Domain-Specific Reconfigurable PAL/PLA Creation for SoC

Mark Holland, Scott Hauck
University of Washington
Dept. of Electrical Engineering
Seattle, WA
{mholland, hauck}@ee.washington.edu

Abstract

As we move to System-on-a-Chip (SoC), where multiple types of resources are integrated on a single chip, it is important to consider how to best integrate reconfigurability into these systems. Reprogrammable logic 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/delay/power gains can be achieved over current, more general reconfigurable fabrics. This paper presents tools that automate the creation of domain specific PLAs and PALs for SoC, including an Architecture Generator for making optimized arrays and a Layout Generator for creating efficient layouts. By intelligently mapping netlists to PLA and PAL arrays, we can reduce 63%-75% of the programmable connections in the array, creating delay gains of 17%-32% over unoptimized arrays.

Introduction

As device scaling continues to follow Moore's Law, chip designers are finding themselves with more available chip real estate. This is true in the design realm of System-on-a-Chip (SoC), where individual, pre-designed subsystems (memories, processors, DSPs, etc.) are integrated together on a single piece of silicon in order to make a larger device. In this new world of SoC integration, it is natural to question the future of reconfigurable logic.

Reconfigurable logic fills a useful niche between the flexibility provided by a processor and the performance provided by custom hardware. Traditional FPGAs, however, provide this flexibility 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 will provide improved performance not only for implementing the designs used in architecture development, but also for implementing other similar designs that might become interesting in the future.

Previous work on Totem [Phillips01, Sharma01, Eguro02, Compton03, Phillips04, Eguro05, Sharma05, Hauck06] has focused on using a 1-D RaPiD array [Ebeling96, Cronquist99] in order to provide reconfigurable architectures for domains that use ALUs, Multipliers, RAMs, and other coarse grained units. But many domains do not benefit from the use of coarse-grained units, and would require a finer-grained fabric. For example, users who want their reconfigurable logic to support state machines, control logic, or other random functions, would have little to no use for these coarse-grained units. An appealing solution for these users would be to create reconfigurable PLAs and PALs, which are efficient at representing seemingly random or non-regular logic functions.

PLAs and PALs are devices that directly implement two level sum-of-products style logic functions [Fleisher75]. They do this with the use of a programmable AND-plane that leads to either a programmable OR-plane (PLA, shown in Figure 1) or a fixed OR-plane (PAL). PLAs are slightly more flexible than PALs because of their programmable OR-planes, while PALs are usually smaller due to their fixed OR-plane.

As shown, PLA and PAL arrays are defined by how many inputs, product terms, and outputs they can represent. Additionally, PAL arrays are constrained by the sizes of their fixed output gates. These are variables that can be adjusted in order to provide arrays that are tailored to specific tasks.

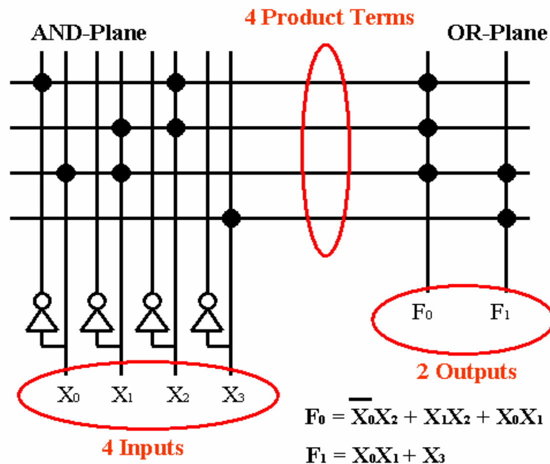

Figure 1. Example PLA. PALs have fixed OR gates instead of the programmable OR-plane.

A desirable aspect of PLAs and PALs is that they provide very predictable timing, an attribute that LUT-based reconfigurable architectures usually cannot provide. The reconfigurable portion of an SoC is likely to be closely coupled with other blocks on the device, and the ability to provide predictable timing may be very attractive to SoC designers. Additionally, the logic density of PLAs and PALs is very similar to LUT-based architectures when considering relatively small applications. For example, control logic and random logic can usually be represented very efficiently with sum-of-products equations, and are therefore represented very well by these arrays. Providing SoC designers with reconfigurable PLAs and PALs will give them the ability to support many different fine-grained functions, or even perform bug fixes or other post-fabrication modifications, all while providing respectable logic density and predictable timing.

The next section of this paper presents some relevant background material, as well as a discussion of other related works. We then cover the flow of our PLA/PAL generation tool, including its two major components – the Architecture Generator and the Layout Generator. This is followed by a discussion of our research methodology and the results of our study. We conclude with a discussion of the contributions of our work, as well as possible avenues of future work.

Background

Reconfigurable PLAs and PALs have existed for many years in commercially available CPLDs, and are produced by companies including Xilinx, Altera, and Lattice. CPLDs are typically sets of reprogrammable PALs or PLAs connected by fairly rich programmable routing fabrics, with other hardware added to increase the functionality of the devices. These commercial CPLDs, however, suffer from the same drawbacks as commercial FPGAs: their generality, while allowing them to be used in many domains, costs them in terms of area, delay, and power. It would be desirable to tailor such devices to specific domains and requirement, which is exactly what Totem does.

Many papers have been published with respect to PAL and PLA architectures, but very few of them have aimed at creating reconfigurable arrays or arrays for SoC. The most applicable [Yan03] explores the development of unidirectional 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 unidirectional 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.

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. In fact, their work has estimated that the use of a standard cell implementation would require 6.4x the area of a "hard" core implementation of the same fabric [Wilton05]. As such, using these soft cores only makes sense if the amount of programmable logic required is relatively small.

Our process differs from [Yan03] in that we create domain-specific hard cores to be used in SoC. Our tools intelligently create a PLA or PAL architecture that fits the specifications provided by the designer, and use pre-optimized layouts to create a small, fast layout of the array that can be immediately placed onto the SoC. This results in area, delay, and power improvements over pre-made programmable cores or soft cores, and is automated in order to provide very fast turnaround time.

Another similar work presents a high-performance programmable logic core for SoC applications [Han05]. Their architectures are similar to those in [Yan03] in that they are unidirectional and they use product term based logic, but they do so while using a unique dynamic logic family called OPL. OPL is a precharged-high logic family, so only discharging is necessary upon function evaluation. Due to this, they show that OPL designs provide 5x speedups over conventional circuit design styles when implementing circuits that map well to wide NOR gates. The work also introduces a novel product-term-based logic structure that utilizes OPL-friendly NOR gates. While most product-term-based reconfigurable logic provides a fixed input, product term, and output capacity, the logic structures proposed in this paper provide further gains by allowing these amounts to be variable. This avoids the area losses caused by unutilized logic resources in most product-term-based designs, because most PLAs are only partially utilized.

One important consideration for this OPL-based design is clock distribution, as the logic family requires successive clock phases to be present with very short separation times. This requires considerable clock-generation overhead, which takes up area that could otherwise be utilized for logic. Power consumption is also increased due to the need for a large number of minimally spaced clocks. The goal of this device, though, was increased speed. A test chip was produced and timing values extracted, and results showed that this new architecture provided an average speedup of 3.7x over a Xilinx Virtex-E FPGA.

This work definitely shows that dynamic PLA-based reprogrammable logic is feasible, but only with a significant amount of design effort. The design and generation of multiple, minimally spaced clocks is a significant task. More importantly, any changes that are made to the logic resources will require significant redesigning of the clocks in order to ensure that correct timing is still achieved. We wish to automate the process of creating reprogrammable PLA and PAL architectures, which involves choosing the appropriate array sizes on the fly. Automatically specifying and laying out a functional clock network for dynamic reprogrammable logic is simply too great of a task for a fully automated process, therefore leading us to eliminate dynamic logic families from our consideration.

While we have chosen not to use standard cells or dynamic logic to implement our reconfigurable arrays, we believe that the algorithms that are presented in this paper would still be effective for designs using these styles. The idea of tailoring a reconfigurable fabric to better fit an application domain is largely implementation independent: regardless of the specific implementation of the resources, area, delay, and power gains can be expected simply due to the removal of unutilized resources. This was displayed in our work utilizing RaPiD arrays, would carry over into PLAs/PALs made of standard cells or dynamic logic, and would even carry over into more traditional 2-dimensional island style FPGA implementations.

