ADAPTIVE FPGAS: HIGH-LEVEL ARCHITECTURE AND A SYNTHESIS METHOD

Valavan Manohararajah, Stephen D. Brown, and Zvonko G. Vranesic

Department of Electrical and Computer Engineering
University of Toronto
manohv|brown|zvonko@eecg.toronto.edu

ABSTRACT

This paper presents preliminary work exploring adaptive field programmable gate arrays (AFPGAs). An AFPGA is adaptative in the sense that the functionality of subcircuits placed on the chip can change in response to changes observed on certain control signals. We describe the high-level architecture which adds additional control logic and SRAM bits to a traditional FPGA to produce an AFPGA. We also describe a synthesis method that identifies and resynthesizes mutually exclusive pieces of logic so that they may share the resources available in an AFPGA. The architectural feature and its associated synthesis method helps reduce circuit size by 28% on average and up to 40% on select circuits.

1. INTRODUCTION

Field programmable gate arrays (FPGAs) have become an increasingly popular medium for implementing digital circuits due to the high costs associated with application specific integrated circuit (ASIC) implementations. An FPGA can be bought off the shelf and can be configured to implement an arbitrary digital circuit. This programmability allows reduced development and verification times, and avoids the large costs involved in chip fabrication. The programmability of FPGAs comes with a steep price, however. An FPGA-based implementation is estimated to be forty times larger and three times slower than a comparable implementation using standard cells [1]. A number of studies have examined methods of improving the area and speed efficiency of FPGAs [2]–[9]. Here we consider an architectural modification that improves the area efficiency of existing FPGA architectures by dynamically changing the functionality of the circuit implemented in the programmable logic fabric.

The key difference between an AFPGA and FPGA is in the structure of the configuration element. In an FPGA, each configuration element determines the functionality of a single programmable resource as illustrated in Figure 1a. Although most of the configuration elements in an AFPGA are the same as in an FPGA, a fraction of them use the alternative structure illustrated in Figure 1b. In this structure, two configuration elements are connected to a single program-

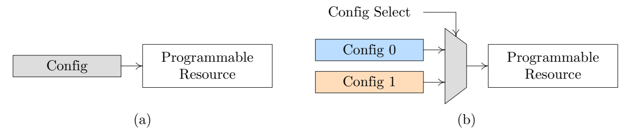

Fig. 1. Structure of a configuration element in an FPGA (a) and an AFPGA (b).

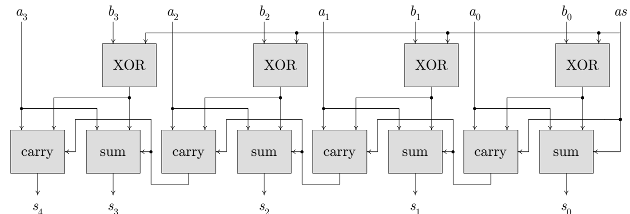

Fig. 2. A four-bit add-subtract unit.

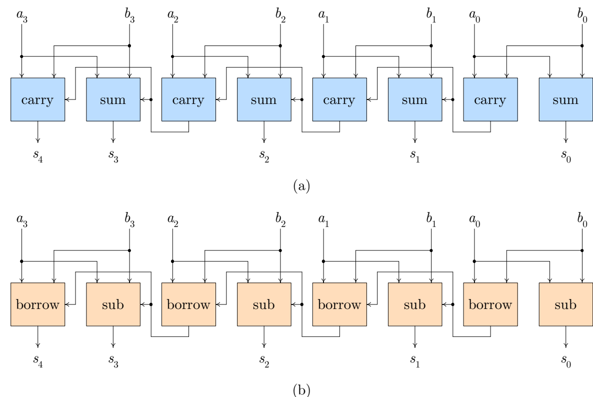

Fig. 3. Specialized versions of the add-subtract unit: (a) is specialized to perform an addition and (b) is specialized to perform a subtraction.

