Improving Performance and Robustness of Domain-Specific CPLDs

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

Scott Hauck
Department of Electrical Engineering
University of Washington
Seattle, WA 98195, USA
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 tailoring the logic to the SoC domain, additional area and delay gains can be achieved over current, more general reconfigurable fabrics. This paper presents our work on creating efficient CPLD architectures for SoC, including the creation of sparse crossbars, and a novel switch smoothing algorithm which makes the crossbars amenable to layout. For our largest architecture, the switch smoothing algorithm reduced the layout's wire jog pitch from 48 to just 3, allowing for a compact VLSI layout. This has helped pave the way for our sparse-crossbar based CPLDs, which require just .37x the area and .30x the delay of our full-crossbar based CPLDs.

However, regardless of how efficient an architecture we develop, it is useless if it does not have enough resources to support the circuits to be implemented. 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.

Categories and Subject Descriptors
J.6 **[Computer Aided Engineering]:** Computer Aided Design (CAD)

B.7.1 **[Integrated Circuits]:** Types and Design Styles – *VLSI (very large scale integration)*

General Terms
Algorithms, Performance, Design, Reliability

Keywords
CPLD, System-on-a-Chip, Reconfigurable Logic, Computer-Aided Design, Sparse Crossbar

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'06, February 22–24, 2006, Monterey, California, USA.
Copyright 2006 ACM 1-59593-292-5/06/0002...$5.00.

1. INTRODUCTION
As the semiconductor industry continues to follow Moore's Law, a switch in design paradigm is occurring. The former "System-on-a-Board" style, which had discrete components individually fabricated and then integrated together on a board, is becoming obsolete. Due to ever increasing integration levels, distinct VLSI components can now be incorporated onto the same piece of silicon, enabling the creation of "System-on-a-Chip" devices.

Integrating several components in the same piece of silicon has several advantages. The most obvious of these is reduced area, as the move from a board to a single chip is a clear win. The smaller area also leads to lower path delays and less power dissipation, two factors that are important in VLSI designs. Another advantage is that inter-device communication can be richer, as pin limitations are no longer a concern.

Of course, as more resources are put onto a single chip, the actual design of that chip becomes more difficult. In SoC designs, this is often alleviated by using hardware description languages (HDLs) to describe the hardware. Synthesis tools map the HDL designs to gates, and they are ultimately laid out using standard cells.

Standard cells, however, do not perform as well as manually laid out designs. Because of this, a second SoC design paradigm has emerged: intellectual property (IP) reuse. The basic idea of IP reuse is that once a device is carefully designed, tested, and verified, the next user who wishes to use the device won't have to repeat any of those steps. IP Cores are becoming available in a wide variety of flavors, including processors, DSPs, memories, and of particular interest to us, reconfigurable logic cores.

Reconfigurable logic fills a useful niche between the flexibility provided by a processor and the performance provided by custom hardware. This usefulness extends to the SoC realm, where reconfigurable logic can provide upgradability, conformity to different but similar protocols, coprocessing hardware, and testing resources. Additionally, the paradigm of IP reuse makes it easier to incorporate reconfigurable logic into SoC as pre-made IP cores.

Traditional reconfigurable logic needs to provide a high level of flexibility so that it will be useful in a wide range of designs. This flexibility, however, comes at the cost of increased area, delay, and power. 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 area, delay, and power penalties that it suffers. The dilemma then becomes creating these domain specific reconfigurable fabrics in a short enough time that they can be useful to SoC designers.

The Totem project is our attempt to reduce the amount of effort and time that goes into the process of designing domain specific reconfigurable logic. By automating the generation process, we will be able to accept a domain description and quickly return a reconfigurable architecture that targets that domain.

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 (Figure 1). In commercial architectures, the functional units tend to be relatively coarse grained in order to provide shallow mappings, leading to low and predictable delays.

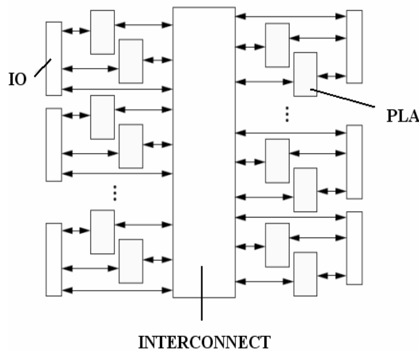

Figure 1. A CPLD with central interconnect

CPLDs have traditionally been used for implementing control logic, state machines, and other seemingly random logic, but they are not limited to these applications: the generality of a CPLD allows it to implement almost any logic of small enough size. This is the same generality, however, which causes performance penalties in terms of area, delay, and power.

In Totem-CPLD we tailor PLA-based CPLDs to a specific application domain, thereby removing some of these performance penalties. Specifically, by altering the sizes of our PLAs in terms of inputs, product terms, and outputs, and by reducing the connectivity of the interconnect structure, CPLD architectures can be created that perform better than "typical" CPLD architectures for a specified domain. This paper presents a method for creating sparse-crossbar based CPLD architectures, including a novel switch smoothing algorithm, which will provide improved performance over the full-crossbar based architectures we previously developed [1]. We also provide an analysis of the CPLD resources that should be added to an architecture in order to support future, unknown circuits that exist in the computational domain that we are targeting.

2. Background

Many papers have been published with respect to CPLD architectures, but very few of them have aimed at creating reconfigurable architectures for SoC. The most applicable of these was "Product-Term Based Synthesizable Embedded Programmable Logic Cores" by A. Yan and S. Wilton [2]. In this paper they explore the development of "soft" or synthesizable programmable logic cores based on PLAs, which they call product term arrays. In their process they acquire the high-level requirements of a design (# of inputs, # of outputs, gate count) and then create a hardware description language (HDL) representation of a programmable core that will satisfy the requirements. This HDL description is then given to the SoC designer so that they can use the same synthesis tools in creating the programmable core that they use to create other parts of their chip. A similar LUT-based design was also proposed [3].

