Automatic Creation of Domain-Specific Reconfigurable CPLDs for SoC

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

Abstract

Many System-on-a-Chip devices would benefit from the inclusion of reprogrammable logic on the silicon die, as it can add general computing ability, provide run-time reconfigurability, or even be used for post-fabrication modifications. Also, by catering the logic to the SoC domain, additional area and delay gains can be achieved over current, more general reconfigurable fabrics. This paper presents tools that automate the creation of domain-specific CPLDs for SoC, including an Architecture Generator for finding appropriate architectures and a Layout Generator for creating efficient layouts. By tailoring CPLDs to the domains that they are supporting, we provide results that beat representative fixed architectures by 4.1x to 9.5x on average in terms of area-delay product.

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 several discrete components individually fabricated and then integrated together on a board, is becoming obsolete. As gate count continues to increase (currently chips can hold hundreds of millions of wirable gates), distinct VLSI components can now be incorporated onto the same piece of silicon and this "System-on-a-Chip" methodology is becoming more prevalent.

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 a library of 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 cost-free upgradability, conformity to different but similar protocols, coprocessing hardware, and uncommitted testing resources. Additionally, the paradigm of IP reuse makes it even easier to incorporate reconfigurable logic into a SoC device as a pre-made IP core.

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 either a single, central

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

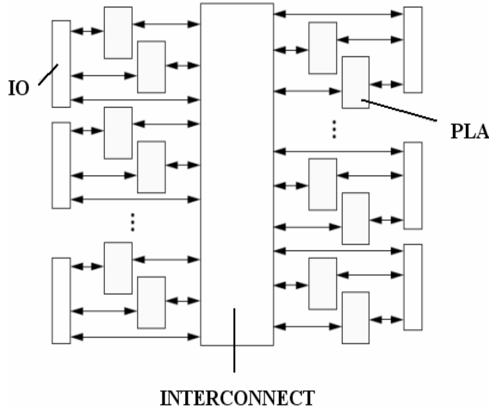

Figure 1. A CPLD with central interconnect

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.

Totem-CPLD will tailor CPLDs to a specific domain, thereby removing some of these performance penalties. Specifically, by altering the sizes of the functional units (PLAs) in terms of inputs, product terms, and outputs, CPLD architectures can be created that perform better than "typical" CPLD architectures for a specified domain.

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 [1]. 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 [2].

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.

In another 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 [3]. 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 both high circuit regularity and predictable area and delay formulation. 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. The interconnect between their PLA units, however, is fairly sparse, and it is confined by their need for directionality. An architecture with more robust routing would undoubtedly be able to support a wider range of designs, and therefore a wider range of domains.

As a precursor to Totem-CPLD, we performed some work in which we explored the feasibility of making domain-specific reconfigurable PLAs and PALs [4]. In this work we wrote an architecture generation tool that mapped domains of circuits to either a PLA or a PAL in such a way that it could remove some of the unneeded programmable connections in the arrays. By doing this intelligently, we were able to remove 60%-70% of the programmable connections in the arrays, which provided delay gains of 15% to 30%. Depopulating the arrays in a PLA is very restrictive to future mappings, however, so we chose not to use PLA depopulation in Totem-CPLD.

In order to create CPLD architectures, we need to have the appropriate tech-mapping tool. We will be using a tool called 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. Area/Delay tradeoffs are also available, but we will be running PLAmap in purely delay-driven mode. PLAmap is run by providing it with a PLA size (inputs, product terms, outputs) and a circuit (in .blif format) to be mapped, after which PLAmap returns the number of PLAs required for

the mapping, the depth of the mapping, and saturation statistics for the PLAs.

3. Approach – Tool Flow

The tool flow for Totem-CPLD is shown in Figure 2.

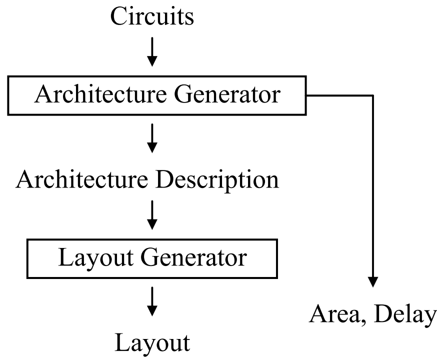

Figure 2. Totem-CPLD Tool Flow

To begin the process, the customer will provide us with a domain specification that contains the circuits that need to be supported. These circuits will be fed into an Architecture Generator, which will find a CPLD architecture that provides good results for the selected domain, outputting the architecture description and the area-delay product of the implementation. The architecture description is then sent to a Layout Generator which creates a full VLSI layout of the specified CPLD architecture.

3.1 Architecture Generator

The Architecture Generator is responsible for reading in multiple circuits and finding a CPLD architecture that supports the circuits efficiently. Search algorithms are used to make calls to PLAmap, after which the results are analyzed according to area and delay models that we have developed. The algorithms then make a decision to either make further calls to PLAmap, or to exit and use the best CPLD architecture that has been found. This is shown graphically in Figure 3. PLAmap assumes full connectivity between the PLAs, and the Architecture Generator accommodates this by connecting all the PLAs through a full crossbar. Future work will consider other interconnect styles.