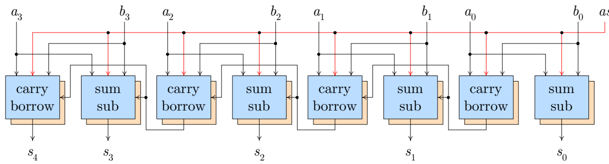

Fig. 4. The four-bit add-subtract unit implemented on an AFPGA.

mable resource through a multiplexer. The functionality of the programmable resource is determined by the configuration element selected by the multiplexer. The motivation for the study of this alternative structure is best illustrated with an example. Consider an FPGA architecture where a logic block can implement any function of three inputs. A possible implementation of a four-bit add-subtract unit for this architecture is given in Figure 2. A subtraction is performed when the control signal as (*AddSub*) is equal to 1 and an addition is performed when it is equal to 0. This implementation uses 12 logic blocks. The add-subtract unit can be specialized to perform either an addition or a subtraction by setting as to zero or one, respectively. After setting as to a constant, logic simplification removes the four XOR blocks reducing the number of logic blocks required to 8. The specialized versions of the add-subtract unit are illustrated in Figures 3a and 3b. Although the logic blocks have been specialized to perform different functions, the routing structure is still identical. By allowing the logic blocks to use the multiplexed configuration element, the two specialized circuits can be combined into a single circuit as illustrated in Figure 4. In the combined circuit, each logic block implements the functionality required by both addition and subtraction. The appropriate function is activated by as which is assumed to connect to a special configuration select signal. This example shows that there are potential area benefits when the multiplexed configuration element is used selectively in an FPGA architecture.

The architectural modification we are proposing adds a control signal and some extra configuration memory in order to reduce the size of the circuit to be implemented. In modern FPGAs [11, 12] configuration memory is a small fraction of total chip area, and a small portion of this configuration memory is duplicated in order to allow two distinct subcircuits to share the same set of programmable resources. A detailed study of this area tradeoff can be found in [10]. Here we present a high-level overview of the architecture and present a synthesis method that helps reduce circuit size significantly when targeting AFPGAs.

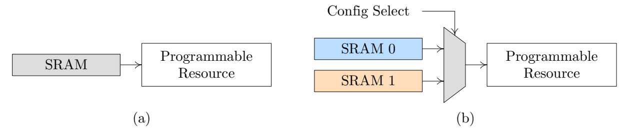

Fig. 5. (a), the configuration element used by FPGAs, and (b), the configuration element used only by AFPGAs.

2. AFPGAS: HIGH-LEVEL ARCHITECTURE

AFPGAs are built by modifying an existing SRAM configurable FPGA. Although other programming methods such as antifuses [13] and floating gate devices [14, 15, 16] can be considered, these are less popular and are not considered in this work.

Before we describe the additional circuitry needed to create AFPGAs, we briefly review the academic FPGA architecture [4] that is the basis for our work. At the highest level, an FPGA architecture is comprised of I/O pins and logic clusters. Programmable routing resources comprised of programmable multiplexers and routing switches allow connections to be made between logic clusters, and between logic clusters and I/O pins. Each logic cluster is made up of a number of basic logic elements (BLEs), and each BLE contains a lookup table (LUT) and a register. Here, we assume that the target architecture uses 4-input LUTs. Previous work has established the area effiency of this particular LUT size [5, 6].

The distinguishing feature of AFPGAs is their use of the adaptive configuration element illustrated in Figure 5b. This is in contrast to FPGAs which use the traditional configuration element illustrated in Figure 5a. In the traditional configuration element, each programmable resource is attached to a single SRAM cell which determines its functionality. In the adaptive configuration element, two SRAM cells connect to the programmable resource through a multiplexer, and the SRAM cell that gets to control the programmable resource is determined by a configuration select signal attached to the multiplexer. If the two SRAM cells contain differing values, the programmable resource being controlled changes its functionality dynamically in response to changes in the select signal.