In a third related work, a highly regular "River" PLA (RPLA) structure is proposed which provides ease of design and layout of PLAs for possible use in SoC [Mo02]. Their proposal is to stack multiple PLAs in a uni-directional structure using river routing to connect them together, resulting in a structure that benefits from high circuit regularity with predictable area and delays. Also touched upon is a reconfigurable version of RPLA, called Glacier PLA (GPLA), which would retain the benefits of RPLA in addition to being programmable.

GPLAs are similar to our work in that they are hard programmable cores that can be integrated into SoC. Their focus on circuit regularity and area/delay predictability prevent them from obtaining high performance, however. Our arrays will be tailored to the specifications of the designer, allowing us to both better suit their exact needs and to make modifications that will result in better area, delay, and power performance.

Much work has been done on the software tools for minimizing sum-of-products style equations so that they will require smaller PLA or PAL arrays for their implementation. To date, the most successful algorithm for minimizing these equations is Espresso, which was developed at Berkeley in the 1980s [Brayton84].

Espresso runs faster and is more effective than other minimization existing methods. Espresso's basic strategy is to iteratively expand its terms (in order to encompass and remove other terms) and then reduce its terms (to prepare for a new, different expansion). This expand and reduce methodology results in a final equation that has a near-optimal number of product terms. Espresso also reduces the number of literals in the equation, which is equivalent to

minimizing the number of connections that need to be made in the PLA or PAL array.

Tool Flow

Our domain-specific PLAs/PALs are created using the flow shown in Figure 2. The input from the customer is a specification of the target domain, containing a set of circuits (in .pla format) that the target architecture must support. In addition to the circuits, there may be a combination of delay or area requirements that the architecture will need to meet. The goal is to create a reconfigurable architecture that efficiently supports the provided circuits, and which is thus likely to efficiently support future circuits that might arise in the same computational domain.

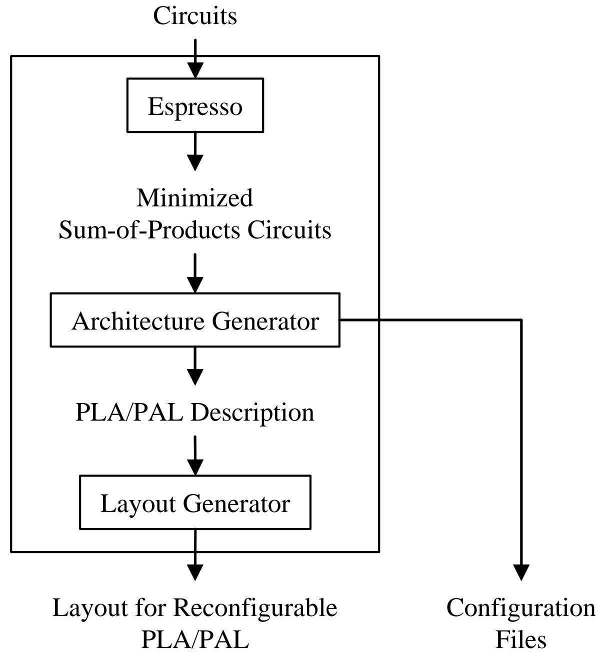

Figure 2. Our PLA/PAL Generation tool flow.

The circuits are first processed by Espresso in order to minimize the number of product terms and literals that they contain. For PLAs this is done using Espresso's default options, and for PALs we run Espresso independently on each PAL output, since the output sharing normally performed by Espresso would be detrimental to PAL implementations. The use of Espresso allows us to implement the circuits using less silicon.

The resulting minimized circuits are then fed into the Architecture Generator, which attempts to create the smallest single PLA or PAL array that is capable of supporting every circuit. Only one circuit can be executing in the hardware at any given time. The Architecture Generator outputs information specifying the chosen PLA or PAL array, and also provides configuration files for configuring each circuit on the specified PLA or PAL. Additionally, any delay or area requirements provided by the customer can be checked after the Architecture Generator creates the array, as we have accurate models for calculating array delay and area.

After the Architecture Generator creates an array specification, this specification is fed to the Layout Generator, which creates a layout of the array in the native TSMC .18µ process. This layout includes the PAL/PLA array as well as the hardware necessary for programming the array.

Architecture Generator

The Architecture Generator must read in multiple circuits and create a PLA/PAL array capable of supporting all of the circuits. The tool is written in C++. The goal of the Architecture Generator is to map all the circuits into an array that is of minimum size and which has as few programmable connections as are necessary. For a PLA, minimizing the number of inputs, outputs, and product terms in the array is actually trivial, as each of them is simply the maximum occurrence seen across the set of circuits. For a PAL we minimize the number of inputs and outputs the same way as for a PLA, and we minimize the number of product terms in the array by making each output OR gate as small as possible.

Non-programmable PLAs and PALs can achieve further area gains by using folding techniques [Makarenko86]. In a typical PLA, only about half of the inputs are involved in any PLA product term, which seems to advocate the "column folding" of a PLA as shown in Figure 3. Programmable PLAs and PALs, as opposed to non-programmable PLAs and PALs, must be able to support many disparate array mappings. Using column folding on the array drastically hinders the connectivity of the PLA or PAL array, and we feel that this fixed segmentation of the AND-plane is restrictive for our application. Row folding is similarly considered to be too restrictive, and is therefore not considered in this work.

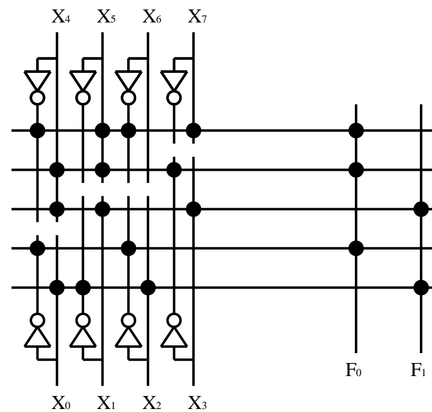

Figure 3. An example of simple column folding. Notice that inputs cannot feed any product term.

The results of removing programmable connections from our arrays is displayed in Figure 4 - the top shows the layout of a complete PLA array, while the bottom shows a PLA with unneeded programmable connections removed. This removal can improve delay by removing capacitive elements from the signal paths, improve power consumption by improving switching time and removing current leakage paths, and might reduce the area after a compactor is applied to the layout.

Figure 5 displays the problem that we face when trying to minimize the number of programmable connections that are necessary in the array. In this example we are trying to map two circuits to the same array (for the sake of this example the circuits, grey and black, implement the same function). A random mapping of the product terms (Figure 5, left) is shown to require 23 programmable connections, while an intelligent mapping (Figure 5, right) is shown to require only 12 programmable connections - a 48% reduction.

In this simple example the circuits happen to be the same, so we were able to obtain a perfect mapping. Circuits that are not identical will also have optimal mappings, which will result in a reduced number of programmable connections. Having fewer connections might allow us to compact the array to save area, and it will lower the capacitance, which will make our array faster.

In order to map circuits intelligently, we must come up with a mapping algorithm. A reasonable starting point is to apply a cost of 1 to a spot where we need a programmable connection, and a cost of 0 to a spot where we don't. This accurately represents our problem because more programmable connections will yield a higher cost – which directly represents the higher area and delay values that the array will produce. Programmable connections in

different locations should produce similar penalties in terms of area and delay, so applying a cost of 1 to all connections is appropriate.

Using this 0/1 cost model, each possible product term matching between two circuits can be assigned a total cost. This converts the product term matching problem into a well-studied problem called optimum bipartite matching, or optimum assignment. An algorithm exists for finding the optimum bipartite matching in $O(m^4)$ time, where m for us is the number of product terms. The optimal algorithm was developed by Kuhn and Munkres [Asratian98].