The Architecture Generator is responsible for finding a PLA size that leads to an efficient CPLD architecture for the given domain. PLAs are specified by their number of inputs (IN), product terms (PT), and outputs (OUT), so the search space for the Architecture Generator is three-dimensional. Searching the entire 3-D space is not viable, as calls to PLAmap can take on the order of hours for larger circuits, and our ultimate goal is to find a suitable CPLD architecture in a matter of hours or days. Also, for each PLA architecture that we test a domain on, PLAmap must be called once for each circuit in the domain. Clearly, minimizing the number of PLAmap calls is important to our runtime. Otherwise effective algorithms such as simulated annealing and particle swarm are far too costly for our scenario, and smart algorithms will be required if we wish to acquire good result in a 3-D search space using relatively few data points.

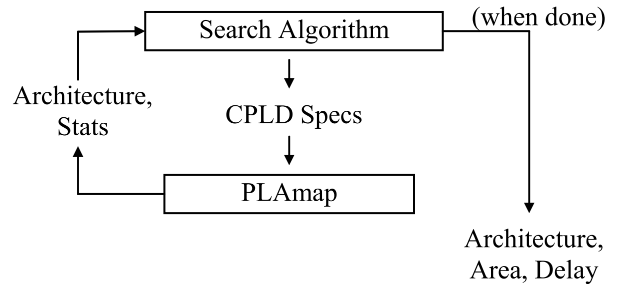

Figure 3. Architecture Generator

In order to gain some intuition about the search space, we ran five random LGSynth93 circuits through PLAmap and acquired a coarse representation of the 3-D space for each circuit. The first thing that we noticed by looking at these results was that the three PLA variables are related, as can be expected. More specifically, the best results are obtained when the number of product terms was roughly between 1x and 3x the number of inputs. Similarly, good results were obtained when the number of outputs was between .25x and .6x the number of inputs. To generalize, a ratio of 1 to 2 to .5 for the IN, PT, and OUT variables respectively was found to consistently provide good results. Within the scope of these ratios, CPLDs with 10-20-5 PLAs were shown to provide generally good results.

Another observation we made was that the best results tended to be grouped around PLAs with a specific input count, as long as the product term and output counts were reasonable. Similarly, good results seemed to be grouped around favorable input/output combinations, such that the number of product terms played a slightly lesser role in affecting the final results. This led us to the concept of breaking the 3-D space into three 1-D spaces, which can be searched sequentially and in much less time. More specifically, our algorithms will start by searching for a good input size (while keeping a 1x-2x-.5x IN-PT-OUT relationship), will next search for a good output size, and will finish by searching for a good product term size.

Lastly, we observed that the 3-D search space generally tends to be well behaved. Results tend to get better as you approach the optimal point, and to get worse as you go away from the optimal point. But far from being

perfectly behaved, there are many small perturbations in the smoothness that lead to local optima. These local optima appear both near and far away from the global optima, so measures will need to be taken to avoid being caught in such local optima.

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

3.1.1 Search Algorithms

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

3.1.1.1 Hill Descent

The Hill Descent algorithm is the first algorithm that we developed, and the most basic. The algorithm starts by running PLAmap on architectures with 10-20-5 and 12-24-6 PLAs. Whichever result is better, we continue to take results in that direction (i.e. smaller or larger PLAs), keeping the 1x-2x-.5x ratio intact and performing steps of IN = +/-2 (like descending a hill). We continue until a local optima is reached, as determined by the first result that does not improve upon the last result. At this point we explore the PLAs with IN = +/-1 of the current local optima. The best result is noted, and the input value is permanently locked at this value, thus ending the input step. This is shown graphically in Figure 4.

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

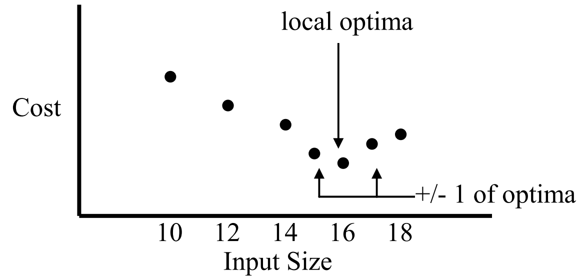

Figure 4. Hill Descent Algorithm

The Hill Descent algorithm is decidedly greedy, as it always moves in the direction of initial improvement. It also has no method for avoiding local minima, as any minima will stop the current step. Therefore it is somewhat difficult for this algorithm to find architectures that vary much in size from the 10-20-5 PLA starting point, but decent results are still obtained due to the fact that the 10-20-5 starting point is a relatively good point in the 3-D search space.

3.1.1.2 Successive Refinement

The successive refinement algorithm is intended to slowly disregard the most unsuitable PLA architectures, thereby ultimately deciding upon a good architecture by process of elimination. In the input optimization step (Figure 5), data points are initially taken for PLAs with input counts ranging from 4 (lower bound) to 28 (upper bound) with a step size of 8. So initially, 4-8-2, 12-24-6, 20-40-10, and 28-56-14 PLAs are run (part a in Figure 5). The left and right edges are then examined, regions that do not contain local/global minima are trimmed from consideration (shaded region of part a), and the bounds are adjusted accordingly. The step size is then halved, and the above process is repeated (part b). This occurs until we have performed an exploration with a step size of 1 (part d).