An AFPGA architecture is obtained by making two modifications to an existing FPGA architecture. First, a portion of the configuration elements in an FPGA are replaced by adaptive configuration elements. Since the adaptive configuration elements require a select signal, the second modification adds select signals, each controlling the configuration elements in a region of the chip. The select signals are accessible from the programmable routing network allowing them to be driven by I/O pins and basic BLE outputs.

The adaptive BLE, illustrated in Figure 6a, is obtained

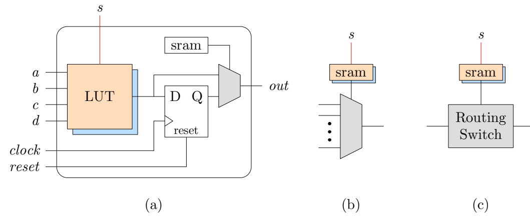

Fig. 6. (a), an adaptive BLE, (b), an adaptive multiplexer, and (c), an adaptive routing switch.

from a traditional BLE by modifying the configuration elements that determine LUT functionality. Using adaptive configuration elements in place of traditional configuration elements, the LUT inside the adaptive BLE has the ability to perform two different functions of its inputs. A select signal, s, connects to the adaptive configuration elements and provides a way of selecting the function computed by the LUT.

Multiplexers and routing switches present in an FPGA can be made adaptive by replacing their configuration elements with adaptive configuration elements. The adaptive multiplexer, illustrated in Figure 6b, has the capability of gating two different signals through to the output depending on the value of the select signal, s. The adaptive routing switch, illustrated in Figure 6c, supports two modes of operation where the current mode is determined by the select signal, s. The switch has the capability of being turned on or off under the control of the select signal if its SRAM cells have been programmed to two different values.

An AFPGA uses adaptive BLEs, multiplexers and routing switches to allow two subcircuits to share the same set of programmable resources.

3. AFPGAS: CAD FLOW

The CAD flow for AFPGAs is illustrated in Figure 7. The first two steps, synthesis and technology mapping, are the same as in an FPGA CAD flow. However, starting with the third step, *adaptation*, there are significant differences between the two CAD flows.

The process of adaptation creates pairs of subcircuits that share programmable resources in an AFPGA. There are two technology mapping steps in the AFPGA CAD flow, the first precedes adaptation and the second follows adaptation. The first mapping step is useful because it improves the performance of adaptation. During adaptation, the selection of control signals is guided by area concerns, and the accuracy of the selection process is greatly enhanced if the circuit has been mapped into LUTs. The second technology mapping step is useful because it produces a smaller circuit by remapping parts of the circuit affected by the simplification operations (described in Section 4.2) that take place during adaptation.

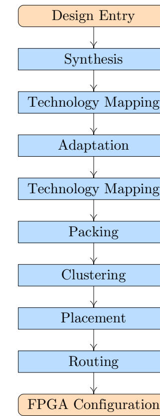

Fig. 7. The AFPGA CAD flow.

An AFPGA architecture is based on an existing FPGA architecture, thus the final four steps in the AFPGA CAD flow, packing, clustering, placement and routing, perform their traditional FPGA CAD tasks in addition to new tasks that are AFPGA specific. In addition to the task of grouping LUTs and registers into BLEs, the packing step groups pairs of LUTs selected from pairs of subcircuits to form adaptive BLEs. Clustering and placement have the additional task of ensuring that the BLEs that are part of a subcircuit pair are clustered and placed in a region with adaptive structures (BLEs, multiplexers and routing switches) whose functionality changes in response to a single configuration select signal. Finally, the routing step has two additional tasks. First, it has to make use of adaptive multiplexers and routing switches when making connections for subcircuit pairs. Second, it has to connect the control signals that generated subcircuit pairs to the appropriate configuration select signals.