Their soft programmable core has the advantages of easy integration into the ASIC flow, and it will allow users to closely integrate this programmable logic with other parts of the chip. The core will likely be made out of standard cells, however, whose inefficiency will cause significant penalties in area, power, and delay. As such, using these soft cores only makes sense if the amount of programmable logic required is relatively small.

As a precursor to Totem-CPLD we performed work in which we explored the feasibility of making domain-specific reconfigurable PLAs and PALs [4]. This included an architecture generation tool that mapped domains of circuits to a PLA or PAL in such a way that it could remove 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%. The depopulation of the arrays in a PLA is very restrictive to future mappings, however, so we chose not to use PLA depopulation in Totem-CPLD.

We have also performed an analysis of the benefits that are achieved by using domain-specific full-crossbar based CPLDs instead of fixed CPLDs [1]. In this work we created an efficient algorithm for finding domain-specific CPLD architectures by determining an effective size for the PLAs in the architecture, and developed an automated layout process for creating full VLSI layouts of the architectures we specify. These domain-specific architectures beat representative fixed architectures by 4.1x to 9.5x in terms of area-delay product, and they outperformed the best fixed architectures we could find by 1.8x to 2.5x in area-delay product.

The advantage of using full crossbars in a CPLD is that it provides full connectivity between the interconnect and the PLA inputs, as shown on the left half of Figure 2. Full crossbar implementations cause CPLDs to scale quite poorly, however, and area can be drastically improved by utilizing sparse crossbars, as shown on the right half of Figure 2. 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.

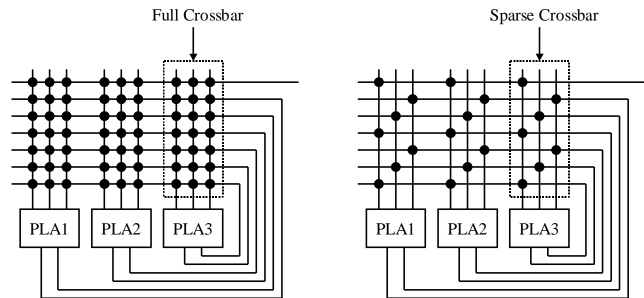

Figure 2. A full crossbar (left) and a sparse crossbar (right). The crossbar gets repeated for each PLA in the CPLD

In this paper we will discuss the important elements of the tool flow developed for creating full-crossbar based CPLDs [1], and present the algorithms necessary for creating sparse-crossbar based CPLD architectures. We will then address the question of

how to add resources to our CPLDs in order to maximize the likelihood that future circuits are supported.

3. Sparse-Crossbar Based CPLDs

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 PLA, as shown in Figure 2. Note that this crossbar is repeated for each PLA in the CPLD architecture.

Figure 3 shows the high level tool flow used for tailoring CPLD logic to a particular application domain. The input from the SoC designer is a specification of the target domain, containing a set of circuits that the architecture must support. These circuits are fed into an Architecture Generator, which will find a CPLD architecture that provides good performance for the selected domain. The architecture description outputted by the Architecture Generator is then sent to a Layout Generator, which creates a full VLSI layout of the specified CPLD architecture in the TSMC .18µ process.

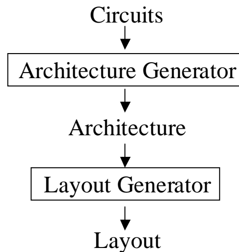

Figure 3. The high level tool flow for Totem-CPLD

3.1 Architecture Generator

The Architecture Generator is responsible for reading in multiple circuits and finding a CPLD architecture that supports the circuits efficiently. The flow for the Architecture Generator is shown in Figure 4. A Search algorithm is used to make calls to a technology-mapper named PLAmap [5], after which the results are analyzed according to area and delay models that we have developed. This includes estimates of the sparse crossbar switch density, which are based on data acquired across a wide range of circuits [14]. 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. A Crossbar Generator is then 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.

3.1.1 Search Algorithm

The goal of the search algorithm is to find a PLA size for which the CPLD architecture efficiently supports the circuits in the application domain. PLAs are characterized by their input, product term, and output capacities, and the same PLA size is used for each occurrence in the CPLD. Several different algorithms were presented and evaluated in [1], and the best of those algorithms is used here.

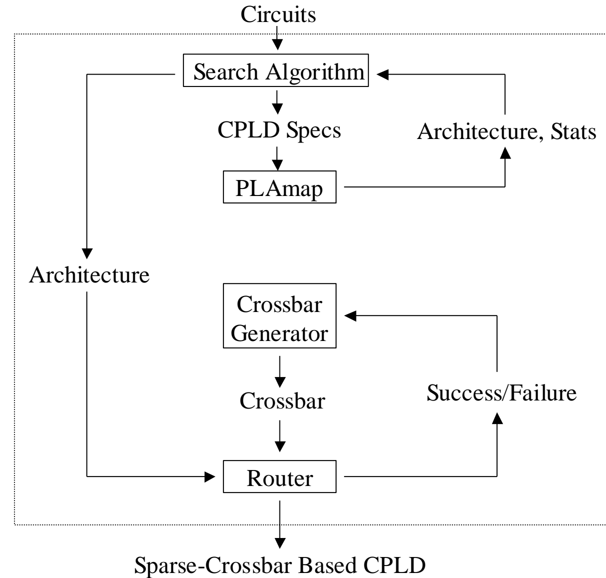

Figure 4. The flow for the Architecture Generator

The PLA search space is 3-dimensional, as we must specify the number of inputs, product terms, and outputs for the PLA. This is a huge space, so we simplify our search by performing three sequential 1-dimensional searches. This is effective largely because the search space is well behaved, and results tend to get better as you approach the optimal point in the 3-D space.

