PLAs with a 1x-2x-.5x IN-PT-OUT ratio. The best data point is noted, and the space around it is explored, maintaining this IN-PT-OUT ratio. This is continued with the best existing point until M points have been explored for this 1-D step.

We next lock the IN and PT values from the best point in the previous 1-D step, and we optimize for outputs. We range the OUT value from 1 to 25, with a step size of 4, and explore this 1-D space by varying only the OUT value. Again, the best point is found, and the space around it explored. This is repeated until M points have been explored in this step.

In the PT step, the IN and OUT values are obtained from the best previous data point and are locked. PT values are then ranged from 2 to 90 with a step size of 8, and the above process is repeated. For the PT step, slightly more than M points are searched in order to allow the 1-D space to be explored to the same granularity as the previous 1-D spaces.

Experimentation with the Run M Points algorithm showed that we should use a value of M=15. Additionally, we found that it was worthwhile to run a second iteration of the above 1-D steps, but using a new ratio in the IN optimization step. The algorithm initially assumes that the PLAs should be in a 1x-2x-.5x relationship in terms of IN, PT, and OUT, but this is just a rough guideline. After the first iteration finds a good data point with a desired IN-PT-OUT ratio, a second iteration of the algorithm is run with this new IN-PT-OUT relationship for the first step, exploring the 3-D search space using a relationship that the domain has already been shown to prefer. For example, if the first iteration (three 1-D steps) chose a 10-30-8 architecture, then the IN-PT-OUT relationship for the next iteration would be 1x-3x-.8x.

Another important note is that many domains we tested migrated to small input values during the input step (i.e. a 4-8-2 PLA). These algorithms are then left with very little flexibility for the corresponding output and product term steps, becoming strictly input limited. To alleviate this, we added a modification to algorithm. If the input step chose a PLA with 4 or fewer inputs, the output step was 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 were propagated to the product term step, and the best overall result of the two branches was taken, effectively alleviating the problem.

B. *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 from Cadence's command line, and uses a SKILL routine that was written by Shawn Phillips [5]. The layouts are designed in the TSMC .18µ process. The PLAs are implemented in pseudo-nMOS in order to provide a compact layout.

C. *Full-Crossbar CPLD Methodology*

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. Table 1 shows the number of circuits and ranges of inputs, outputs, and gates for each domain.

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

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.

The first architecture uses 10-12-4 PLAs, as suggested by a 1991 analysis of PLA sising in reconfigurable architectures [6]. Secondly, our own initial analysis using several LGSynth93 circuits showed that 10-20-5 PLAs tended to show good performance. Third, we will compare against a Xilinx CoolRunner-like architecture. The CoolRunner [7] 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 any 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.

D. *Full-Crossbar CPLD Results*

We ran our algorithm on each of the five domains to acquire domain-specific architectures, and we also mapped each domain to the fixed architectures described above. These results are shown in Table 2. All results are normalized to the values obtained with the Run M Points algorithm. The bottom row shows the geometric mean for area*delay.

TABLE 2
ARCHITECTURE RESULTS FOR DOMAIN-SPECIFIC AND FIXED ARCHITECTURES.

Domain	Domain Specific		Xilinx	El Gamal	Observed
			36-48-16	10-12-4	10-20-5
	Arch	A*D	A*D	A*D	A*D
Sequential	14-38-4	1.00	2.28	2.67	2.35
Arithmetic	10-22-2	1.00	46.49	46.71	18.73
Combinational	12-70-4	1.00	9.40	9.46	4.10
Floating Point	8-18-2	1.00	31.60	14.58	9.00
Encryption	8-32-2	1.00	7.43	4.28	3.22
Geo. Mean		**1.00**	**11.85**	**9.41**	**5.55**

From Table 2 it is apparent that domain-specific CPLD architectures are a win over fixed architectures. The algorithm that we developed always came up with a better CPLD architecture than any of the fixed architectures. On average, the fixed architectures perform 5.5x to 11.8x worse than the domain-specific architectures found by the Run M Points algorithm in terms of area*delay.

To be thorough, we wanted to test our reconfigurable architectures against the best fixed architectures we could find. We have already found the architectures that each domain prefers by using our search algorithms, so it makes sense that these architectures might work well as fixed architectures.

TABLE 3
RESULTS OF RUNNING EACH DOMAIN ON THE DOMAIN-SPECIFIC ARCHITECTURES FOUND BY OUR BEST ALGORITHM

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**

Table 3 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. Notice that the new fixed architectures still perform 1.8x to 2.5x worse than the domain-specific architectures, even though we hand picked them to go well with our domains.

IV. Sparse-Crossbar Based CPLDs

Full crossbar implementations cause CPLDs to scale quite poorly, and area can be drastically improved by utilizing sparse crossbars. When using sparse crossbars, there is no guarantee that a signal from the interconnect can reach any PLA input, so a router must be employed. This section presents a method for creating domain-specific sparse-crossbar based CPLDs for use in SoC devices. We will define a crossbar as the connection matrix that provides connectivity between the general interconnect wires and the input wires of a single PLA. The same high level tool flow from Fig. 1 is still appropriate, although we will need to modify the Architecture Generator and the Place & Route Tool in order to create and map to the sparse-crossbar based CPLDs.