4. ADAPTATION

4.1. Introduction

A circuit may be viewed as being composed of several subcircuits. The process of adaptation replaces some of these subcircuits with adaptive subcircuits capable of sharing the adaptive resources present in an AFPGA. An adaptive subcircuit consists of two *half* circuits, each a specialized version of the subcircuit being replaced. The creation of an adaptive subcircuit begins with the selection of a control signal which is then used to identify a subcircuit whose area is greatly reduced when the control signal is assumed to be a constant. Two specialized versions of the subcircuit can be generated, one assuming that the control signal is zero and

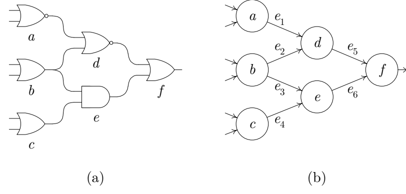

Fig. 8. A circuit and its corresponding graph.

the other assuming that the control signal is one. Since the control signal can only be in one of two possible states, zero or one, the specialized subcircuits are not needed at the same time and can be combined to form the two halves of an adaptive subcircuit. The control signal connects to the adaptive subcircuit and determines which of the two halves is to be active at any time.

4.2. Preliminaries and Problem Definition

The combinational portion of a boolean circuit can be represented as a directed acyclic graph (DAG) $G = (V(G), E(G))$. A node in the graph $v \in V(G)$ represents a logic gate, primary input or primary output, and a directed edge in the graph $e \in E(G)$ with head, $u = head(e)$, and tail, $v = tail(e)$, represents a signal in the logic circuit that is an output of gate u and an input of gate v. The set of *input edges* for a node v, $iedge(v)$, is defined to be the set of edges with v as a tail. Similarly, the set of *output edges* for v, $oedge(v)$, is defined to be the set of edges with v as a head. A *primary input* (PI) node has no input edges and a *primary output* (PO) node has no output edges. An *internal* node has both input edges and output edges. The set of distinct nodes that supply input edges to v are referred to as *input nodes* and is denoted $inode(v)$. Similarly, the set of distinct nodes that connect to output edges from v are referred to as *output nodes* and is denoted $onode(v)$.

The circuit of Figure 8a and its graph representation, Figure 8b, will be used as an example when illustrating the notions defined below.

A node v in the graph is considered to be a boolean variable v whose value is determined by the boolean function $\mathcal{F}_v$ at node v. Although we use the same notation for both a node and the variable it is associated with, it will be clear from the context which one of the two is meant. An edge e from u to v indicates that the function used to compute v, $\mathcal{F}_v$, depends on variable u. In the example, the function at d depends on a and b, and is defined to be $\mathcal{F}_d(a, b) = \overline{a + b}$.

In addition to the use of the standard logic values 0 and 1, the symbol '$*$' is used to represent an unknown or undefined logic value.

The ordered pair $[v, l]$, where $l \in \{0, 1\}$, represents the assignment of logic value l to variable v. An *implication* of the assignment $[v, l]$ is another assignment that is a consequence of $[v, l]$. The implication set, $impl[v, l]$, is the set of implications that are a consequence of the assignment $[v, l]$. In the example, an implication of the assignment $[b, 0]$ is the assignment $[e, 0]$, and the implication set of $[d, 1]$ is as follows

$$impl[d, 1] = \{[d, 1], [a, 0], [b, 0], [e, 0], [f, 1]\}.$$

Note that the implication set of an assignment includes the assignment itself.

Two laws regarding implications will be useful when generating implication sets. The *transitive* law states that if $[v, l] \in impl[u, k]$ and $[w, m] \in impl[v, l]$ then $[w, m] \in impl[u, k]$, and the *contrapositive* law states that if $[v, l] \in impl[u, k]$ then $[u, \overline{k}] \in impl[v, \overline{l}]$.