Our search algorithm initiates each step by making a wide sweep of the 1-D space, and then iteratively explores points near the best current point. For each 1-D space, the algorithm collects data for M points before progressing to the next step. Experimentally, a value of M=15 was found to provide good results.

The algorithm first looks for a good input size by taking data points for PLAs with inputs ranging from 4 to 28, with a step size of 4. In this search step, a ratio of 1x-2x-.5x is maintained for the input, product term, and output values, as this was experimentally found to be effective. The best data point is found, and results are taken on either side of it with the largest step size that results in unexplored data points (options are 4, 2, and 1). This is repeated on the best current data point, which is constantly updated, until M runs have been performed for the input step. Once the direct neighbors of a point have been computed, it is eliminated from further explorations; this allows other promising candidates to be explored as well. This is shown in Figure 5.

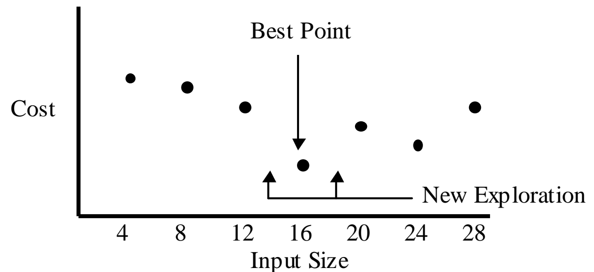

Figure 5. In the search algorithm, the best point is always chosen and the regions to its left and right are explored

For the output step, we lock the input and product term values of the best result found in the input step. We then range the output values from 1 to 25, with a step size of 4, while keeping the input and product term values fixed. We then repeat the search algorithm mentioned above. The product term step repeats this process, with product term values ranging from 10 to 90, fixed input and output values, and a step size of 8 (so possible step sizes are 8, 4, 2, and 1 now). Note that the product term step has a larger interval and step size than the previous steps, so we allow it to run slightly more than M runs in order to provide the appropriate coverage of it's 1-D space.

This algorithm can be seen as exploring the best regions one at a time. It will explore a promising region until it runs out of granularity, then will turn to the second best region, and so on. In this way it also considers multiple parts of the 1-D space, as determined by the value chosen for M. For more details, see [14].

3.1.2 PLAmap

We employ a CPLD technology-mapper named PLAmap, which is currently the best technology-mapping algorithm for CPLDs [5]. PLAmap is a performance driven mapping algorithm whose goal is to minimize the delay/depth of the mapped circuit. The algorithm works on a directed acyclic graph (DAG) in three stages: the first stage labels all the nodes according to their logic depth, the second stage maps these nodes into PLAs, and the third stage attempts further packing of the PLAs in order to reduce the PLA count. PLAmap is run by providing it with a PLA size (inputs, product terms, outputs) and a circuit to be mapped, after which PLAmap returns the number of PLAs required for the mapping, the depth of the mapping, and other mappings statistics.

3.1.3 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 [6].

The general idea presented by Lemieux and Lewis is that maximum crossbar routability will be acquired if subsets of input wires span as many output wires as possible. This makes intuitive sense, as providing a larger set of reachable outputs should make it more likely that each input can be assigned to a unique output. Next, by representing crossbar input lines as vectors of 1s (switch present) and 0s (no switch present), the switch placement problem is shown to be equivalent to the problem of creating a communication code which maximizes the hamming distance between different code words. By utilizing a cost function that has been used for creating efficient communication codes [7], shown below, an efficient switch placement can be found. In the equation, bv_x and bv_y represent the bit vectors for input rows x and y respectively, and the cost function is minimized across all crossbar input vectors pairs. A more detailed treatment of this algorithm can be found in [6].

$$Cost = \sum_{\forall x, y | x \neq y} \frac{1}{HammingDistance(bv_x, bv_y)^2}$$

This rest of this subsection will introduce our implementation of the sparse-crossbar-generation algorithm from [6]. The goal of the algorithm is to create a sparse crossbar of maximum routability given the values n (inputs), m (outputs), and p (switches).

3.1.3.1 Initial Switch Placement

To initiate the algorithm, p switches must be placed such that they are evenly distributed both on the input and output lines. Requiring a smooth switch distribution will help us obtain good routability, as each line will be roughly as connected as any other line We developed a simple, deterministic routine that achieves this smooth switch distribution [14]; an example is shown in Figure 6 for a crossbar with 12 inputs, 8 outputs, and 53 switches.

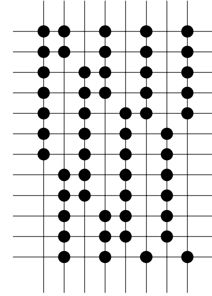

Figure 6. The initial switch placement for a 12-input, 8-output, 53-switch crossbar

3.1.3.2 Moving Switches

Switches are initially placed in a very regular manner, so in order to obtain good routability from our sparse crossbar we must modify the switch placement profile. We must be careful, however, to rearrange the switches in such a way that they are still evenly distributed among the input and output lines. The two switch movements presented in [6] enforce this property, as they ensure that switch movements do not change the number of switches that exist on an input or output line (see Figure 7). At the intersection of two input lines and two output lines, if switches exist on one of the diagonals but not on the other diagonal then the switches can be moved as shown. This ensures that each of the input and output lines retains the same number of switches, while providing a new and possibly better switch placement.