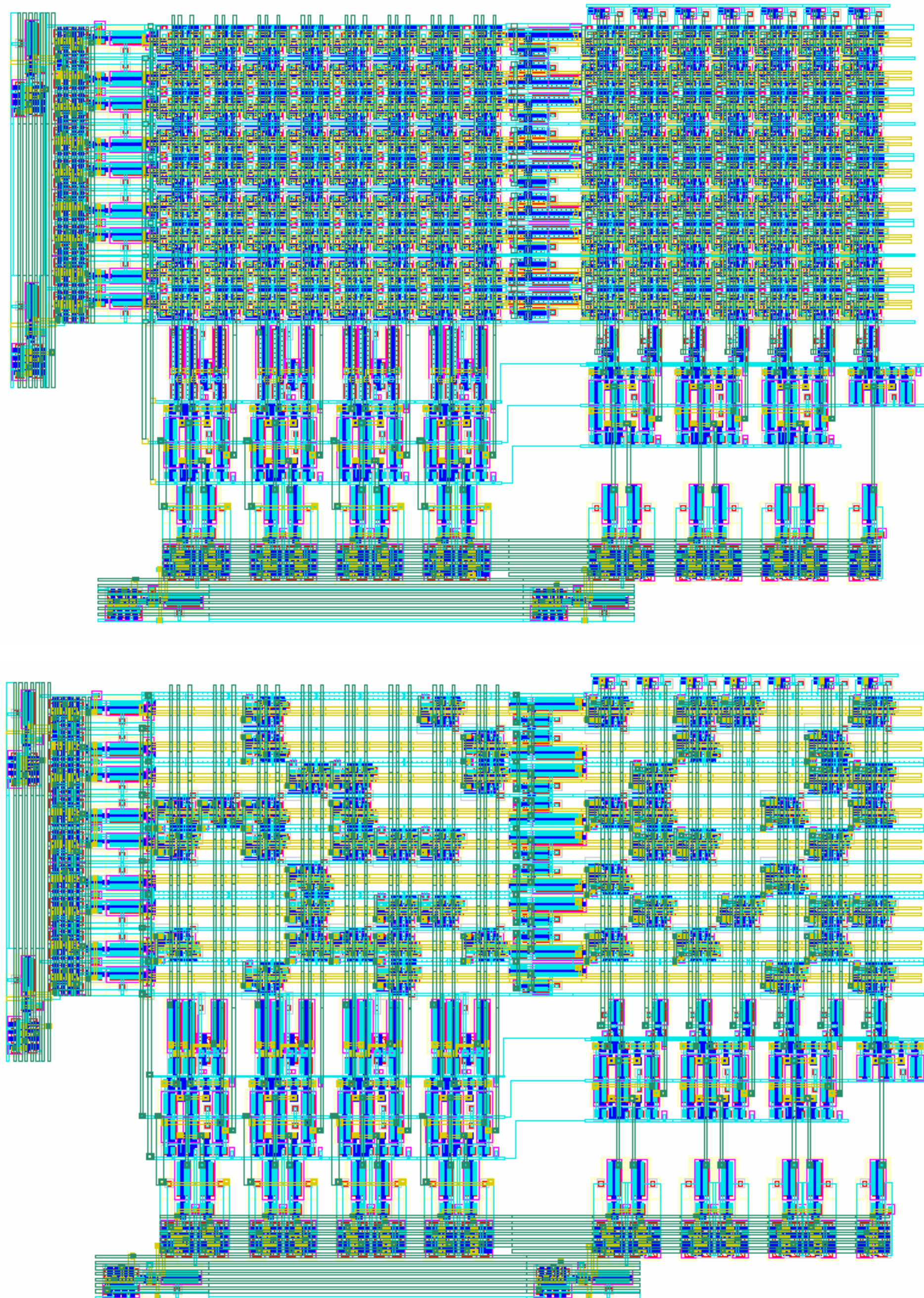

Figure 4. Example of full (top) and sparse (bottom) PLA layouts.

Unfortunately, our problem is to map n circuits onto the array, and n is likely to be greater than two. Our literature search found no efficient algorithms that can find the optimal matching given more than two circuits. One possibility, however, is to perform the Kuhn/Munkres Algorithm on two circuits at a time, and to use a tree structure to combine the mappings. This is shown in Figure 6. In this example we show two different ways of mapping four circuits (N0 – N3) together. Unfortunately, we feel that these methods would lead to poor mappings due to the greedy nature of the formulation.

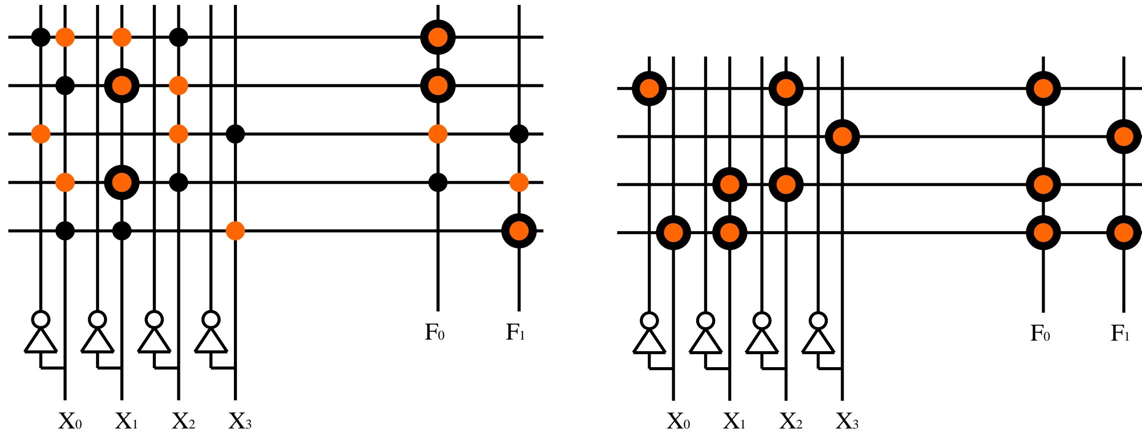

Figure 5. The left PLA shows two circuits mapped randomly, requiring 23 programmable connections. On the right they are mapped intelligently, requiring only 12 connections.

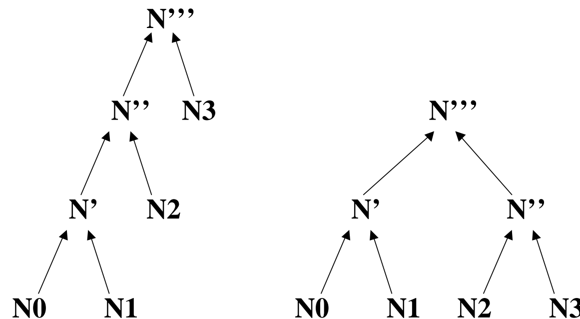

Figure 6. Tree options for using the Kuhn/Munkres algorithm.

For our application, simulated annealing has proven to be very successful at mapping circuits to an array. The algorithm's goal is to minimize the number of programmable connections. We define a basic "move" as being the swapping of two product term rows within a circuit (we will introduce more complicated moves later), and the "cost" of a mapping is the number of programmable bits that it requires. The traditional annealing concept of a bounding box has no notion here, as our metric is not distance dependent, so any product term can swap with any other product term from the same circuit in a given move. For our annealing we use the temperature schedules published in [Betz97].

The development of a cost function requires serious consideration, as it will be the only way in which the annealer can measure circuit placements. The previously mentioned cost function, in which we applied a cost of 1 to locations requiring a programmable bit and a cost of 0 to locations not requiring a bit, initially seems reasonable for our annealer. But looking deeper, the use of a simple 1/0 cost function would actually hide a lot of useful information from the annealer. The degree to which a programmable bit is required (how many circuits are using the array location) is also useful information, as it can tell the annealer how close we are to removing a programmable connection.

Figure 7 displays this notion. In this example, we have one programmable connection used by two circuits and another connection used by five circuits. Both locations require a programmable bit, but it would be much wiser to move to situation A than to situation B, because situation A brings us closer to freeing up a connection.

The cost function that we developed captures this subtlety by adding diminishing costs to each circuit that uses a programmable connection. If only one circuit is using a connection the cost is 1; if two circuits use a connection it costs 1.5; three circuits is 1.75, then 1.875, 1.9375, and so on. Referring again to Figure 7 and using this cost function, moving to A is now a cost of -.45 (a good move) while moving to B is a cost of .19, which is a bad move. The cost function is $COST = 2 - .5^{(x-1)}$, where x is the number of circuits that are using a position. As seen, each additional circuit that loads a position incurs a decreasing cost. For example, going from 7 to 8 is much cheaper than going from 1 to 2.

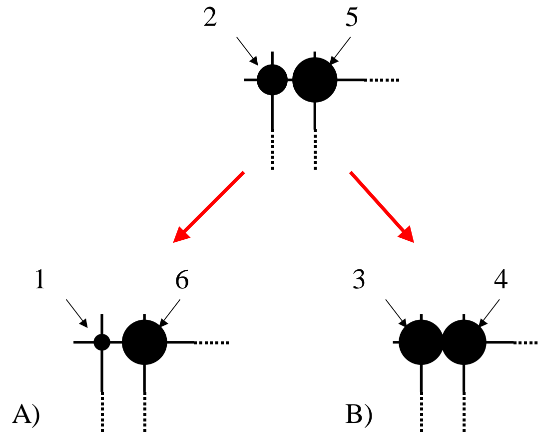

Figure 7. Example of possible annealer moves.