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

The Successive Refinement algorithm is greedy in the way it trims sub-optimal PLAs from the edges of its consideration. It does not trim sub-optimal regions from the middle, however, and can therefore require more PLAmap runs than is absolutely necessary. Typically, several local optima get explored at maximum granularity, providing a good survey of the areas around the minima at a small cost to runtime.

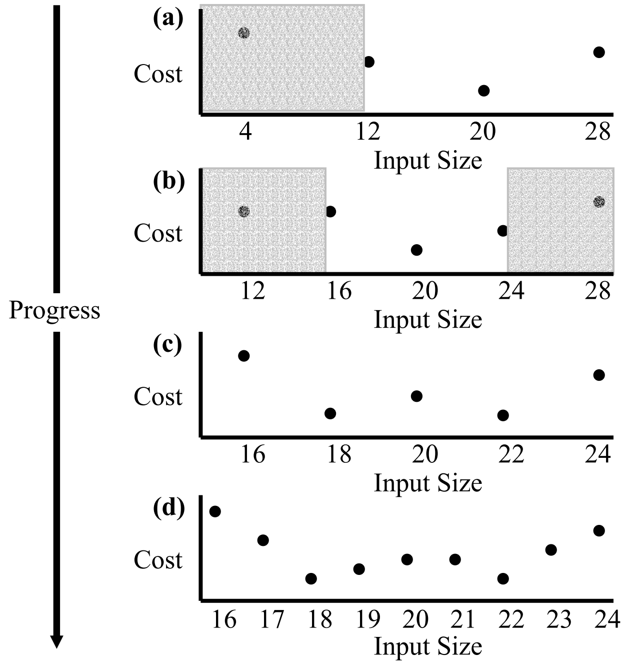

Figure 5. Input Optimization Step of the Successive Refinement algorithm. At each iteration, shaded regions are trimmed and the step size halved.

3.1.1.3 Choose N Regions

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

Like the Successive Refinement algorithm, the input optimization step of the Choose N Regions algorithm is initiated by taking data points for PLAs with inputs ranging from 4 to 28, but now with a step size of 4. N regions are then chosen for further exploration (N=2 was experimentally found to be a good value). A region consists of a data point on the left side, a data point on the right side, and the unexplored space between them. The N best regions are the regions with the best primary result, where the primary result is: min(left result, right result). For ties, the region with the best secondary result max(left, right) is taken (see Figure 6). These N regions are retained, the step size is halved, and we iterate on the new regions. This continues until N regions have been explored with a step size of 1.

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

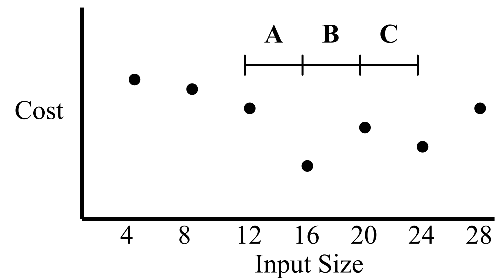

Figure 6. Choose N Regions algorithm. Region B is the best, because it has the best primary point (along with A) and the best secondary point. Region A is 2nd best, region C is 3rd best.

The Choose N Regions algorithm has the advantage of retaining, at all steps, N regions of consideration. This allows the algorithm to hone into multiple local minima, as well as throw out old minima that get replaced by new, better results.

3.1.1.4 Run M Points

The Run M Points 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.

Again, the input optimization step starts by taking data points for PLAs with inputs ranging from 4 to 28, with a step size of 4. Next, 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 shown in Figure 7. The process 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.

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, and repeat the Run M Points algorithm mentioned above. The product term step repeats this process, with product term values ranging from 10 to 90 and a step size of 8 (so possible step sizes are 8, 4, 2, and 1 now).

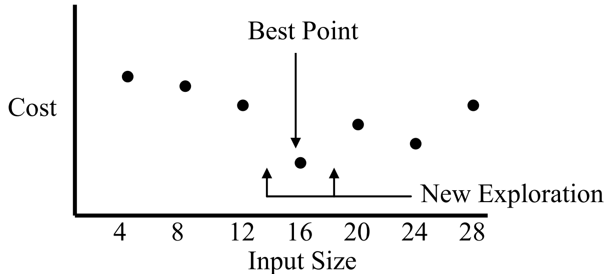

Figure 7. Run M Points algorithm. The best point is always chosen, and the regions to it's left and right are explored.

Because we are exploring to either side of the best result, the range of 10 to 90 is not strictly enforced for the product term step, as exploration around 10 or 90 would take data points on both sides of the given point. This concept is true for all steps in the Run M Points algorithm. Also note that the input and output steps have the same interval size and step size, while the product term step has a larger interval and larger step size. To account for this, the product term step is allowed to run slightly more than M runs so that it can closely explore as many regions as the input and output steps.

While the Choose N Regions algorithm explores N possible optima in parallel, the Run M Points algorithm can be seen as exploring the optima one at a time. It will explore the best optima until it runs out of granularity, then will turn to the second best optima, and so on. In this way it also considers multiple possible optima, as determined by the value chosen for M.