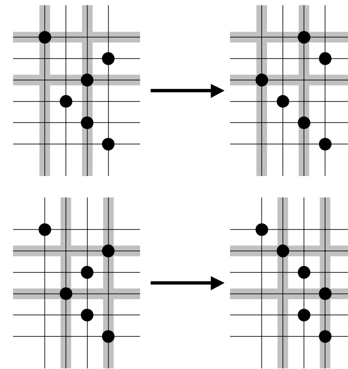

Figure 7. Switches must occur on one diagonal of the intersecting lines, but not on the other diagonal

Switch moves are proposed by choosing random input and output lines. Lemieux and Lewis initially used simulated annealing to conditionally accept moves based on cost and temperature criteria, but they found that negative cost moves were nearly always found again and undone, and that a greedy scheme worked just as well. We therefore reproduce this, and use a purely greedy strategy for accepting moves.

3.1.3.3 Algorithm Termination

The algorithm in [6] allows the user to specify how many switch movements are attempted. Our process is automated, however, and must know how many switch moves should be attempted as a function of n, m, and p in order to obtain a good sparse crossbar. Experiments showed that attempting $n*m$ switch moves provided good results with a reasonable runtime: beyond this number of moves, the tool was never able to improve the cost function by more than an additional 1% regardless of how many moves were attempted.

3.1.4 Router

Our sparse crossbars do not provide full capacity, so we use a routing algorithm to ensure that signals can reach their destinations. PLAmap provides the necessary mapping information, which includes the inputs and outputs of each PLA in the CPLD mapping. Inputs and outputs can be assigned to any track within a PLA, and every physical PLA output feeds a specific wire in the interconnect (see Figure 2), so this results in a very simple routing graph. Only two types of routing decisions need to be made: which physical PLA output track an output will map to, and which PLA input track a horizontal routing track will connect to.

We use the Pathfinder [8] routing algorithm for our CPLD architectures. All of our resources are of similar delay, so we can use their negotiated congestion router, which does not account for delay. The corresponding cost function for using a resource is shown below, where b_n is the base cost, h_n is the history cost, and p_n is the sharing cost. We set b_n to 0 so that the cost function simply considers $h_n * p_n$, and we allow the algorithm to run for 1000 iterations before judging a mapping to be unroutable with a given sparse crossbar. We can use an extremely large number of iterations because of how simple our routing graphs are: a single iteration of the pathfinder algorithm on our largest architecture takes only a couple seconds.

$$cost_n = (b_n + h_n) * p_n$$

3.1.5 Determining Switch Count

In order to determine how many switches our crossbars need, our tool performs a binary search on the crossbar switch count. For an n input, m output crossbar, the minimum switch count is n and the maximum switch count is $n*m$. Starting with a crossbar with $.5(n*m)$ switches, we iteratively create a sparse crossbar and route on the specified CPLD architecture, adjusting the switch count according to whether our route fails or succeeds. We do this to maximum granularity, which requires only log $(n*m)$ iterations of the router.

3.2 Switch Smoother

The resulting sparse crossbar has a switch distribution that is going to look fairly random. Because of this, there are likely to be regions in the crossbar that have relatively high switch densities, as well as regions that have relatively 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. This issue of switch smoothing was not considered in the work done by Lemieux and Lewis [6].

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 in Figure 8, 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 in Figure 9. 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.

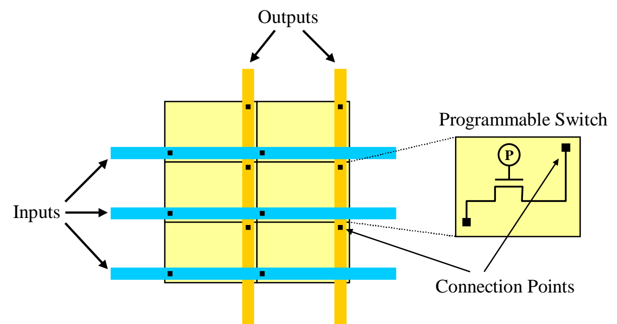

Figure 8. Layout of full crossbars is easy because there is a switch at the intersection of every input and output line

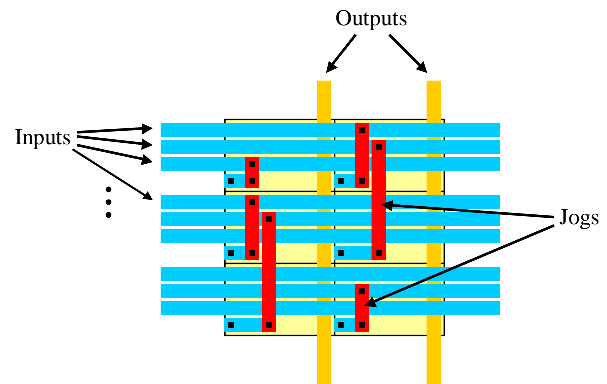

Figure 9. Layout of sparse crossbars requires us to pack the input lines closer together and to add wire jogs in order to connect input lines to the proper switches

If our layout contains N input wire pitches per vertical switch pitch, then we can minimize the number of vertical jogs by requiring that each output line be attached to exactly one switch for every N input lines. This will ensure that each input line attaches to a switch that is underneath it. Since we are creating an effectively random switch placement in our sparse crossbars, it is unlikely that the switches will initially be aligned in this manner.

The input lines in our sparse crossbar can be permuted, however, 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 vertical wire jogs pitches.

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

Figure 11 provides the pseudocode for the switch smoothing algorithm. Figure 10 shows layouts derived from applying the switch smoothing algorithm to a very small crossbar, displaying the improvement in the lengths and pitches of the required wire jogs.