If the output or any of the inputs of a function $\mathcal{F}_v$ have been assigned values, then a *justification* operation, $\text{JUSTIFY}(v)$, can be performed on v. The operation returns a set of assignments that are a consequence of the assignments incident on v. The scope of the operation is limited to the assignments that can be derived by examining $\mathcal{F}_v$; the assignments returned only apply to the functions's inputs or to its output. Returning to our example, if $d = 1$ and $f = *$ then $\text{JUSTIFY}(f)$ returns the assignment $[f, 1]$. Similarly, if $d = 1$, $a = *$, and $b = *$ then $\text{JUSTIFY}(d)$ returns two assignments, $[a, 0]$ and $[b, 0]$.

The structure of a graph can be simplified based on the assignments made to its variables. For example, in Figure 8, if $d = 1$ then the function at d can be replaced by the constant-1 function, $\mathcal{F}_d = 1$, and edges e_1 and e_2, and node a can be removed. Similarly, if $b = 0$, then the function at d can be replaced by $\mathcal{F}_d = \overline{a}$, and edge e_2 can be removed. After some assignments have been made to the variables in G, the simplification operation, $\text{SIMPLIFY}(H)$, on a subgraph H of G returns a new subgraph which is a simplified version of H based on the assignments present. The process of simplification can be expressed as a sequence of two simple operations. The first operation simplifies a node's function based on the assignments present and removes the input edges that are no longer needed by the new function. The second operation removes a node and its input edges if all of its output edges have been removed. These two operations are continually applied until they have no effect on the graph.

The subgraph *controlled* by a variable v, H_v, consists of the nodes and edges affected by the simplifications that are possible after using the assignments in $impl[v, 0]$ and $impl[v, 1]$. The subgraph used to compute v is not part of the subgraph controlled by v. In our example, $impl[b, 0] = \{[b, 0], [e, 0]\}$ and $impl[b, 1] = \{[b, 1], [d, 0]\}$. If we use the assignments in $impl[b, 0]$ then the subgraph consisting of nodes, $\{d, e, c, f\}$, and edges, $\{e_2, e_3, e_4, e_6\}$, is affected by the simplifications that are possible, and if we use the

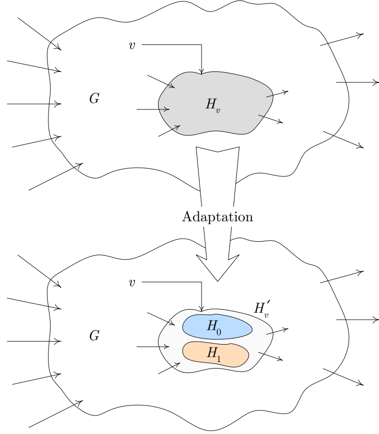

Fig. 9. The process of adaptation.

assignments in $impl[b, 1]$ then the subgraph consisting of nodes, $\{d, e, a, f\}$, and edges, $\{e_2, e_3, e_1, e_5\}$, is affected by the simplifications that are possible. Combining these two subgraphs, we obtain the subgraph controlled by b which is essentially the graph of Figure 8b with node b removed.

Figure 9 illustrates the transformation that takes place during adaptation. First, a control variable, v, and the subgraph it controls, H_v, are identified. Then, two specialized versions of H_v are generated; H_0 is obtained by simplifying H_v after applying the assignments in $impl[v, 0]$ and H_1 is obtained by simplifying H_v after applying the assignments in $impl[v, 1]$. Finally, the two subgraphs are combined to form the adaptive subgraph, H_v'. Variable v remains connected to the adaptive subgraph and determines which one of H_0 and H_1 is to be active at any time.

The area of a subgraph H, $area(H)$, is defined to be the number of nodes in it. In an adaptive subgraph, only one of the component subgraphs is active at any time. Thus, the area of an adaptive subgraph is defined to be the maximum of the areas of the component subgraphs,

$$area(H_v') = \max\{area(H_0), area(H_1)\}. \qquad (1)$$

Note that the area of an adaptive subgraph H_v' is always smaller than or equal to the area of H_v because H_v' is composed of the two subgraphs H_0 and H_1 which are both simplified versions of H_v.