3.1.2 Algorithm Add-Ons

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

3.1.2.1 Radial Search

As mentioned before, the 3-D search space for this problem is relatively well shaped, but not perfectly so. There are many local optima that might prevent the above algorithms from finding the global optima. One way to look outside of these local optima is to search the 3-D space within some radius of the current optima. So for a radius R search around an X-Y-Z architecture, we would vary IN from X-R to X+R, PT from Y-R to Y+R, and OUT from Z-R to Z+R, testing all architectures in this 3-D subspace.

We have a strict time constraint on the runtime of the Architecture Generator, so performing the (2R+1)^3 extra PLAmap runs necessary for a radius = R search is not feasible as part of our finalized tool flow. Given looser time constraints and moderately sized circuits, however, small radial searches are not out of the question. Another reason to run radial searches is that it can search a small (but good) part of the 3-D search space exhaustively, and give an idea of how well the basic algorithms are performing. For this reason, we have performed radial searches of up to R = 3 at the conclusions of the basic algorithms listed above.

3.1.2.2 Algorithm Iteration

The Architecture Generator algorithms all assume that the PLAs should be in a 1x-2x-.5x relationship in terms of inputs, product terms, and outputs. This is just a rough guideline, however, and is very rarely the optimal ratio for a given domain. Thus, an interesting idea is to run the basic algorithms (with or without a radial search) and then look at the resulting PLA to obtain a new IN-PT-OUT relationship. A second iteration of the algorithm can be run with this new IN-PT-OUT relationship, exploring the 3-D search space using a relationship that the domain has already been shown to prefer. For example, if the first iteration chose a 10-30-8 architecture, then the IN-PT-OUT relationship for the next iteration would be 1x-3x-.8x. A second iteration has been carried out for all of the algorithms on each domain.

3.1.2.3 Small PLA Inflexibility

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

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

applicable cases.

3.2 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 [6]. The layouts are designed in the TSMC .18-micron process.

Figure 8 displays a small CPLD that was created using the Layout Generator. For clarity's sake, the encoding logic required for programming the RAM bits is not shown, but would appear along the left and bottom of the laid out CPLD. Pre-made cells exist for every part of the PLA and crossbar units, including the RAM encoding logic. The Layout Generator simply puts together the pre-made layout pieces as specified by the architecture description that the Architecture Generator provides. The PLAs are implemented in pseudo-nmos in order to provide a compact layout at the cost of power dissipation (power dissipation is considered in the future work section).

4. Methodology

The use of PLAmap restricts us to the use of blif format circuits. The LGSynth93 suite has a large number of blif circuits, but their functions are unknown, so they are difficult to group into domains. For preliminary testing purposes, we did choose to create three domains out of the LGSynth93 suite: a small domain (43-106 gates), a medium domain (246-457 gates), and a large domain (2246-3606 gates). Each of these domains has six circuits.

In order to create more realistic domains, we had to find other means of obtaining circuits, as well as determine a method of getting non-blif circuits into the blif format. Circuits are most easily obtained in HDL (Verilog and VHDL) formats, so the problem became transforming HDLs into blif. A second constraint composed by PLAmap is that the blif circuits must be 2-bounded: i.e. they must be composed of gates with no more than two inputs.

After much effort, we developed a method for translating files from HDL format into blif. The process is shown in Figure 9. The HDL files are initially loaded into Altera's Quartus 2 program, which is able to dump the designs