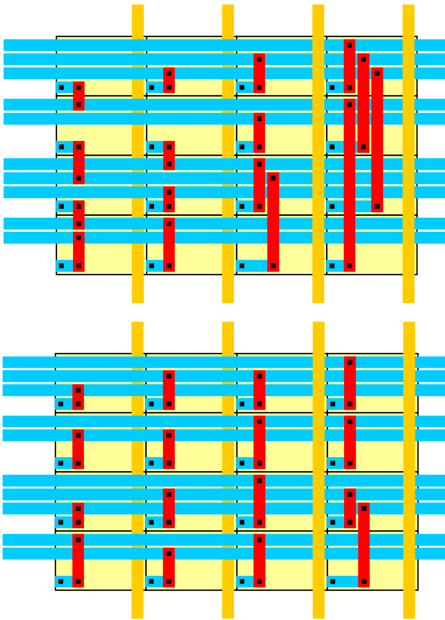

Figure 10. Representative layouts of the unsmoothed (top) and smoothed (bottom) crossbars from Figure 11.

3.3 Layout Generator

The Layout Generator is responsible for taking the CPLD architectures description from the Architecture Generator and turning it into a full VLSI layout. It does this by intelligently 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 [13]. The layouts are designed in the TSMC .18-micron process.

```
smoothSwitches(n, m, p, sw) {
    ordered_sw[n][m];     //matrix to hold the smoothed switch matrix
    swCount[m];      // number of switches placed on each output line
    for(i=0;i<m;i++) { swCount[i]=0; }
    inPlaced[n];      //whether we've place an input line yet
    for(i=0;i<n;i++) { inPlaced[i]=0; }
    swDens = p/(n*m);     //switch density
    tempCost;

    for(i=0;i<n;i++) { //must place n smoothed input lines
        tempInput = -1;
        bestCost = infinity;
        for(j=0;j<n;j++) { //check which of n lines is next
            //Calculate the cost of inserting input line j next
            tempCost = getSmoothCost(sw[j], I, swCount, swDens);
            if(tempCost < bestCost) {
                if(!inPlaced[j]) { //if not placed yet, mark as best seen
                    tempInput = j;
                    bestCost = tempCost;
                }
            }
        }
        ordered_sw[i] = sw[tempInput];   //set proper line
        for(k=0;k<m;,++) {   //update switches per output
            swCount[k] += ordered_sw[i][k];
        }
        inPlaced[tempInput] = 1;  //mark line as placed
    }
    sw = ordered_sw;  //set switches to smoothed result
}
```

Figure 11. Pseudocode for the switch smoothing algorithm

Figure 12 displays a small CPLD that was created using the Layout Generator. For clarity's sake, the encoding logic required for programming the RAM is not shown, but would appear along the left and bottom of the CPLD. Pre-made cells exist for every part of the PLA and crossbar, including the RAM encoding logic. The Layout Generator simply puts together the pre-made layout pieces as specified by the 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.

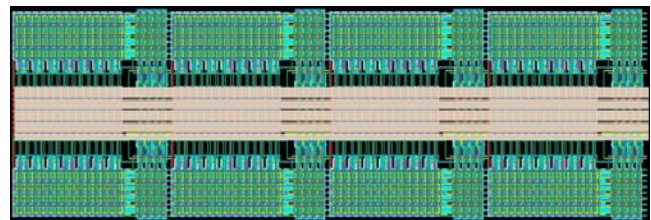

Figure 12. The created sparse-crossbar based CPLD layout

4. Sparse-Crossbar CPLD Methodology

We have compiled five domains of circuits from various sources. These include combinational and sequential domains, consisting of files gathered from the LGSynth93 Benchmark Suite [9]. These circuits are grouped simply for their combinational or sequential characteristics. The remaining three domains consist of floating-point, arithmetic, and encryption files respectively. These files were accumulated from a variety of sources, including OpenCores.org [10], from Altera software developers, as Quartus 2 megafunctions, and from floating-point libraries [11].

We have also created both area and delay models for the sparse-crossbar based CPLD architectures. The area model was compiled by using the actual sizes of the VLSI components that are tiled to create the full CPLD layout, and provides the exact size of the full architecture. The delay model was created by running simulations on a variety of CPLD architectures using hSpice. More information about the circuits and models that we use can be found in [14].

5. Sparse-Crossbar CPLD Results

We have chosen a layout topology that restricts us to 8 or fewer jog pitches per crossbar output. This means that any crossbar requiring more than 8 jog pitches would force us to spread out our switches in the east-west direction, wasting silicon area in our final layout. Table 1 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 area penalties in the final layouts if the crossbar-smoothing algorithm had not been applied. Also note that the final smoothed crossbars easily meet our requirement of 8 or fewer jog pitches.

Table 1. 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 2 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 from [1]. 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-based CPLDs.

Table 2. 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**

One interesting result from Table 2 is that each domain improved by roughly the same amount. This would be expected if the domain-specific architectures were roughly the same size, but they are not. Table 3 displays this, including the normalized area of the full-crossbar based implementations and the percentage of area dedicated to routing for each implementation. Intuition suggested that the domains that are more routing heavy would acquire greater gains by using sparse crossbars, which did not occur.

Table 3. Switch densities of sparse crossbars related to the area of the sparse-crossbar-based CPLD

Domain	Full XBAR		Sparse XBAR
	Normalized Area	Routing Area	Switch Density
Sequential	1.00	61.5%	13.6%
Combinational	22.12	84.5%	20.0%
Arithmetic	21.03	93.0%	28.8%
Floating Point	97.26	96.8%	35.5%
Encryption	1006.86	98.7%	41.8%

The reason that the sparse crossbars are providing similar area improvements is that the sparseness of the crossbars is varying according to the size of the CPLD, which is also shown in Table 3. The circuits in the sequential domain have fewer signals to route, and they can therefore subsist on very sparse crossbars. The circuits in the encryption domain, however, have a large number of signals to route, and require more connectivity in their crossbars.

5.1.1 Using Other Evaluation Metrics

We also ran our sparse-crossbar based CPLD flow in both area driven and delay driven modes. The results of this are shown in Table 4, which displays the results of running our chosen algorithm on each of our five domains for area-delay driven, area driven, and delay driven metrics. All results are normalized to the area-delay driven results.