Because PLAs and PALs are structurally different, we need an annealing algorithm that can work on both types of arrays. Additionally, we don't know what hardware might exist on the SoC at the periphery of our arrays. The existence of crossbars at the inputs and outputs to our arrays would allow us to permute the input and output locations between circuit mappings. For example, circuit1 might want the leftmost array input to be in0 while circuit2 wants it to be in3. An external crossbar would allow Totem to accommodate both circuits and decrease the area and delay of the required array, giving the user further benefits.

Thus we are presented with a need for four annealing scenarios: using a PLA with fixed I/O positions, using a PLA with variable I/O positions, using a PAL with fixed I/O positions, and using a PAL with variable I/O positions. The differences between the annealing scenarios are shown in Figure 8. Given a PLA with fixed I/O positions, the only moves that we can make are swaps of product terms within a circuit (A). However, given variable I/O positions (B) we can also make swaps between the inputs of a circuit or between the outputs of a circuit, which will likely provide us with further reduction in programmable connection cost.

The outputs in a PAL put restrictions on where the product terms can be located, so the PAL with fixed I/O positions only allows product terms to be swapped within a given output OR gate (C). In the PAL where we can vary the I/O positions, we actually order the outputs by size (number of product terms) for each circuit such that the larger output gates appear at the bottom. This minimizes the overall sizes of the output OR gates. We are then permitted to make three types of moves: swapping input positions, swapping product term positions, and swapping output positions of equal size, as shown in (D).

The simulated annealing algorithm always chooses random items when attempting to make a move, whether they are product terms, inputs, or outputs. For PLAs with variable I/O positions, 50% of the moves are product term swaps and 50% are I/O swaps, with the ratio of input to output swaps equal to the ratio of inputs to outputs. For PALs with variable I/O positions, 50% of the moves are product terms and 50% are input moves, with the output moves not currently considered because results showed no gain from including them.

When the Architecture Generator is done annealing, it creates a file that completely describes the array. This file is then read by the Layout Generator so that a layout of the array can be created. The Architecture Generator also outputs a configuration file for each circuit so that the circuit can be implemented on the created array.

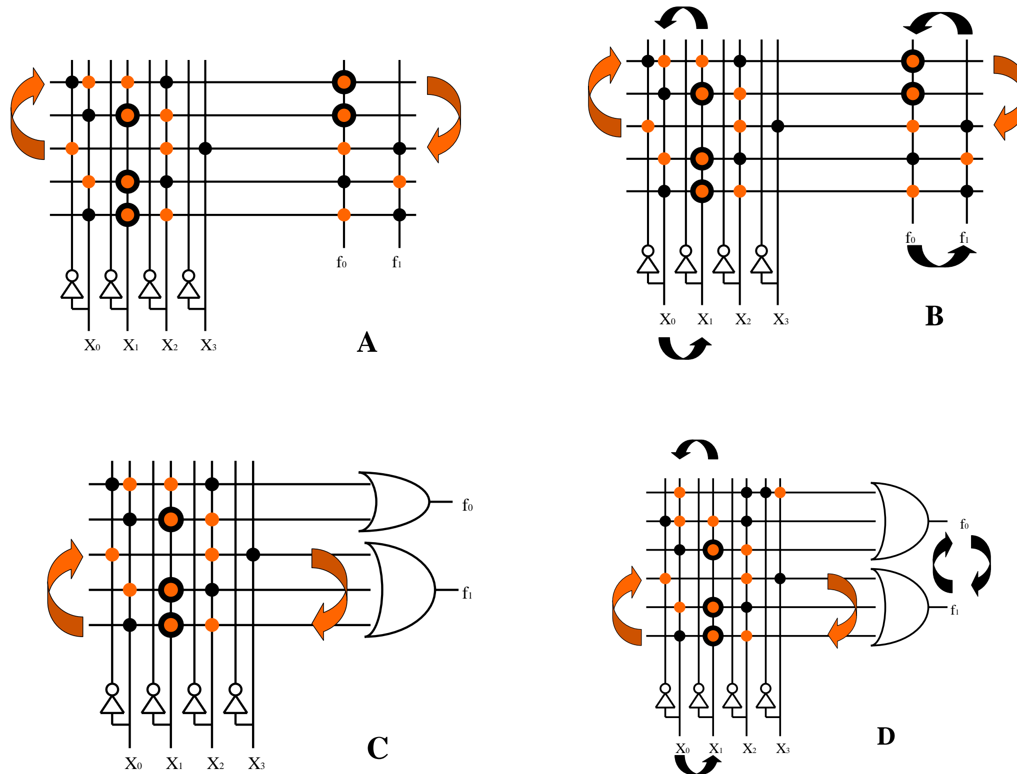

Figure 8. Allowable annealing moves for the four scenarios.

Layout Generator

The Layout Generator is responsible for taking the array description created by the Architecture Generator and turning it into a full layout. It does this by combining instances of pre-made layout cells in order to make a larger design (Figure 9). After the cells are laid down, a compaction tool is optionally run on the design in order to create a more compact layout. The Layout Generator runs in Cadence's LayoutPlus environment, and uses a SKILL routine that was written by Shawn Phillips [Phillips04, Phillips05]. Designs are made in the native TSMC .18µ process.

Figure 4 shows two PLAs that our Layout Generator created: the first PLA displays the compactness of our layouts, while the second array gives an example of the array depopulation that our algorithm achieves. Very small arrays have been shown for clarity, but the arrays we create are often orders of magnitude larger. Pre-made cells exist for every part of a PLA or PAL array, including the decoder logic needed to program the arrays. The Layout Generator simply puts together these pre-made layout pieces as specified by the Architecture Generator, thereby creating a full layout. The input file created by the Architecture Generator contains cell names and layout positions, and the SKILL routine must simply iteratively place the units as it is instructed.

Most existing reconfigurable PLA or PAL based reconfigurable architectures use a pseudo-nMOS or "sense amplifying" design style [XILINX00], as shown in Figure 10. PLAs and PALs are well suited to pseudo-nMOS logic because the array locations need only consist of small pull-down transistors controlled by a programmable bit, and only pull-up transistors are needed at the edges of the arrays. The programmable bits (not shown) are in series with the pull-down transistors in the arrays. These compact pseudo-nMOS structures provide tight layouts and low signal propagation delays for PLAs and PALs, at the cost of increased static power dissipation.

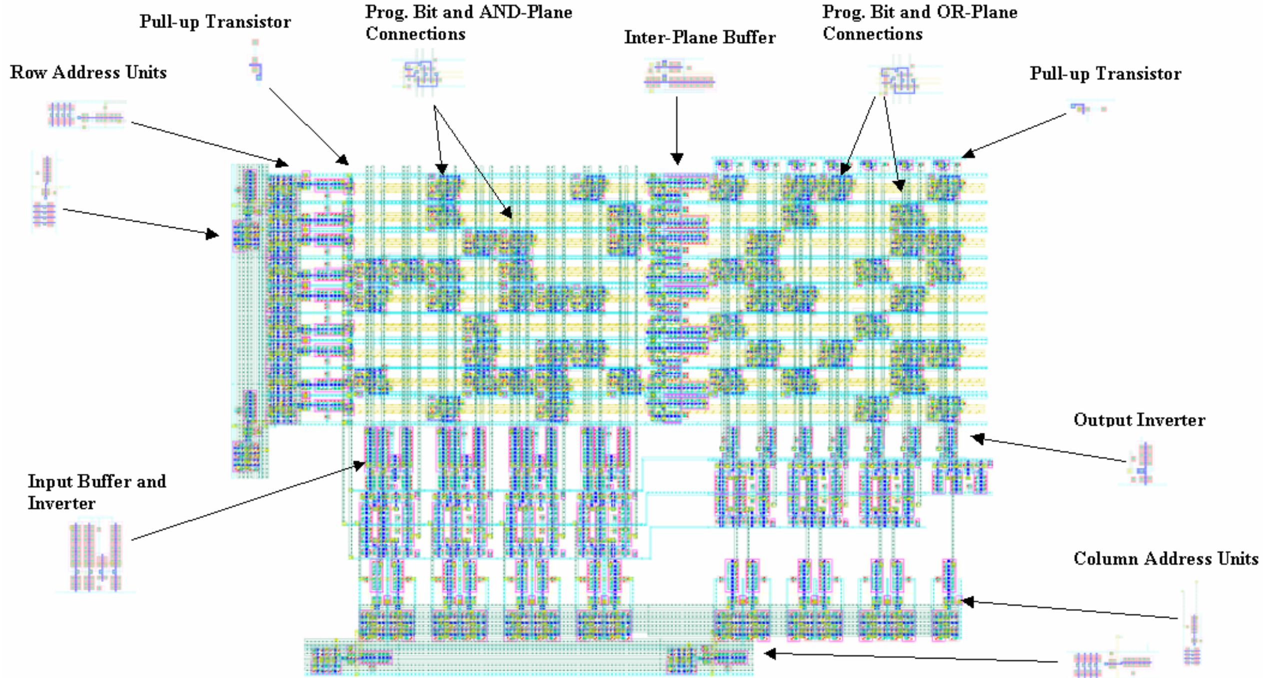

Figure 9. **Sparse PLA layout, and the corresponding basic tiles used by the Architecture Generator.**