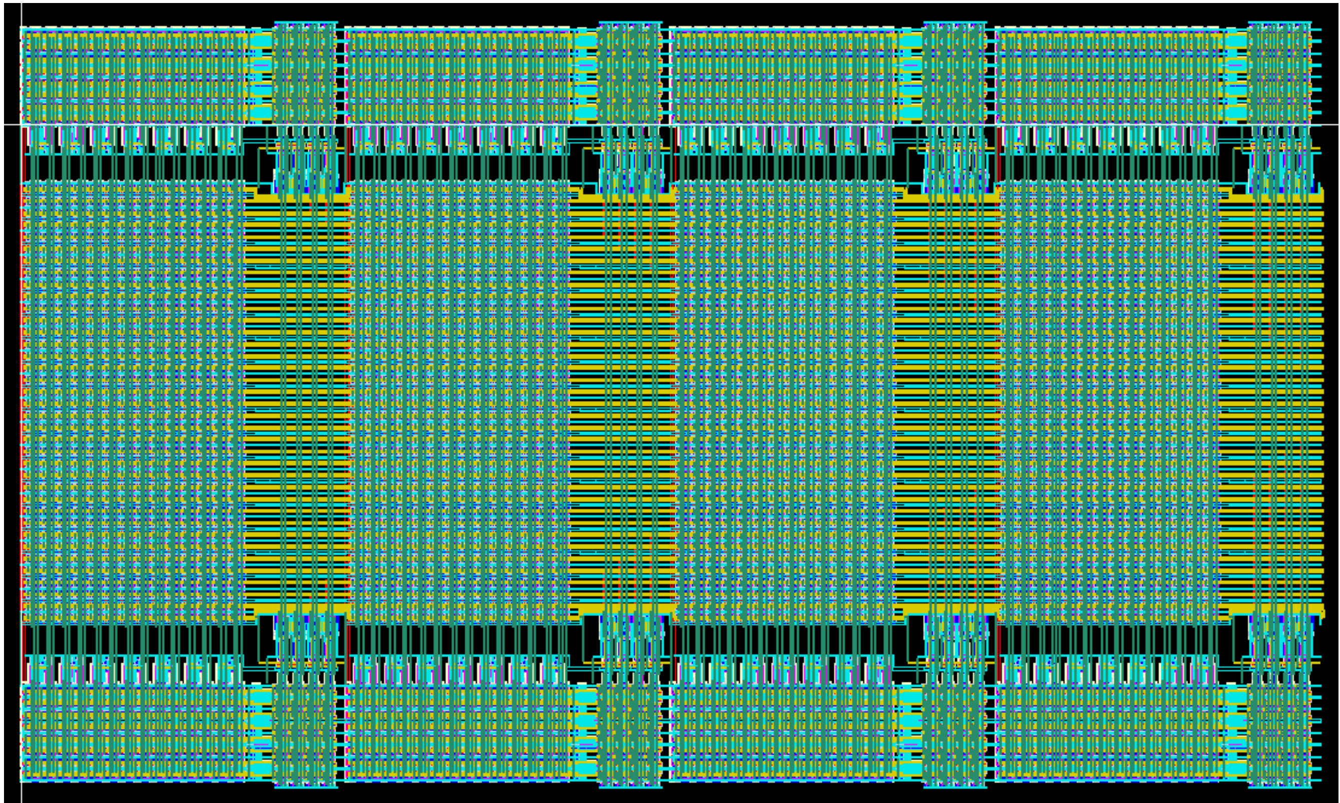

Figure 8. A small CPLD with eight 7-8-4 PLAs (top and bottom) and a crossbar.

into a blif file (developers at Altera were very helpful in providing this hidden functionality to us). SIS is then used to transform the blif file into a 2-bounded network, which PLAmap will be able to accept.

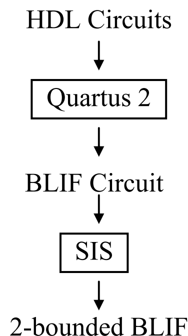

Figure 9. File transformation process.

We created five domains of circuits. Two of them, the combinational and sequential domains, consist of files gathered from the LGSynth93 Benchmark Suite. The actual functions of these files are unknown, but they are grouped for their combinational or sequential characteristics. These circuits were acquired in blif format, and simply needed to be 2-bounded by way of SIS.

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, from Altera software developers, as Quartus 2 megafunctions, and from open source floating point libraries. All of these files were provided in HDL format, and went through the entire flow shown in figure 9 in order to be used in our work.

The floating point domain consists of several different units, including floating point multipliers, adders, and dividers. Also included is an LNS divider, an LNS multiplier, LNS and floating point square root calculators, and a floating point to fixed-point format converter.

The arithmetic domain consists of several different implementations of multipliers and dividers, as well as a square root calculator and an adder/subtractor. The encryption domain consists of the Cast, Crypton05, Magenta, Mars, Rijndael, and Twofish encryption algorithms (all sans memories), all of which were recent competitors to become the advanced encryption standard [7]. The domains are all summarized in Table 1.

The domain-specific CPLD architectures that we create are going to be compared to results obtained by implementing all of the domains in fixed CPLD architectures. All results are in terms of area-delay product, and are calculated using the delay and area models that we developed for the actual architectures that

Table 1. The domains used in our work.

Comb.	IN	OUT	GATES	REGs
C1355	41	32	542	0
C17	5	2	8	0
C1908	33	25	460	0
C3540	50	22	1045	0
C432	36	7	175	0
C499	41	32	406	0
C5315	178	123	1978	0
C6288	32	32	2350	0
C880	60	26	352	0
c8	28	18	219	0
cm138a	6	8	26	0
cm150a	21	1	46	0
cm151a	12	2	23	0
cm152a	11	1	31	0
cm162a	14	5	42	0
cm163a	16	5	42	0
cm42a	4	10	22	0
cm82a	5	3	22	0
cm85a	11	3	44	0
cmb	16	4	47	0
Seq.	**IN**	**OUT**	**GATES**	**REGs**
s1196	15	14	481	18
s1238	15	14	552	18
s208.1	11	1	77	8
s344	10	11	122	15
s349	10	11	125	15
s382	4	6	150	21
s400	4	6	160	21
s420.1	19	1	165	16
s444	4	6	173	21
s526	4	6	241	21
s526n	4	6	272	21
s838.1	35	1	341	32
s953	17	23	369	29
Encryption	**IN**	**OUT**	**GATES**	**REGs**
cast	298	166	12934	301
crypton05	388	260	6980	261
magenta	452	387	4876	713
mars	389	172	23637	1071
rijndael	388	132	11618	261
twofish	261	196	14784	517
Arithmetic	**IN**	**OUT**	**GATES**	**REGs**
MultAddShift	32	32	4392	0
MultBooth2	34	33	759	37
MultBooth3	34	33	2238	36
MultSeq	34	33	529	53
serial_divide_uu	28	17	645	53
Adder	32	17	379	0
SQRT	32	33	3302	0
AddSub	33	16	370	0
Mult	32	32	4361	0
Div	32	32	3283	0
AbsValue	32	32	302	0
FP	**IN**	**OUT**	**GATES**	**REGs**
FPMult	65	35	9895	698
fpadd	44	22	2213	0
fpmul	44	22	3687	0
fpdiv	44	22	5505	0
fpsqrt	22	22	2675	0
lnsdiv	44	22	340	0
lnsmul	44	22	331	0
lnssqrt	21	22	24	0
float2fix	36	34	704	142
fp_mul	67	57	8500	174
fp_sub	67	35	1786	199
fp_add	67	35	1786	199

we create and lay out in the TSMC .18-micron process.