A. *Architecture Generator*

For sparse-crossbar based CPLDs the process of architecture generation has additional steps, as a Crossbar Generator is used to create sparse crossbars of different switch counts, and a Router is employed to determine the smallest sparse crossbar which allows all the circuits in the domain to be routed on the CPLD. This is done in the form of a binary search on the sparse crossbar switch count.

1) *Crossbar Generator*

When creating sparse crossbars, the objective is to maximize the routability of the crossbar for a given switch count, where routability is defined as the likelihood that an arbitrarily chosen subset of inputs can be connected to outputs. While switch placement is deterministic for crossbars that provide full capacity, it is not obvious how the switches should be placed in a sparse crossbar in order to maximize routability. This problem, though, has been effectively addressed by a sparse-crossbar-generation tool created by Lemieux and Lewis [8]. They use simulated annealing and a hamming-distance based cost function to move switches within a crossbar, ultimately acquiring a sparse crossbar with an effective switch distribution. We utilize this sparse-crossbar-generation algorithm in our tool flow, and also add to it in order to make the crossbars amenable to layout.

a) *Switch Smoother*

The sparse crossbars acquired from the algorithm in [8] will have switch distributions that look fairly random. Because of this, there are likely to be regions in the crossbars that have relatively high or low switch densities. The preceding algorithm has ensured that our crossbar is highly routable, but it has not ensured that the switches are spread out in such a manner that they will lead to an efficient layout.

The switches and their corresponding SRAM bits determine the area required by the crossbar, as the input and output wires can simply be brought in on metal layers above these devices. In order to achieve a compact layout, therefore, it is desirable to pack the switches as closely as possible. This is easy for a full crossbar, shown on the left of Fig. 2, because there is a switch at the intersection of every input and output wire. When we use a sparse crossbar, there are fewer switches per input line, and this causes the input lines to be more closely packed. Jogs are now necessary to connect input lines to some of their switches, as shown on the right of Fig. 2. In order to keep the switches packed tightly, we must limit the number of vertical jogs required in our sparse crossbars. Otherwise, the pitch required by the vertical jogs would start to dominate the east-west dimension of the crossbar layout.

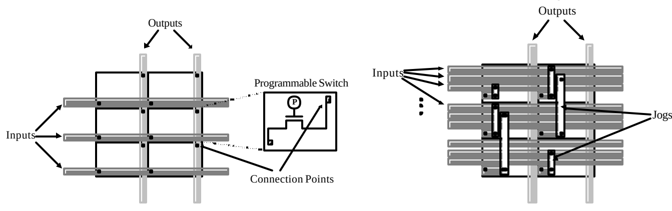

Fig. 2. Layout of full crossbars (left) and sparse crossbars (right)

In a layout with N input wire pitches per vertical switch pitch, we can minimize the number of vertical jogs by requiring that each output line be attached to exactly one switch per every N input lines. This ensures that each input line attaches to a switch that is underneath it. The input lines in our sparse crossbar can be permuted, so we can move them around in order to enforce this property.

The switch smoothing algorithm operates by ordering the crossbar input lines such that each output line is attached to roughly one switch per every N input lines. It does this in a greedy fashion, placing one input line at a time such that this property is enforced. This is achieved by minimizing the cost function shown below as each input line is placed. In this equation for an n input, m output sparse crossbar, x is the current output line, S_x is the number of switches that have been placed in output line x, P is the current input line, and N is the number of input lines per switch pitch in the layout. The cost function ensures that no output line drifts far from having, at any point, exactly one switch per every N input lines that have been placed. This, in turn, assures that horizontal wires in our resulting layout will connect to switches that are relatively close to the horizontal wire, requiring few wire jogs pitches.

$$\sum_{\forall x}(S_x - \frac{P}{N})^2$$

B. Layout Generator

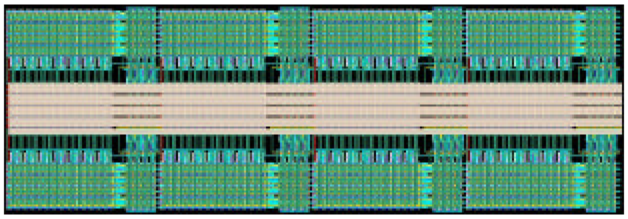

Fig. 3. A sample sparse-crossbar based CPLD layout

The Layout Generator functions the same as for full-crossbar based CPLDs, except that it now creates sparse crossbars instead of full crossbars. Again, all the necessary building blocks have been efficiently laid out, and must be intelligently tiled in order to create a sparse-crossbar based CPLD. Fig. 3 displays a sparse-crossbar based CPLD that was created using this process.

C. Place & Route Tool

Just as with our full-crossbar based CPLDs, the task of PLA placement is unnecessary for our sparse-crossbar based CPLDs due to the fact that all signals in the interconnect span the entire device, and connect to each PLA equally. Our Place & Route Tool must simply compose of PLAmap, which is used to map incoming circuits to CPLDs, and Pathfinder [9], which is used to route the signals on the CPLD architectures.