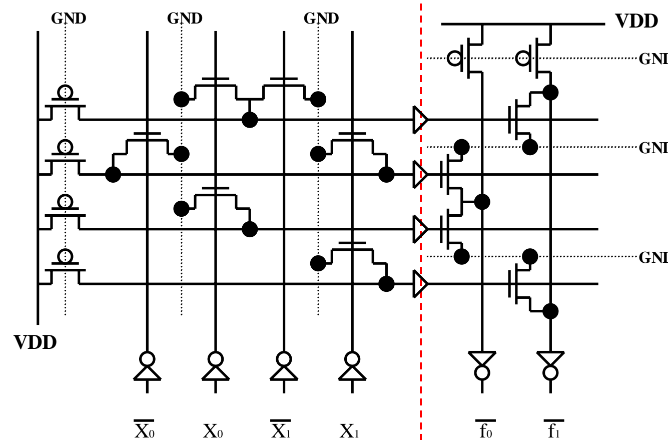

Figure 10. A PLA created using pseudo-nMOS.

We have chosen to implement our PLA and PAL arrays using pseudo-nMOS. In addition to good area and delay performance, pseudo-nMOS arrays are very regular, making their layouts amenable to piecewise, additive construction. This allows the Layout Generator to create efficient PLAs and PALs out of a limited number of layout units. Programmable PLA/PAL layout styles such as static CMOS [Xilinx00] do not scale in a regular fashion, and are not as well suited to automatic layout generation. Non-programmable PLAs often use dynamic logic styles, but dynamic styles add significant complications to the clock generation and distribution in our reconfigurable arrays, and are not suited for a fully automated process. Be reminded, however, that our architecture generation algorithms would work for any of these implementation styles, and that the use of pseudo-nMOS is not critical to the conclusions of this work. Our algorithms can be expected to provide area, delay, and power gains largely independent of the hardware implementation style.

With respect to transistor sizing, we examined the range of PLA and PAL arrays that our tool flow was creating and we sized our transistors in order to provide good performance for what we found to be a representative array size. In

the ideal case, we would be able to size every transistor according to the exact specification of the array that we are creating. This is a prohibitively large design space, however, and would require far too many layout units to be available for Layout Generation, as well as complicating the process of tiling the units into compact full arrays. We therefore are accepting a reasonably small amount of performance degradation in order to retain a flow that is amenable to automation.

Another positive aspect about the use of pseudo-nMOS is that it makes the delay calculations for the arrays quite simple. In CMOS design, an RC model is often used to obtain first-order estimates of propagation delays [Uyemura99]. Using this model, a cutoff transistor is represented by an open circuit, and an active transistor is represented by a closed switch in series with a resistor. The delay is then based on the charging or discharging of some output capacitance in response to a change in input voltage. Figure 11 shows this for an inverter circuit.

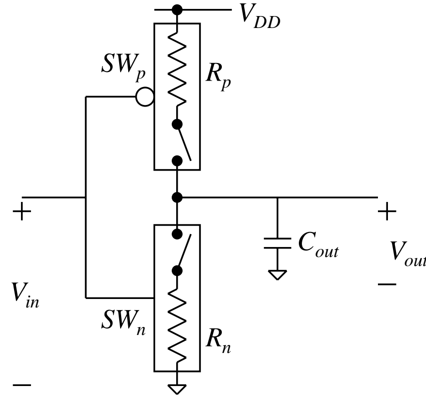

Figure 11. The RC model for an inverter [46]

Using this RC model, the propagation delay through a transistor can be estimated by Equation 1 (derivation found in [Uyemura99]). In this equation, X is a constant, R is the resistance of the transistor, and C_{out} is the output capacitance. Since we will not be changing the sizes of any of the transistors in our arrays, the value of the resistance R is constant for a given transistor and can be absorbed into the constant X. Noting this, the delay through any transistor in our PLA/PAL array is now estimated to be linearly proportional to the output capacitance.

$$t_{prop} = X * R * C_{out} \tag{1}$$

All that remains is to determine the worst-case propagation path in our pseudo-nMOS PLAs. All AND-plane array locations are laid out the same, and will contribute the same capacitance to the corresponding signal path. This is also true of the locations in the OR-plane. We are neglecting some very small capacitive variations that will be caused by differences in location within the AND-plane or OR-plane, but these are small enough to be insignificant. By summing the propagation delays through each of these sections, we obtain the propagation delays through the PLA: one for a rising output and one for a falling output. Greater detail on this derivation can be found in [Holland05].

We consequently used hSpice simulations to acquire timing characteristics for a wide range of our reprogrammable PLA and PAL arrays, varying their sizes in terms of inputs, outputs, and product terms. Applying these results to our RC models of the arrays, we were able to derive a direct relationship between the delay of a signal and the number of programmable locations seen by the signal. The worst-case delay of an array can then be calculated by finding the path that is connected to the largest number of programmable connections. Removing more programmable connections tends to improve the worst-case path through the array, resulting in predictable delay improvements.

Methodology

The use of PLAs and PALs restricts us to the use of .pla format circuits. The first source of circuits is the Espresso suite (the same circuits on which the Espresso algorithm was tested). A second set of circuits comes from the benchmark suite compiled by the Logic Synthesis Workshop of 1993 (LGSynth93). As a whole, these circuits are commonly used in research on programmable logic arrays. The circuits are generally fairly small, but this suits our needs as this work focuses on single arrays, not CPLDs.

Table 1. The circuits used, with their information and groupings.

Group	Circuit	Inputs	Outputs	P.Terms	Connections
1	ti	47	72	213	2573
	xparc	41	73	254	7466
2	b2	16	17	106	1941
	shift	19	16	100	493
	b10	15	11	100	1000
	table5.pla	17	15	158	2501
	misex3c.pla	14	14	197	1561
	table3.pla	14	14	175	2644
3	newcpla1	9	16	38	264
	tms	8	16	30	465
	m2	8	16	47	641
	exp	8	18	59	558
4	seq	41	35	336	6245
	apex1	45	45	206	2842
	apex3	54	50	280	3292

Table 1 gives information on the main circuits that we used for gathering results, including the number of inputs, outputs, product terms, and programmable connections. In sum-of-products notation, each occurrence of a variable is called a literal. For a PAL, the number of literals is equal to the number of programmable connections that are needed in the array. For a PLA one must add the number of literals to the number of product terms in the equations in order to obtain the total number of programmable connections in the array. The connection counts in Table 1 are for PLA representations. The circuits are grouped according to size, as this will be a factor in how well our algorithms perform.

Results

Programmable Connection Reductions

The Architecture Generator uses simulated annealing to reduce the total number of programmable bits that the resultant array will require. While tools like VPR can have annealing results where costs are reduced by orders of magnitude, such large cost improvements are not possible for our annealer because our cost function is very different.

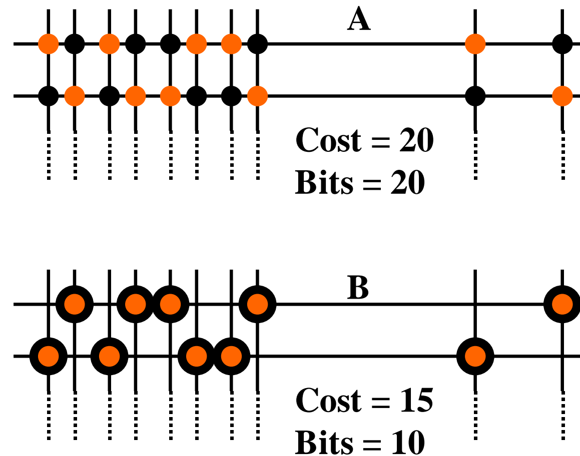

Figure 12. The best possible cost reduction for two circuits is 25%, which is 50% fewer programmable connections.