The goal of adaptation is to replace controlled subgraphs with adaptive subgraphs such that the area of the resulting graph is minimized. Consider the process of adaptation on the graph of Figure 8b using b as the control variable. We identified the subgraph controlled by b earlier. The subgraphs of Figures 10a and 10b are obtained when the controlled subgraph is simplified assuming b is 0 and 1, respectively. These two subgraphs are combined into the adaptive subgraph illustrated in Figure 10c. Although

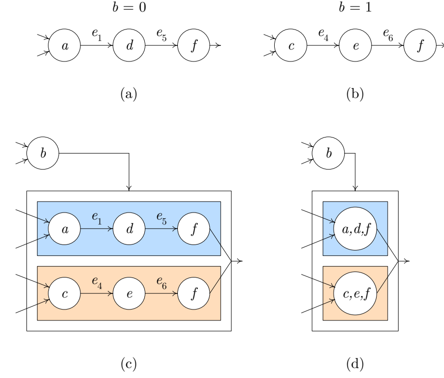

Fig. 10. The process of adaptation on the graph of Figure 8.

the adaptive subgraph reduces the area of the graph by two nodes, even more gains are possible if the adaptive subgraph is remapped. In the original graph, all nodes had two inputs, and if we assume that the target architecture consists of 2-LUTs then remapping produces the graph of Figure 10d. The final graph has a three-fold reduction in area compared to the original.

4.3. Generating Implication Sets

Implication sets are central to the problem of adaptation. They are used to identify controlled subgraphs as well as to generate adaptive subgraphs. Our use of implication sets is not new; they have found several uses in the literature. They have been used to solve satisfiability (SAT), automatic test pattern generation (ATPG), and logic synthesis problems. Thus, there have been a number of algorithms proposed to generate implication sets [17]–[22]. Our work uses a modified version of the algorithm proposed in [22]. The primary benefits of this algorithm over the others are its speed, its ability to generate all possible implication sets simultaneously and its ability to discover a large number of implications. We briefly review the algorithm here.

The algorithm maintains implication sets for each signal in the circuit. Initially, each implication set will contain a single entry: the assignment itself. As the algorithm progresses the implication sets are expanded. The algorithm stops when there are no further changes observed on any of the implication sets. An implication set is expanded by first making the assignments within it. Then justification operations are used to discover new implications. The transitive law is used to discover further implications of the newly discovered implications. In addition to the transtive law, during the process of discovery, the contrapositive law is used to add new entries to other implication sets.

1	ADAPT()
2	IMPLICATIONSETS()
3	$C \leftarrow V(G)$
4	**forever**
5	$bestv \leftarrow$ **nil**, $bestc \leftarrow 0$
6	**for** $v \in C$
7	**skip if** $v \notin V(G)$
8	$H_0 \leftarrow$ SPECIALIZE$(v, 0)$
9	$H_1 \leftarrow$ SPECIALIZE$(v, 1)$
10	$c \leftarrow$ COST(H_v, H_0, H_1)
11	**if** $c > bestc$
12	$bestv \leftarrow v, bestc \leftarrow c$
13	**end if**
14	**end for**
15	**break if** $bestv =$ **nil**
16	$H_0 \leftarrow$ SPECIALIZE$(bestv, 0)$
17	$H_1 \leftarrow$ SPECIALIZE$(bestv, 1)$
18	$G \leftarrow G - H_{bestv}$
19	$G \leftarrow G \cup$ FORMADAPTIVE(H_0, H_1)
20	**end forever**

Fig. 11. The procedure used to generate adaptive subgraphs.

1	SPECIALIZE(v, l)
2	**for** $[u, k] \in impl[v, l]$
3	**skip if** $u \notin V(G)$
4	$u \leftarrow k$
5	**end for**
6	$H' \leftarrow simplify(H_v)$
7	**for** $[u, k] \in impl[v, l]$
8	**skip if** $u \notin V(G)$
9	$u \leftarrow *$
10	**end for**
11	**return** H'

Fig. 12. The procedure used to generate a specialized version of H_v under the assumption that v has logic value l.