D. Sparse-Crossbar CPLD Results

We have chosen a layout topology that restricts us to 8 or fewer jog pitches per crossbar output. Table 4 displays the pre-smoothing and post-smoothing jog pitches required of the sparse crossbars that we created for our five domains (the performance of these architectures appears later in this section). As the table shows, four of the five crossbars would have resulted in severe area penalties in the final layouts if the crossbar-smoothing algorithm had not been applied.

TABLE 4
RESULTS OF RUNNING THE SWITCH SMOOTHING ALGORITHM ON CROSSBARS ACQUIRED FOR EACH DOMAIN

Domain	Crossbar			Required Jog Pitches	
	Inputs	Outputs	Switches	Pre-Smoothing	Post-Smoothing
Sequential	155	14	293	6	2
Arithmetic	613	21	1216	12	3
Combinational	735	14	2647	19	2
Floating Point	1574	18	4627	17	3
Encryption	5002	23	32099	48	3

Our goal in using sparse crossbars is to reduce the area and delay of our CPLD architectures. Table 5 displays the area, delay, and area-delay product results obtained by using sparse-crossbar based CPLD architectures, compared to the results for full-crossbar based architectures. Our search algorithm found different PLA sizes for our new CPLD architectures, migrating to other effective architectures that were better able to leverage the use of sparse crossbars. The best sparse-crossbar-based CPLDs are shown to require .37x the area, .30x the delay, and .11x the area-delay product of our best full-crossbar CPLDs.

TABLE 5
FULL AND SPARSE-CROSSBAR BASED CPLD RESULTS

	Full Crossbar				Sparse Crossbar			
Domain	Arch	Area	Delay	A*D	Arch	Area	Delay	A*D
Sequential	14-38-4	1.00	1.00	1.00	14-18-4	0.36	0.48	0.17
Arithmetic	10-22-2	1.00	1.00	1.00	3-16-2	0.31	0.25	0.08
Combinational	12-70-4	1.00	1.00	1.00	14-52-3	0.40	0.39	0.15
Floating Point	8-18-2	1.00	1.00	1.00	18-55-3	0.37	0.21	0.08
Encryption	8-32-2	1.00	1.00	1.00	23-79-4	0.41	0.27	0.11
Geo. Mean		**1.00**	**1.00**	**1.00**		**0.37**	**0.30**	**0.11**

V. ADDING CAPACITY TO CPLDS

Many SoC designers will not know the circuits that they wish to implement in reconfigurable logic on their devices. In these instances, the designer will usually know the domain of the circuits that they will implement in reconfigurable logic, just not the exact specifics of the circuits. In this situation our goals change slightly, as we now wish to create an architecture that not only supports the sample circuits, but which is as likely as possible to support an unknown circuit in the same domain.

To simulate this situation, we took our domains, removed circuits from them, and acquired domain-specific architectures for the reduced domains. We then reintroduced the removed circuits, and added resources in different ways to see which resource strategy picked up these circuits most effectively.

Multiple resource strategies were attempted, including strategies which add switches to our sparse crossbars, and strategies which use larger PLAs or more PLAs. Hybrid strategies which merge these base strategies were also attempted. Our results showed us that the most effective strategy was to add 5% to the number of switches in the crossbars, and to use additional PLAs of the same size that are in the base architecture. More details can be found in [10].

VI. CONCLUSION

In this paper we presented a complete tool flow for creating domain-specific CPLDs for System-on-a-Chip devices. We presented a novel architecture generation algorithm which was able to find CPLD architectures that effectively supported specific application domains. Using this algorithm, we found full-crossbar based CPLD architectures that outperformed representative fixed architectures by 5.5x to 11.8x in terms of area-delay product. We were then able to get another 9x improvement on area-delay by using sparse-crossbars in the CPLDs. Vital to the implementation of sparse-crossbars was a novel switch-smoothing algorithm, which was able to reduce the number of vertical wire jogs in a layout from 48 to 3 in the largest architecture.

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 .18-micron process. We also touched upon the concept of adding capacity to our domain-specific

CPLDs, concluding that it is good to add 5% to the switch density of the sparse crossbars used in the CPLD architectures and PLAs of the base size.

References

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

[2] F. Mo, R. K. Brayton, “River PLAs: A Regular Circuit Structure”, *DAC*, 2002.

[3] M. Holland, S. Hauck, “Automatic Creation of Reconfigurable PALs/PLAs for SoC”, FPL, 2004.

[4] “1993 LGSynth Benchmarks”, <http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.cbl.ncsu.edu/CBL_Docs/lgs93.html> (March 25, 1997).

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

[6] J. Kouloheris, A. El Gamal, “FPGA Performance vs. Cell Granularity”, Proc. Custom Integrated Circuits Conference, 1991.

[7] Xilinx, Inc., *CoolRunner-II CPLD Family: Advance Product Specification*, March 12, 2003.

[8] G. Lemieux and D. Lewis, *Design of Interconnection Networks for Programmable Logic*, Boston, Kluwer Academic Publishers, 2004.

[9] L. McMurchie, C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays*, pp. 111-117, February 1995.

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