In actuality, the best cost improvement that our annealer can obtain is bounded, as shown by Figure 12. In part A we have the worst possible placement of the two circuits, and in part B we have the best possible placement. Notice that our cost only goes from 20 to 15, while the total number of actual bits we require goes from 20 to 10. These are actually the bounds of a two circuit anneal: the cost function can never improve more than 25%, and the number of programming bits required can never improve more than 50%. Similarly, with three circuits the best improvement in cost function occurs when three circuits are initially mapped to unique locations, but are all mapped onto the same locations in the final placement. For this case the maximum cost function improvement is 41.7%, while the optimal reduction in the number of programming bits is 66.7%. Similar analysis can be performed on groups of four or more circuits. Since reducing the number of bits is our final objective, the results that we present will show the number of bits required for a mapping rather than the annealing cost.

Determining the minimum possible programming bit cost of a circuit mapping is very difficult. As previously mentioned, there is an $O(n^4)$ exact algorithm for determining the minimum cost of a two circuit mapping, but we have chosen not to implement the exact algorithm because we will often be dealing with more than two circuits.

Because of this, however, we do not have a method for determining the optimal bit cost of an arbitrary mapping. But we can know the optimal mapping of a circuit mapped with itself: it is simply the number of connections in the circuit, as all the connections from the first circuit should map to the same locations as the connections from the second circuit. This can be done with any quantity of the same circuit, and the optimal solution will always remain the same. By doing this we can see how close our annealing algorithms come to an optimal mapping.

Table 2 shows the results obtained from applying this self-mapping test to several circuits using each of the four algorithms on a dual processor Intel Xeon 3.0 GHz machine with 512KB of L2 cache. The “optimal” column shows the number of programmable connections required in the perfect mapping, the “random” column shows the connections required after our algorithms randomize the mappings, and the “achieved” column shows the results obtained by our algorithms. The table shows that when two circuits are mapped with themselves using the PLA-fixed algorithm that the final mapping is always optimal. The PLA-variable algorithm had difficulty with only one circuit, *shift*, which was 15.21% from optimal. Note that for this example the random placement was 93.91% worse than optimal, so our algorithm still showed major gains.

Table 2. Running the algorithms on multiple occurrences of the same circuit. The "Error" column denotes deviation from the optimal result.

Alg.	Netlist	# Netlists	Optimal	Random	Achieved	Error	Runtime
PLA-Fixed	shift	2	493	823	493	0.00%	10
	table5.pla	2	2501	3976	2501	0.00%	17
	newcpla1	2	264	436	264	0.00%	2
	m2	2	641	920	641	0.00%	2
	tms	2	465	647	465	0.00%	1
PLA-Var.	shift	2	493	956	568	15.21%	26
	table5.pla	2	2501	4155	2501	0.00%	55
	newcpla1	2	264	449	264	0.00%	4
	m2	2	641	979	641	0.00%	5
	tms	2	465	684	465	0.00%	2
PAL-Fixed	shift	2	399	637	399	0.00%	3
	table5.pla	2	6312	9901	6312	0.00%	26
	newcpla1	2	250	336	250	0.00%	1
	m2	2	557	798	557	0.00%	2
	tms	2	548	772	548	0.00%	1
PAL-Var.	shift	2	399	737	452	13.28%	12
	table5.pla	2	6312	10352	6312	0.00%	392
	newcpla1	2	250	396	250	0.00%	2
	m2	2	557	859	557	0.00%	9
	tms	2	548	876	548	0.00%	9

For the PAL-fixed algorithm, all of the tests returned an optimal result. The PAL-variable algorithm had a similar result to the PLA-variable algorithm, as the *shift* circuit was only able to get 13.28% from optimal (vs. 84.71% from optimal for a random placement). The near optimal results shown in Table 2 give us confidence that our annealing algorithms should return high quality mappings for arbitrary circuit mappings as well.

Table 3 shows the results of running the PLA-fixed and PLA-variable algorithms on the different circuit groups from Table 1. The reduction in bit cost is the difference in the number of programmable connections needed between a random mapping of the circuits and a mapping performed by the specified algorithm. In the table, all possible 2-circuit mappings were run for each specific group and the results were then averaged. The same was done for all possible 3-circuit mappings, 4-circuit, etc., up to the number of circuits in the group.

There are some interesting things to note from the results in Table 3. Firstly, the PLA-variable algorithm always finds a better final mapping than the PLA-fixed algorithm. This is to be expected, as the permuting of inputs and outputs in the PLA-variable algorithm gives the annealer more freedom. The resulting solution space is much larger for the variable algorithm than the fixed algorithm, and it is intuitive that the annealer would find a better mapping given a larger search space. The practical implications of this are that an SoC designer will acquire better area and delay results from our reconfigurable arrays by supplying external hardware to support input and output permutations – although the area and delay overhead of the crossbar would need to be considered to see if the overall area and delay performance is still improved.

Another thing to notice is that the reduction always increases as the number of circuits being mapped increases. This, too, is as we would expect, as adding more circuits to a mapping would increase the amount of initial disorder, while the final mapping is (hopefully) always close to optimally ordered. Note that this does not say that we end up with fewer connections if we have more circuits, it only says that we reduce a greater number of connections from a random mapping. The trends shown in Table 3 for the PLA algorithms hold for the PAL algorithms as well.

Table 3. Average improvement in programming bits for PLA-Fixed and PLA-Variable algorithms over random placement as a function of circuit count.

Group	# Circuits	PLA-Fixed	PLA-Var.
1	2	3.62%	14.27%
2	2	10.19%	14.52%
	3	16.26%	22.97%
	4	20.20%	28.52%
	5	23.15%	32.49%
	6	25.64%	35.46%
3	2	9.41%	16.44%
	3	14.33%	23.20%
	4	17.79%	29.81%
4	2	3.41%	19.02%
	3	6.01%	28.83%

Another important concept is how well circuits match each other, as higher reductions will be possible when the circuits being mapped have similar sizes or a similar number of connections. With regards to array size, any circuits that are far larger than another circuit will dominate the resulting size of the PLA or PAL array, and we will be left with a large amount of space that is used by only one or few circuits, resulting in poor reduction. If the size of the resulting array is close to the sizes of each circuit being mapped to it then we would expect the array to be well utilized by all circuits. This is shown in Figure 13, which shows the bit reduction vs. array utilization. The array utilization is defined as the percentage of the final PLA's area that is being utilized by both circuits. The PLA-variable algorithm was used for these results, and the circuit pairs were chosen at random from the entire Espresso and LGSynth93 benchmark suites.

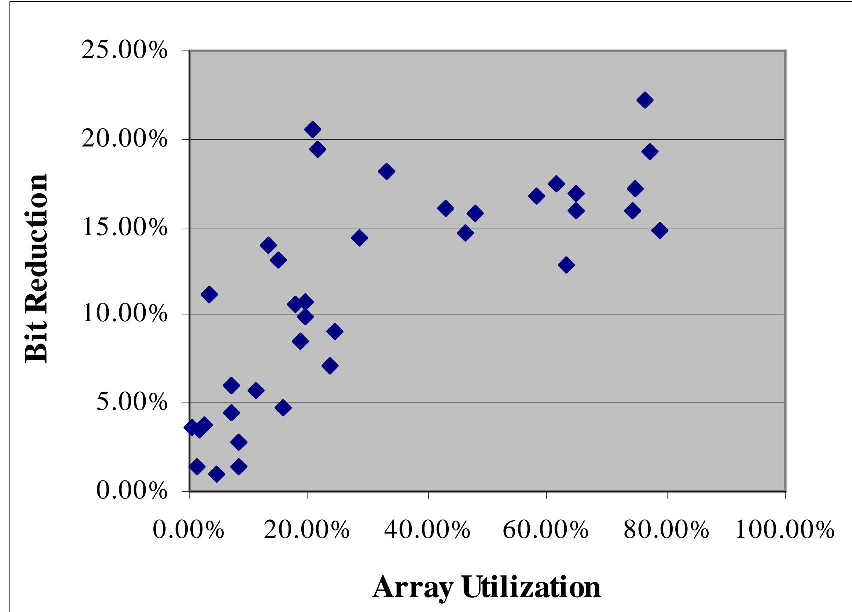

Figure 13. The bit reduction obtained vs. percent array utilization for random circuit pairs.