We have chosen three different fixed architectures to compare our results to, all of which will use a full crossbar to connect the PLA units in order to conform to our area and delay models. A 1991 analysis of PLA sizings in reprogrammable architectures by Kouloheris and El Gamal [8] showed that PLAs with 8-10 inputs, 12-13 product terms, and 3-4 outputs provide the best area performance for CPLDs. To model this, the first architecture we will compare to uses 10-12-4 PLAs.

Secondly, our own initial analysis of running several LGSynth93 circuits through PLAmap showed that 10-20-5 PLAs tended to show good performance. We will use this as our second fixed architecture.

Third, we will compare against a XILINX CoolRunner-like architecture. The XILINX CoolRunner uses 36-48-16 PLAs for its functional units, so we will compare our domain-specific results to a fixed architecture that uses these PLAs.

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

5. Results

Of the four Architecture Generator algorithms, two of them, the Choose N Regions and Run M Points algorithms, have a user-supplied variable. In the Choose N Regions algorithm we must choose how many regions get explored each iteration, while in the Run M Points algorithm we need to determine how many overall PLAmap runs get executed in each of the three steps.

For the choose N Regions algorithm, the input step and output step both break the 1-D search space into 6 regions, meaning that N can be no larger than 6. We decided to vary N from 1 to 4 when evaluating the algorithm, as this would result in a reasonable number total PLAmap runs. Results for running the Choose N Regions algorithm on the small, medium, and large LGSynth93 domains are shown in Figure 10. The figure shows that gains are achieved by increasing N from 1 to 2, but that further gains are not achieved when setting N to 3 or 4. From this, we determined that N = 2 is a good value to use.

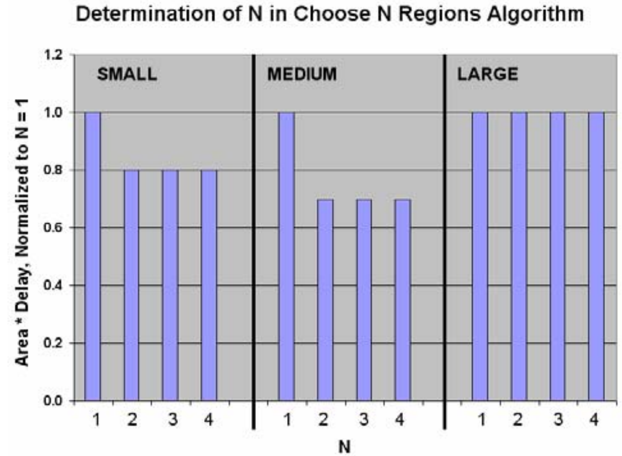

Figure 10. Determination of N in Choose N Regions algorithm.

For the Run M Points algorithm, we must determine how many total PLAmap runs are performed for each step of the algorithm. Setting M to 25 would exhaustively search the 1-D input and output spaces, while setting M to 7 would only search the top level without making any interesting descents. We decided that setting M to 10, 15, and 20 would provide a good span of results in a reasonable number of PLAmap runs. The results of running this test on the LGSynth93 domains are shown in Figure 11. The graph shows that M = 15 always outperformed M = 10, and the going up to M = 20 only provided further gains in the small domain, and those gains were very small. From these results, we chose to use M = 15 in future runs of the Run M Points algorithm.

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

	Algorithms												Fixed Architectures		
	Hill Descent			Succ. Refinement			Choose N Regions			Run M Points			10-12-4	10-20-5	36-48-16
Domain	**Arch**	**A*D**	**Runs**	**Arch**	**A*D**	**Runs**	**Arch**	**A*D**	**Runs**	**Arch**	**A*D**	**Runs**	**A*D**	**A*D**	**A*D**
Combinational	12-25-4	2.38	13	12-71-4	1.00	91	12-71-4	1.00	40	5-16-2	2.34	49	10.72	4.14	10.91
Sequential	12-23-6	1.00	11	12-23-6	1.00	79	12-23-6	1.00	38	12-23-6	1.00	50	2.08	1.65	1.54
Floating Point	9-28-5	2.44	15	4-8-1	2.75	88	10-24-2	1.00	67	10-24-2	1.00	85	10.80	6.05	24.17
Arithmetic	10-20-2	1.00	14	10-20-2	1.00	57	10-20-2	1.00	66	10-20-2	1.00	85	37.68	13.58	36.63
Encryption	10-23-3	1.00	13	10-46-3	1.00	63	10-46-3	1.00	39	10-46-3	1.00	51	2.91	1.99	5.26
G Mean / Avg		**1.42**	**13.2**		**1.22**	**75.6**		**1.00**	**50.0**		**1.19**	**64.0**	**7.66**	**4.07**	**9.53**

Now that we have chosen the user-defined variables in the Choose N Regions and Run M Points algorithms, all of the algorithms are fully specified. Next, we took our five main domains and ran each of the four algorithms on each domain. Additionally, we mapped the circuits of each domain to the fixed architectures that we described earlier. These results are shown in Table 2. All results are normalized to the values obtained for the Choose N Regions algorithm, and the columns pertaining to architecture area-delay product are shaded. The columns labeled "Runs" depict how many architectures each algorithm tested for each domain. The bottom row shows the geometric mean for area-delay products, and the average for runs.