Table 4. Results of running our chosen algorithm in area-delay driven, area driven, and delay driven modes for sparse-crossbar based CPLDs

Sparse	Area*Delay Driven			Area Driven			Delay Driven		
XBAR	Area	Delay	A*D	Area	Delay	A*D	Area	Delay	A*D
comb	1.00	1.00	1.00	0.80	0.93	0.75	1.13	0.81	0.91
seq	1.00	1.00	1.00	1.07	1.17	1.25	1.49	0.83	1.24
fp	1.00	1.00	1.00	0.92	1.74	1.60	1.52	0.68	1.04
arith	1.00	1.00	1.00	0.93	1.18	1.10	3.39	0.61	2.08
enc	1.00	1.00	1.00	1.04	1.43	1.50	1.60	0.48	0.77
GeoMean	**1.00**	**1.00**	**1.00**	**0.95**	**1.26**	**1.20**	**1.69**	**0.67**	**1.13**

Table 4 shows that, when creating sparse-crossbar based CPLD, both area and delay driven modes are successful at optimizing for their target metric. Using area as our metric, our domain-specific architectures require only .95x the area of those found with area-delay as the metric. Using delay as the metric, we find architectures that require only .67x the delay of those found with area-delay as the metric. The metric that is not being directly considered, however, tends to show degraded performance. When PLAmap is successful at optimizing for delay, it causes the area minimizing heuristics to be less successful. Conversely, when PLAmap is less successful at delay optimization, the area minimizing heuristics are more successful.

An interesting side note from Table 4 is that three of our area and delay driven architectures actually provide better area-delay performance than our area-delay driven architectures. This is an artifact of the fact that we must route our circuits on these sparse-crossbar based CPLDs architectures, determining the minimum switch count on which the circuits will route. The points that

display improved area-delay performance are instances where the router was somewhat lucky and successfully routed all the circuits on a small sparse crossbar, therefore providing better delay and area performance than was predicted by our models.

6. Adding Capacity to CPLDs

Many SoC designers who use our flow will know the circuits that they wish to implement in reconfigurable logic on their device, but some will not. This could be because the entire SoC design is still being finalized, because they are unsure of which elements will be implemented in reconfigurable logic, or even because the circuit they wish to implement in reconfigurable logic is likely to be modified or upgraded in the near future. 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. In this section, we will address this question by determining what resources should be added to a CPLD in order to support future, unknown circuits.

There are a limited number of CPLD architectural characteristics that can be augmented in our architectures. We are using sparse crossbars to connect interconnect wires to the PLAs, so the switch density of the crossbars is something that can be modified in order to support routing rich designs. In terms of logic resources, our CPLDs exclusively use PLAs. PLAs can be modified in terms of their input, product-term, and output counts, and we can also modify how many PLAs we have in our architecture. This gives us a total of five variables to consider when adding capacity: crossbar switch density, PLA input count, PLA product-term count, PLA output count, and PLA count.

It may also be beneficial to augment all three PLA variables at once, so we will consider two strategies in which we do this: the first strategy will be to augment all three variables using their existing in-pt-out ratio, and the second will be to simultaneously augment each of the variables by an additive factor. Lastly, there might be some utility to a strategy that uses larger PLAs and allows an increase in PLA count: we will consider this strategy as well, using a fixed multiplicative factor to make the PLAs larger. Table 5 lists the CPLD augmentation strategies that we will employ in this section.

Table 5. Strategies for adding capacity to our CPLDs

Strategy	Description
Switch Density	Crossbar switch density = c*base, allow extra PLAs
PLA Count	Allow extra PLAs of base size
PLA Inputs	Augment PLA inputs, keep PLA count fixed
PLA Product Terms	Augment PLA product terms, keep PLA count fixed
PLA Outputs	Augment PLA outputs, keep PLA count fixed
PLA Size1	PLA size = (c*in, c*pt, c*out), keep PLA count fixed
PLA Size2	PLA size = (d+in, d+pt, d+out), keep PLA count fixed
PLA Count and Size1	Put λ extra area towards PLA count, 1-λ toward PLA size

For the strategies which have fixed PLA sizes but which allow additional PLAs, we find our mapping simply by providing PLAmap with the PLA size of the architecture. PLAmap then provides us with the required PLA count, and the performance characteristics are calculated. For strategies that alter the PLA sizes while keeping the PLA count fixed, we iteratively call PLAmap with larger and larger PLA sizes (using the smallest possible increment) until a mapping fits the PLA count constraint. If the PLA variables in question get increased to three times their initial values without finding a mapping that fits the PLA count constraint, then it is considered a failure.

7. Adding Capacity Methodology

We want to simulate a situation in which the SoC designer knows the general domain of circuits that will be implemented in reconfigurable logic, but does not necessarily know the exact circuits. We accomplish this by taking the domains that we already have and by removing one or more of the circuits in order to create reduced domains. These reduced domains are used to create domain-specific architectures according to our tool flow. If the architecture created for the reduced domain is different than the architecture for the full domain, then we reintroduce the removed circuit(s) and determine what additional CPLD resources (if any) are required to implement the new circuit.

This process was applied to every circuit in each of our five domains. Each circuit was individually removed, and the reduced architecture was created. We also created five new sub-domains by grouping very similarly sized circuits from the main domains, and we applied this process to each circuit in these sub-domains as well. These ten domains had a total of 92 circuits in them. Each of the 92 circuits was removed from its domain, and 35 of the corresponding reduced architectures were different than their full-domain architectures. Using knowledge of the 35 cases where removing a single circuit resulted in a different architecture, we then created 14 more such cases by removing multiple circuits from the domains. In all, we created 49 reduced architectures that were different from the architectures obtained by the full domains.