Mapping circuits with a similar number of connections also results in better reductions. If circuit A has far more connections than circuit B then the total number of connections needed will be dominated by circuit A: even if we map all of the connections from circuit B onto locations used by circuit A we will see a small reduction percentage because B contained so few of the overall connections. It is intuitive that having a similar number of connections in

the circuits being mapped will allow a higher percentage of the overall programmable connections to be removed. This is shown in Figure 14, where we used the PLA-variable algorithm on random circuit pairs from the benchmark suites. A connection count of 80% means that the smaller circuit has 80% of the number of connections that the larger circuit has.

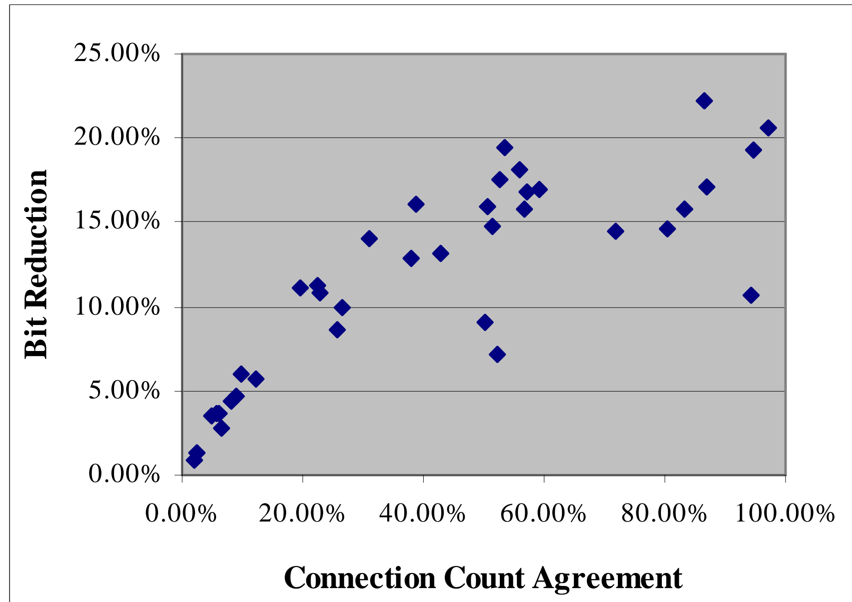

Figure 14. The bit reduction obtained vs. connection count agreement for random circuit pairs.

We used Hspice to develop delay models of the PLA and PAL arrays that we create. Table 4 shows the delay and programmable bit results obtained for several runs of the algorithms, along with average improvements over the full arrays and the random arrays (circuits randomly placed and unneeded connections removed). All algorithms show improvements in delay over the full and random placements.

The PLA-Variable algorithm does better than the PLA-Fixed algorithm with respect to programmable connections because of the input and output permutability that it is allowed. This does not scale directly to delay improvements, however, as the algorithms have no concept of path criticality, and the connections that they remove are often from non-critical paths. Thus, further reduction in connections does not always lead to further reduction in delay.

Table 4. Reductions obtained in number of programmable bits and delay for PLA/PAL algorithms. Count is the number of benchmarks in that test. All columns other than Full are normalized to Full. Improvements are geometric mean improvement across the full set shown.

		PLA Algorithms								PAL Algorithms							
		Programmable Bits				Delay (ps)				Programmable Bits				Delay (ps)			
Netlists	**Count**	**Full**	**Rand.**	**PLA-F**	**PLA-V**	**Full**	**Rand.**	**PLA-F**	**PLA-V**	**Full**	**Rand.**	**PAL-F**	**PAL-V**	**Full**	**Rand.**	**PAL-F**	**PAL-V**
misex3c.pla, table3.pla	2	8274	0.444	0.383	0.362	3620	0.853	0.802	0.802	16856	0.412	0.388	0.375	7641	0.814	0.789	0.786
alu2, f51m	2	2156	0.317	0.258	0.250	1708	0.633	0.537	0.532	2320	0.328	0.304	0.267	1667	0.629	0.579	0.548
ti, xparc	2	42418	0.231	0.223	0.193	5343	0.857	0.854	0.845	156604	0.190	0.188	0.173	18421	0.775	0.764	0.757
b2, shift, b10	3	5830	0.496	0.431	0.389	2329	0.887	0.858	0.832	27512	0.244	0.228	0.214	7780	0.659	0.617	0.616
newcpla1, tms, m2	3	1598	0.626	0.539	0.487	1268	0.975	0.959	0.952	2592	0.415	0.365	0.327	1731	0.952	0.943	0.843
gary, b10, in2, dist	4	6664	0.519	0.399	0.304	2760	0.963	0.946	0.931	13718	0.362	0.308	0.222	4480	0.770	0.689	0.666
newcpla1, tms, m2, exp	4	2124	0.621	0.505	0.450	1459	0.973	0.907	0.941	3132	0.509	0.442	0.359	1966	0.907	0.879	0.778
gary, shift, in2, b2, dist	5	7480	0.590	0.470	0.396	2785	0.979	0.950	0.939	28842	0.317	0.277	0.230	8055	0.692	0.630	0.613
b2, shift, b10, table5.pla, misex3c.pla, table3.pla	6	18321	0.360	0.268	0.247	4015	0.926	0.749	0.777	73948	0.220	0.176	0.145	13746	0.684	0.592	0.549
Improvement vs. Full		**-**	**55.5%**	**63.0%**	**67.2%**	**-**	**11.3%**	**17.1%**	**17.2%**		**68.2%**	**71.5%**	**75.5%**		**24.2%**	**29.0%**	**32.4%**
Improvement vs. Random		**-**	**-**	**16.9%**	**26.3%**	**-**	**-**	**6.6%**	**6.7%**	**-**	**-**	**10.5%**	**22.8%**	**-**	**-**	**6.3%**	**10.8%**

Similar to the performance of the PLA algorithms, the PAL-Variable algorithm performs better than the PAL-Fixed algorithm in terms of both programmable connections and delay. This is again because of the extra flexibility provided by the variable algorithm, in permuting inputs and outputs.

On average, the PLA-Fixed and PLA-Variable algorithms improved upon the delay of a full PLA array by 17.1% and 17.2% respectively. The PAL-Fixed and PAL-Variable algorithms improved upon the delay of a full PAL array by 29.0% and 32.4% respectively. Overall delay improvements of 6.3% to 10.8% were achieved vs. a random placement.

Area Reductions

In the Architecture Generator, unneeded programmable connections are removed from the PLA and PAL arrays that we create. This leaves the arrays full of randomly distributed empty space that a compaction tool should be able to leverage in order to make a smaller, more compact layout. We took several PLA and PAL layouts and applied Cadence's compactor to them, but found that the compactor was unable to reduce the area of any of the arrays (and in fact resulted in a larger area implementation in all cases). For example, applying the compactor to the depopulated PLA from Figure 4 resulted in the layout shown in Figure 15.