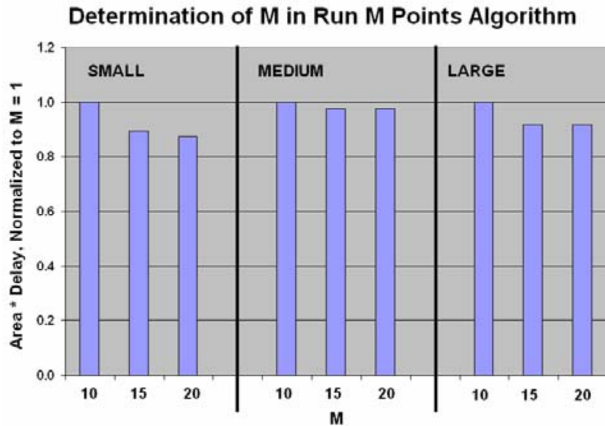

Figure 11. Determination of M in Run M Points algorithm.

From Table 2 it is apparent that creating domain-specific CPLD architectures is a win over using fixed architectures. For each of the five domains that we considered, the algorithms that we developed always came up with a better CPLD architecture than any of the fixed architectures. The closest that any of the fixed architectures comes to a domain-specific algorithm in terms of area-delay product is .54x, which occurred with the 36-48-16 architecture versus the domain-specific architectures found for the sequential domain. At the other end of the spectrum, the 10-12-4 architecture was 37.7x worse than the domain-specific architectures found for the arithmetic domain. Considering the mean performance, the fixed architectures perform 4.1x to 9.5x worse than the Choose N Regions algorithm.

Among the algorithms, the Successive Refinement, Choose N Regions, and Run M Points algorithms tend to choose the same architectures. The only different architectures are from the Run M Points algorithm on the combinational domain, and the Successive Refinement algorithm on the floating point domain. The simpler Hill Descent algorithm was able to match these results for three of the five domains (although with a different architecture for the encryption domain), but came up with decidedly sub-optimal results for the Combinational and Floating Point domains. With respect to runtime, the Hill Descent algorithm took 3.7x to 5.7x fewer runs than the other algorithms.

The successive refinement, Choose N Regions, and Run M Points algorithms all chose 4-8-2 PLAs for the floating point and arithmetic domains in the first step, causing them to be stuck in architectures with small PLAs. The algorithm add-on described in 3.1.2.3 was applied to these instances to remove them from their suboptimal areas. This caused a slight increase in the number of runs that were needed for these algorithms, most notably in the Choose N Regions and Run M Points results.

In addition to the base algorithms, each algorithm was also run with a second iteration that used the IN-PT-OUT ratios found in the first iteration. The thought is that the first iteration would find an IN-PT-OUT ratio that the domain prefers, and the second iteration can use this multiplicative ratio to hone in on a good architecture. The best result from running a second iteration of the algorithms on each domain is shown in Table 3.

Table 3. Best base algorithm results compared to best results after a second iteration.

Domain	Base Alg.	Arch.	Iterated	Arch.
Combinational	1.00	12-71-4	0.79	9-80-4
Sequential	1.00	12-23-6	0.84	18-42-8
Floating Point	1.00	10-24-2	0.82	8-18-2
Arithmetic	1.00	10-20-2	0.91	7-14-2
Encryption	1.00	10-46-3	0.70	13-46-4
Geo. Mean	**1.00**		**0.81**	

As Table 3 shows, running a second iteration of the algorithms was able to improve upon our result for every domain, by anywhere from .09x to .30x. A mean gain of .19x was achieved, at a runtime cost of about 2x (because the second iteration takes about as long as the first iteration). All of the algorithms were able to find improvements by running a second iteration, but it is interesting to note that the best iterated result always came from iterating the Run M Points algorithm. In general, this table shows that running a second iteration can be a profitable add-on to the base algorithms.

While we cannot realistically run the entire 3-D search space on any of these domains, we can run a radial search on the results found in order to see if there are better optima lying near the found results. This does not show us how close we get to optimal, but it gives us a small idea of how much more improvement there might be to gain. In Table 4, the best results found for each domain using an r = 3 radial search are compared to the best

results found using our basic algorithms. Since most architectures were found by multiple algorithms, it didn't tend to matter which algorithm ran a radial search, as long as the search was run for each architecture. As shown, results were found that reduced the area delay product by up to .30x. The runtime cost for the radius = 3 add-on is about 5x to 10x when compared to the base algorithms.

Table 4. Best base algorithm results compared to best results when augmented by radial search.