With these 49 interesting reduced architectures, we will be intelligently adding resources to the architectures in order to determine what is required to support the removed circuits. The basic question we wish to answer is this: given an architecture augmentation strategy, how much of an area penalty must we tolerate in order to support the additional circuits? We will analyze this information using graphs of the form shown in Figure 13. In the figure, the x-axis displays the area requirement of a given strategy, normalized to the reduced domain. The y-axis displays the number of domains that can support their additional circuits for the specified area requirement. Results that are toward the upper left of the graph are good, because many domains can be supported by the strategy with a small area overhead.

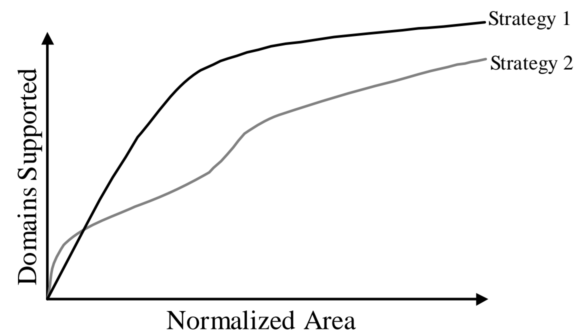

Figure 13. The basic graph we will use to examine our different CPLD augmentation strategies

We will also compare the different resource strategies according to the geometric mean of their data points. Each strategy will have data points in the exact same y-locations on the graphs, so their x-values will be normalized to a particular strategy, and their means calculated and compared: this will give a numerical value that can be used for comparisons. The graphical data can be difficult to compare at times, and the graphs often do not include all the data points (in order to increase the visibility of the interesting areas), so a numerical metric will be helpful.

8. Adding Capacity Results

When mapping to reconfigurable architectures, providing sufficient routing resources is necessary in order to fully utilize the architecture's logic elements. This suggests that finding a good crossbar switch density should be the first thing done in our architectures, as providing sufficient crossbar connectivity will allow any increased logic resources to be utilized efficiently.

We therefore started by running an exploration of how switch density affects the amount of area required to map future circuits to our architectures. In this exploration, the reduced architectures were given additional switches according to a multiplicative factor and allowed to use as many extra PLAs as necessary in order to map the additional circuits. The results of this exploration are shown in Table 6 and Figure 14, representing architectures with the *base* number of switches, *base**1.05 switches, *base**1.10 switches, *base**1.20 switches, and *base**1.50 switches.

Table 6. Results of adding switches to crossbars and allowing any number of PLAs. Results are geometric mean

Switch Count	Mean Result
BASE	1.00
BASE * 1.05	1.01
BASE * 1.10	1.03
BASE * 1.20	1.07
BASE * 1.50	1.18

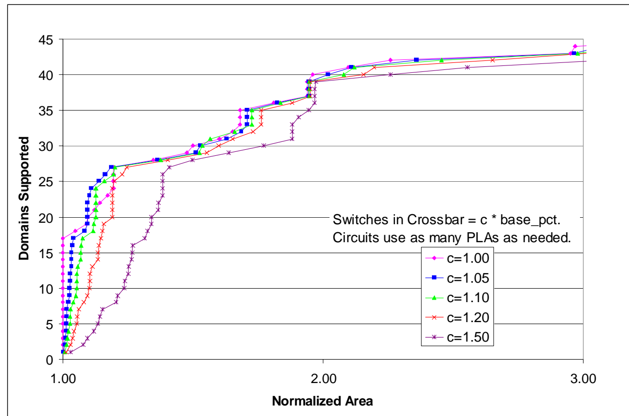

Figure 14. Results of adding switches to crossbars

The data shows that good results are obtained when either 0% or 5% extra switches are added to the basic crossbar in the CPLD. While both of these strategies are effective, two factors led us to prefer the strategy of adding 5% to the switch count. First, Table 3 showed that larger architectures require a higher switch density than smaller architectures, suggesting that we will want to provide more switches to support larger circuits. Second, a higher switch count makes it less likely that we will have routing problems that will cause us to spread out mappings among additional PLAs.

We next considered the logic in our architectures, which can be augmented by adding PLAs, by adding inputs, product terms, or outputs to our PLAs, or by adding all three of these variables to our PLAs in either a multiplicative manner, c*(in-p-out), or in an additive manner, c+(in-pt-out). We performed each of these experiments with our 49 interesting reduced architectures, and acquired the results shown in Table 7 and Figure 15.

Table 7. Results of adding logic resources. All strategies use 5% more switches than base architecture, and failure rate details how many domains were not supported by a strategy.

Strategy	Mean Result	Failure Rate
#PLAs	1.00	0%
#PLA inputs	1.05	51%
#PLA pterms	1.02	49%
#PLA outputs	1.02	49%
c*(in-pt-out)	1.16	0%
c+(in-pt-out)	1.24	0%

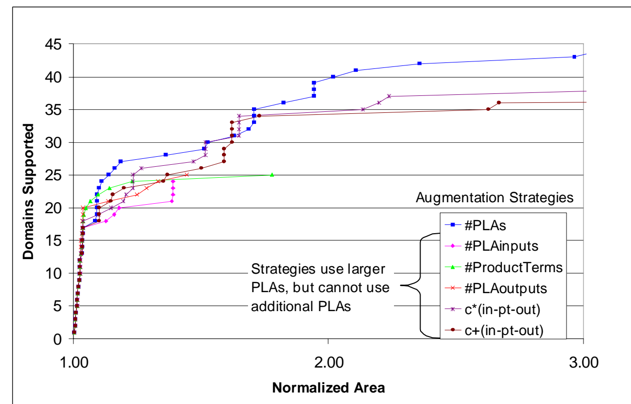

Figure 15. Results of adding logic resources.