4.4. Generating Adaptive Subgraphs

The procedure used to generate adaptive subgraphs is presented in Figure 11. It begins with a call to IMPLICATIONSETS which generates all implication sets in the graph. Then, before any modifications are made to the graph, a copy of the variables is placed in C. The implication sets that were generated only apply to the variables that are present in the unmodified graph, but some of these variables will disappear as the graph is modified during adaptation. Set C allows us to keep track of the variables for which implication sets are available.

The inner loop (lines 6–14) uses a greedy method to find a control variable that can be used to generate an adaptive subgraph, and the outer loop (lines 4–20) keeps replacing subgraphs with their adaptive equivalents until the inner loop fails to find a control variable. For every variable that is present in both C and the modified graph G, the inner loop determines a cost that reflects its suitability as a control variable. The cost of a variable v is determined by COST and is based on the controlled subgraph, H_v, as well as the two subgraphs H_0 and H_1 produced by SPECIALIZE. The variable with the highest non-zero cost is selected as the control variable to be used in generating an adaptive subgraph.

Given a variable v and a logic value l, the specialization procedure, illustrated in Figure 12, produces a version of H_v simplified using the implications in $impl[v, l]$. Note that some assignments in $impl[v, l]$ cannot be used (lines 3 and 8) as the variables they apply to may no longer be a part of the modified graph G.

Returning to Figure 11, having selected a control variable in the inner loop, the outer loop removes the subgraph controlled by the variable and replaces it with an adaptive subgraph composed of the two specialized versions of the removed subgraph.

Since the goal of adaptation is to minimize the area of the resulting circuit, the cost computed for a variable v is based on the number of edges eliminated in H_v to obtain H_0 and H_1. Although the area savings can be measured directly by counting the number of nodes eliminated, the edge-based measure is finer grained and better reflects the area obtained after the technology mapping step that follows adaptation; the technology mapping step may be able to pack more logic into the nodes where some edges have been eliminated. The cost is computed as the minimum of two values c_0 and c_1 where c_0 is the number of edges eliminated to obtain H_0 and c_1 is the number of edges eliminated to obtain H_1. Furthermore, there are two special cases that may cause the cost of a variable to be zero preventing its selection by the inner loop (lines 11–13). It is assumed that at minimum each control signal in the target architecture determines the functionality of the adaptive structures within a region that is approximately the size of a cluster. Thus, to produce adaptive subgraphs that are at least as big as a cluster, the cost of a variable is set to zero if it does not eliminate as many edges as there are inputs to a typical cluster in the target architecture. For example, if the target architecture uses clusters with four BLEs, then a variable's cost is set to zero if it does not eliminate at least 10 edges [2]. If a variable v controls a large subgraph H_v and has disproportionate values for c_0 and c_1 then it is unlikely to be a good control variable and its cost is set to zero. For example, consider a circuit with a global reset signal. The reset signal controls most of the circuit, and most of the circuit can be removed if it is assumed to be one. However, if the reset signal is assumed to be zero, most of the circuit remains unchanged. If an adaptive subcircuit is generated using the reset signal, area would not be reduced significantly and a large part of the circuit would no longer be able to participate in other specialization operations. Far better area results can be obtained for the circuit by using a number of control signals, each of which controls a smaller subcircuit and is able to produce balanced values for c_0 and c_1.

The complexity of generating adaptive subgraphs (lines 4–20) can be derived as follows. The number of nodes in the graph is denoted n, and the average number of nodes in the subgraph controlled by a variable is denoted k. The

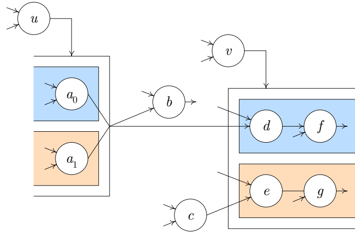

Fig. 13. Existence of adaptive subgraphs imposes constraints on the way cones are generated.

inner loop simplifies the subgraph controlled by each variable to determine a cost, thus its complexity is $O(kn)$. The outer loop will iterate $O(n/k)$ times as each iteration replaces a subgraph of size $O(k)$ with an adaptive subgraph. Therefore, the complete process has a complexity of $O(n^2)$. On a Pentium III 1 GHz computer, adaptive subgraphs can be generated for the 20 largest MCNC [23] circuits in 19 minutes.