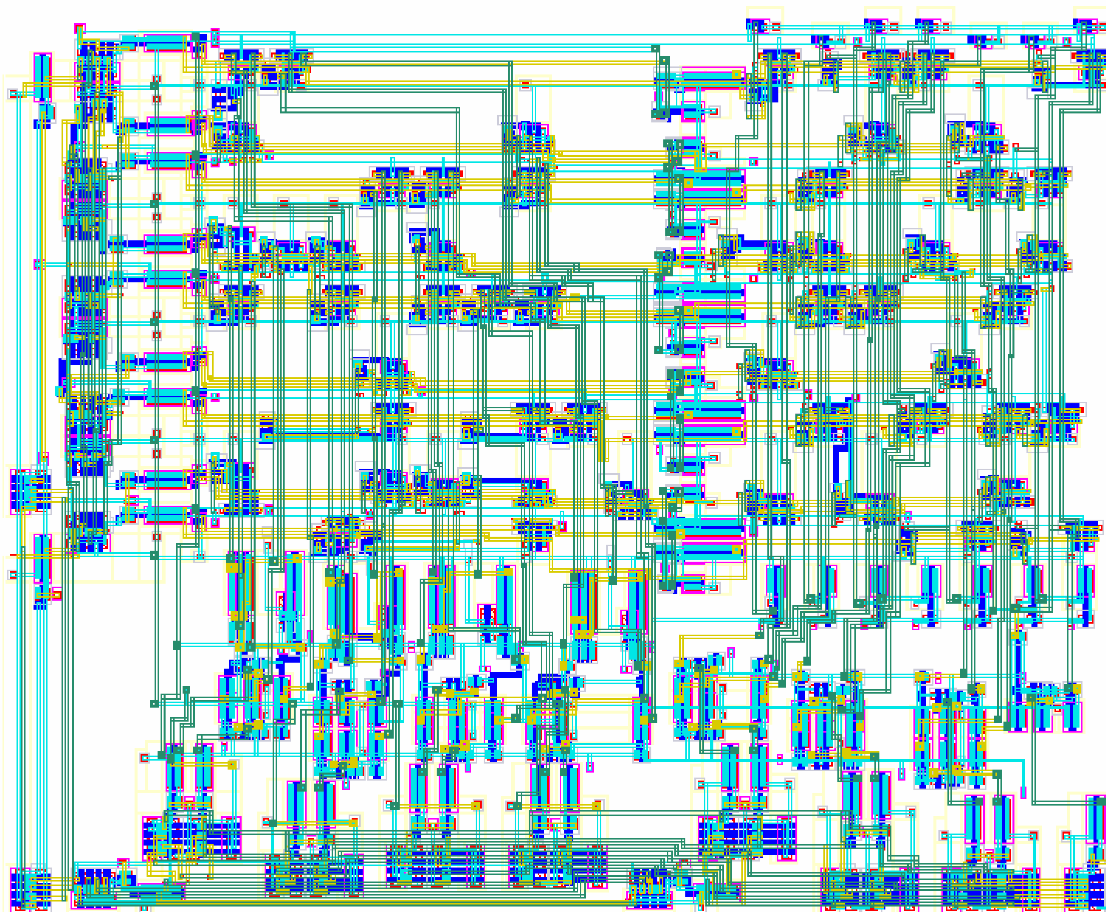

Figure 15. The result of applying a compactor to the sparse PLA from Figure 4. This layout is actually 5% larger than the uncompacted layouts in Figure 4.

The failure of the compactor is due to the high regularity of PLA and PAL arrays, and the cross-connections between all of the units. The compactor iteratively attempts to compact in the vertical and horizontal directions, but

PLAs and PALs have strong vertical and horizontal relationships between array elements which prevent the compactor from making any headway.

Conclusions

In this paper we have presented a complete tool flow for creating domain specific PLAs and PALs for System-on-a-Chip. We have presented an Architecture Generator that, given netlists as an input, maps the netlists onto a PLA or PAL array of minimal size which uses a near-optimal number of programmable connections. Results show that, as the number of netlists being mapped to an array increases, the bit reduction between a random mapping and an intelligent mapping increases.

We also presented a Layout Generator that takes PLA or PAL descriptions from the Architecture Generator and successfully creates optimized layouts by tiling pre-made layout units. Delay improvements of 17% to 32% were achieved over full arrays, but compaction was unable to provide us with any area improvements. The largest improvements were obtained when the PLA/PAL inputs and outputs were permutable, but an SoC designer would need to examine the overhead of input and output crossbars in order to decide whether an area or delay gain would actually be achieved. Our designs utilized a pseudo-nMOS design style, but we emphasize that similar performance improvements should be achieved from other implementation styles, including standard cells and dynamic logic.

If the PLA or PAL array being created will only be used to implement the circuits that were used to design the array, then depopulating the arrays is a good idea as it will provide delay gains over a fully populated array. If other designs are going to be implemented on the array, however, the removal of programmable connections would make it unlikely that a future circuit would successfully map to the array. Thus the removal of programmable connections from a PLA or PAL is not suggested if unknown circuits are going to be mapped to the array in the future.

Generally speaking, realistic PLA and PAL arrays are limited in size due to area and delay considerations, and cannot support very large circuits. When performing large amounts of computation, better performance can be achieved through the use of architectures that utilize a large number of smaller functional units. Thus, techniques such as our work on Totem-CPLD [Holland05, Holland05a] are needed to group domain-specific PLAs and PALs into higher capacity domain-specific CPLDs.

Further Work

In our Layout Generator, the greatest hurdle was the amount of cross-constraints between different elements in the system. These cross-constraints limited the ability of the compactor to optimize the layout. A better alternative would be to design the Layout Generator from the start to use compactable layouts. For example, the drivers at the edges of the layouts were optimized for packing well when surrounding a fully populated array. However, if we packed these units less tightly, perhaps by having two staggered rows at the periphery, then we might achieve better compaction results. Similar techniques have worked well in our sparse crossbar Layout Generator for domain-specific CPLDs [Holland05].

Acknowledgements

This work was supported by grants from NSF. Mark Holland is supported by an NSF Graduate Fellowship. Scott Hauck is supported in part by a Sloan Research Fellowship and an NSF CAREER award. The authors would like to thank Shawn Phillips for help with the Cadence SKILL code used in our Layout Generator.

References

[Asratian98] Asratian, Denley, Häggkvist, *Bipartite Graphs and Their Applications*, Cambridge University Press, UK, 1998.

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

[Brayton84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.

[Compton03] K. Compton, *Architecture Generation of Customized Reconfigurable Hardware*, Ph.D. Thesis, Northwestern University, Dept. of ECE, 2003.

[Cronquist99] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Twentieth Anniversary Conference on Advanced Re-search in VLSI*, 1999.

[Ebeling96] C. Ebeling, D. C. Cronquist, P. Franklin, "RaPiD – Reconfigurable Pipelined Datapath.", *Lecture Notes in Computer Science 1142—Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, R.W. Hartenstein, M. Glesner, Eds. Springer-Verlag, Berlin, Germany, pp. 126-135, 1996.

[Eguro02] K. Eguro, *RaPiD-AES: Developing an Encryption-Specific FPGA Architecture*, Master's Thesis, University of Washington, Dept. of EE, 2002.

[Eguro05] K. Eguro, S. Hauck, "Resource Allocation for Coarse Grain FPGA Development", to appear in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October, 2005.

[Fleisher75] H. Fleisher, L. Maissel, "An Introduction to Array Logic", *IBM Journal of Research and Development*, vol. 19, no. 2, pp. 98-109, March 1975.

[Han05] Y. Han, L. McMurchie, C. Sechen, "A High Performance Programmable Logic Core for SoC Applications", *13th ACM International Symposium on Field-Programamable Gate Arrays*, 2005.

[Hauck06] S. Hauck, K. Compton, K. Eguro, M. Holland, S. Phillips, A. Sharma, "Totem: Domain-Specific Reconfigurable Logic", submitted to *IEEE Transactions on VLSI Systems*.

[Holland05] M. Holland, *Automatic Creation of Product-Term Based Reconfigurable Architectures for System-on-a-Chip*, Ph.D. Thesis, University of Washington, Dept. of EE, 2005.

[Holland05a] M. Holland, S. Hauck, "Automatic Creation of Domain-Specific Reconfigurable CPLDs for SoC", *International Conference on Field Programmable Logic and Applications*, 2005.

[Makarenko86] D. Makarenko, J. Tartar, "A Statistical Analysis of PLA Folding", *IEEE Transactions on Computer-Aided Design*, January, 1986.

[Mo02] F. Mo, R. K. Brayton, "River PLAs: A Regular Circuit Structure", *DAC*, 2002.

[Phillips01] S. Phillips, *Automatic Layout of Domain Specific Reconfigurable Subsystems for System-on-a-Chip,* Master's Thesis, Northwestern University, Dept. of ECE, 2001.

[Phillips04] S. Phillips, *Automating Layout of Reconfigurable Subsystems for Systems-on-a-Chip*, Ph.D. Thesis, University of Washington, Dept. of EE, 2004.

[Phillips05] S. Phillips, S. Hauck, "Automating the Layout of Reconfigurable Subsystems Using Circuit Generators", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[Sharma01] A. Sharma, *Development of a Place and Route Tool for the RaPiD Architecture*, Master's Thesis, University of Washington, Dept. of EE, 2001.

[Sharma05] A. Sharma, *Place and Route Techniques for FPGA Architecture Advancement*, Ph.D. Thesis, University of Washington, Dept. of EE, 2005.

[Uyemura99] J. P. Uyemura, *CMOS Logic Circuit Design*, Kluwer Academic Publishers, Boston, 1999.

[Xilinx00] Xilinx, Inc, "Fast Zero Power (FZP™) Technology", <http://www.xilinx.com/products/coolpld/wp119.pdf>, 2000.

[Yan03] A. Yan, S. Wilton, "Product Term Embedded Synthesizable Logic Cores", *IEEE Conference on Field-Programmable Technology*, 2003.

[Wilton05] S.J.E. Wilton, N. Kafafi, J. Wu, K. Bozman, V. Aken'Ova, R. Saleh, "Design Considerations for Soft Embedded Programmable Logic Cores", *IEEE Journal of Solid-State Circuits, vol. 40, no. 2*, Feb 2005, pp. 485-497.