Domain	Base Alg.	Arch.	Rad. = 3	Arch.
Combinational	1.00	12-71-4	0.77	10-71-6
Sequential	1.00	12-23-6	0.89	15-26-6
Floating Point	1.00	10-24-2	0.83	8-21-2
Arithmetic	1.00	10-20-2	1.00	10-20-2
Encryption	1.00	10-46-3	0.70	13-46-4
Geo. Mean	**1.00**		**0.83**	

The second iteration and radial search algorithm add-ons show that there are some more gains to be had, and it will be interesting to see if we can find new algorithms that can find these results while still using the same number of PLAmap runs. In general, the second iteration and radial search add-ons show that our algorithms are doing very well, as they are within .30x of the best results we can easily find.

5.1 Benefits of Domain-Specific Devices

We have shown that our domain-specific architectures outperform certain representative fixed architectures by 4.1x to 9.5x, but these are not necessarily the best possible fixed architectures for our set of domains. In fact, we have already found the architectures that each domain prefers, so it makes sense that these architectures might work well as fixed architectures. Table 5 shows the area-delay performance of each domain mapped to the best architectures found, normalized to the domain-specific architecture results. These new fixed architectures still only perform within 2.0x to 7.0x of the domain-specific results, even though we have hand selected them to go well with our domains. This shows that even if you manage to pick the best possible domain-generic fixed architecture, there is a bound as to how close you can come to domain-specific results – in this case, domain-specific beats fixed architectures by 2x.

Table 5. Results of running each domain on the best domain-specific architectures found.

Domain	10-71-6	18-42-8	8-18-2	7-14-2	13-46-4
Combinational	1.00	9.62	5.46	4.33	4.11
Sequential	2.60	1.00	3.81	6.04	2.05
Floating Point	9.66	15.76	1.00	1.02	3.26
Arithmetic	6.95 *	33.49	1.82	1.00	6.34
Encryption	1.71	3.42	1.09	1.19	1.00
Geo. Mean	**2.56**	**7.04**	**2.10**	**1.99**	**2.80**

In Table 5, the entry with an asterisk (*) had to be estimated because the PLAmap software was unable to run the SQRT circuit on the specified architecture for unknown reasons.

6. Conclusion

In this paper we have presented a complete tool flow for creating domain-specific CPLDs for System-on-a-Chip devices. This includes an Architecture Generator which finds a domain-specific CPLD architectures by using any of four basic search algorithms. When compared to realistic fixed CPLD architectures, the domain-specific architectures perform 4.1x to 9.5x better in terms of area-delay product. Additionally, iterating the algorithms and performing radial searches around the chosen architectural points show that our fast algorithms are finding architectures that are within .30x of the best architectures that we can easily find.

Of the base results, the Choose N Regions algorithm provided the best results in terms of performance, with a runtime that was beaten only by the simple Hill Descent algorithm. Thus the Choose N Regions algorithm is the best to use if you are only running a single iteration. If doubling the runtime is acceptable, then the Run M Points algorithm should be used and it should be run with a second iteration, as the best overall results we found were obtained from this technique.

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

7. Future Work

Our cost function currently has no concept of power consumption, which is a very important component of modern VLSI components. Future analysis will need to incorporate power values in order to robustly evaluate our architectures.

Many CPLDs use central routing architectures that are smaller than crossbars, or they use hierarchical routing structures. It would be interesting to see how much more performance could be achieved by using these routing structures.

We have provided results strictly in regards to area-delay product. It is conceivable that an SoC designer would be much less interested in area-delay product than in simply area or delay. A likely scenario would have the SoC designer setting a hard area (or delay) limit that needs to be met, and we would need to provide him with the fastest (or smallest) architecture that meets the area constraint. Similar scenarios can be constructed with regards to power consumption. We will need to develop algorithms that can find good domain-specific architectures given such constraints.

And finally, we have seen from the radial search and second iteration add-ons that there is at least .30x more performance to be gained. It would be interesting to develop new algorithms in order to obtain those gains.

Acknowledgments

The assistance of Mike Hutton and Swati Pathak at Altera was essential in getting this work done, 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] A. Yan, S. Wilton, "Product Term Embedded Synthesizable Logic Cores", *IEEE International Conference on Field-Programmable Technology*, 2003.
[2] N. Kafafi, K. Bozman, S.J.E. Wilton, ``Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores", FPGA, 2003.
[3] F. Mo, R. K. Brayton, "River PLAs: A Regular Circuit Structure", *DAC*, 2002.
[4] M. Holland, S. Hauck, "Automatic Creation of Reconfigurable PALs/PLAs for SoC", FPL, 2004.
[5] D. Chen, J. Cong, M. Ercegovac, Z. Huang, "Performance-Driven Mapping for CPLD Architectures", FPGA, 2001.
[6] S. Phillips, "Automating Layout of Reconfigurable Subsystems for Systems-on-a-Chip", PhD Thesis, University of Washington, Dept. of EE, 2004.
[7] FIPS PUB 197, Advanced Encryption Standard (AES), National Institute of Standards and Technology, U.S. Department of Commerce, November 2001
[8] J. Kouloheris, A. El Gamal, "FPGA Performance vs. Cell Granularity", Proc. Custom Integrated Circuits Conference, 1991.