As the data shows, the most efficient strategy for supporting additional circuits is simply to add more PLAs to the architecture. The multiplicative and additive strategies are also capable of supporting all of the additional circuits, but the strategies in which we simply add inputs, outputs, or product terms to the PLAs are insufficient to even support all the additional circuits.

The strategies of adding PLAs and adding inputs/outputs/product terms to the PLAs multiplicatively are both shown to perform reasonably well in Table 7, so another idea is to attempt a mixture of these two strategies. In this hybrid strategy, λ of the additional area resources are provided to additional PLAs, while $(1-\lambda)$ of the additional area resources are provided to larger PLAs (using a multiplicative scaling factor), where $0 \leq \lambda \leq 1$. Resources are slowly added, using these ratios, until the removed circuit is supported by the architecture. Table 8 and Figure 16 display the results of running this new hybrid strategy with λ values of 1.00, 0.75, 0.50, 0.25, and 0.00.

Table 8. Results of using the new hybrid strategy. All strategies use 5% more switches than the base reduced architecture, and can use as many PLAs as required

λ	Mean Result
1.00	1.00
0.75	1.05
0.50	1.05
0.25	1.17
0.00	1.16

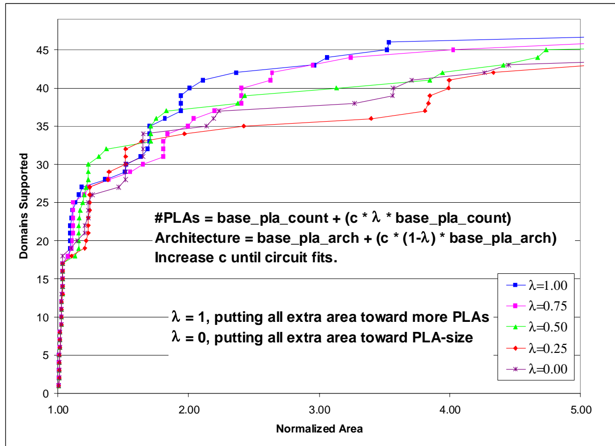

Figure 16. Results of the hybrid strategy.

As shown, the strategies of giving 75% and 50% of the additional area resources toward more PLAs are both shown to be relatively effective, but the data still demonstrates that the most effective method of supporting future circuits is simply to add additional PLAs of the base size.

9. Conclusions

This paper introduced the incorporation of sparse crossbars into our CPLD architectures, including the methods used to create the sparse crossbars for a specific architecture. We introduced a switch smoothing algorithm which takes a sparse crossbar and permutes its input lines such that they are amenable to layout. After running through the switch smoothing algorithm, crossbars which would have required up to 48 vertical wire jogs were reorganized such that they require no more than 3 wire jogs, allowing for fully compact crossbar layouts.

The sparse-crossbar based CPLDs described in this paper require only .37x the area and .30x the delay of the full-crossbar based architectures from [1]. The performance gains were evenly spread across the domains, despite the fact that larger CPLDs seem to have more to gain from reducing the routing area. This was explained by the fact that domains that require more routing resources are not able to depopulate their crossbars as much as domains that require fewer routing resources.

Also, using area and delay driven modes, we were able to find architectures that were more optimized for their respective metrics. Architectures found in area driven mode required .95x the area and 1.26x the delay of those found using area-delay, and architectures found in delay driven mode required .67x the delay and 1.69x the area of those found using area-delay as the metric.

This paper also explored the concept of adding capacity to our domain-specific CPLDs, for situations where the SoC designer does not know all the circuits they wish their reconfigurable logic to support. Our results showed that, in terms of routing resources, it is good to add 5% to the switch density of the sparse crossbars used in the CPLD architectures. In terms of logic resources, the most area-efficient method of supporting future circuits is simply to add more PLAs of the same size found in the base architecture. This is consistent with the strategy that is most commonly employed with reconfigurable devices, in which additional capacity is provided by adding more of the resources found in the base architecture.

Acknowledgments

The assistance of Mike Hutton and Swati Pathak at Altera was essential, as they provided the blif dumper for Quartus and many useful circuits. Tom Lewellen at UW also provided us with several useful circuits, and Steve Wilton provided a vqm to blif converter that was vital in our early data accumulation. Deming Chen provided assistance with PLAmap.

Mark Holland was supported in part by an NSF Fellowship, and Scott Hauck by a Sloan Fellowship.

References

[1] M. Holland, S. Hauck, “Automatic Creation of Domain-Specific Reconfigurable CPLDs for SoC”, *International Conference on Field Programmable Logic and Applications*, 2005.

[2] A. Yan, S. Wilton, “Product-Term Based Synthesizable Embedded Programmable Logic Cores”, *IEEE International Conference on Field-Programmable Technology*, pp. 162-169, 2003.

[3] N. Kafafi, K. Bozman, S. Wilton, “Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores”, *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2003.

[4] M. Holland, S. Hauck, “Automatic Creation of Reconfigurable PALs/PLAs for SoC”, *International Conference on Field Programmable Logic and Applications*, pp. 536-545, 2004.

[5] D. Chen, J. Cong, M. Ercegovac, Z. Huang, “Performance-Driven Mapping for CPLD Architectures”, *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2001.

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

[7] A. El Gamal, L.A. Heinachandra, I. Shperling, V. K. Wei, “Using simulated annealing to design good codes.”, *IEEE Transactions on Information Theory*, 33(1):116:123, January 1987.

[8] 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.

[9] “1993 LGSynth Benchmarks”, <http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.cbl.ncsu.edu/CBL_Docs/lgs93.html> (March 25, 1997).

[10] OpenCores.org, “OPENCORES.ORG”, <http://www.opencores.org/> (2004).

[11] M. Leeser, “Variable Precision Floating Point Modules”, <http://www.ece.neu.edu/groups/rpl/projects/floatingpoint/> (May 20, 2004).

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

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

[14] 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.