5. TECHNOLOGY MAPPING AFTER ADAPTATION

Although the technology mapping procedure used after adaptation is the same as the one that precedes adaptation, the existence of adaptive subgraphs imposes constraints on the way cones are generated. A node v that is a part of an adaptive subgraph may include a predecessor node u in a cone at v only if u is a part of the same adaptive subgraph or u is not a part of any adaptive subgraph. A node v that is not a part of an adaptive subgraph may only include predecessors that are not a part of any adaptive subgraph in a cone at v. For example, in Figure 13, d can be a part of cone at f, e can be a part of a cone at g, and c can be a part of a cone at e. However, a_0/a_1 cannot be a part of a cone at b or d.

6. RESULTS

The 20 largest MCNC [23] circuits were used to study the benefits of the adaptation step. Before undergoing adaptation, each circuit was first synthesized using SIS (`script-rugged`) [24] and then technology mapped into 4-LUTs using IMap [25]. IMap was also used for the second technology mapping step following adaptation. For each circuit, Table 1 presents the number of control variables used during adaptation (*Ctrls*), the area before adaptation (*Unadapted*), the area following adaptation (*Adapted*), the area following the second technology mapping step (*Mapped*), and the fraction of LUTs in the final circuit that are adaptive (*Adaptive*).

Although an average of 7.1 control signals were used during adaptation, there are significant differences in the

Circuit	*Ctrls*	*Unadapted (LUTs)*	*Adapted (LUTs)*	*Mapped (LUTs)*	*Adaptive (Ratio)*
`C6288`	15	904	904	714	0.69
`alu4`	7	1035	820	780	0.97
`apex2`	9	1201	960	921	0.95
`apex4`	4	1087	850	733	0.89
`bigkey`	1	1594	1366	912	0.87
`clma`	13	4545	2801	2686	0.98
`des`	18	1225	1092	1010	0.85
`diffeq`	12	849	836	823	0.73
`dsip`	1	1153	1145	691	0.99
`elliptic`	15	2113	2100	2070	0.62
`ex1010`	6	2532	1663	1480	0.94
`ex5p`	2	931	710	602	0.93
`frisc`	22	2269	2087	1997	0.85
`i10`	13	779	680	661	0.81
`misex3`	4	1086	678	611	0.98
`pdc`	4	1882	1349	1190	0.97
`s38417`	30	3672	3477	3362	0.86
`s38584.1`	16	3730	3586	2916	0.93
`seq`	5	1089	789	726	0.96
`spla`	5	1337	981	853	0.99
GeoAvg	7.10	1510.20	1241.06	1090.78	0.88
Ratio		1.0	0.82	0.72	

Table 1. The effect of adaptation on the 20 largest MCNC circuits.

number of control variables used to adapt each circuit. Some circuits such as `bigkey` produce significant area reductions with a single control variable while circuits such as `clma` produce similar area gains with a much larger number of control variables (13). Architectural decisions will be guided by the larger number of control variables as an architecture that supports several control variables can be made to mimic a single control variable, whereas an architecture that supports a single control variable cannot mimic the presence of several control variables.

When adaptation is used in isolation, an average area savings of 18% is observed. The addition of the second technology mapping step increases the area savings to 28%.

A large fraction of the circuit produced by adaptation (88%) is adaptive. From an architectural point of view, this suggests that most of the LUTs in an AFPGA must be of the adaptive variety to effectively support the circuits being produced by adaptation.

7. WORK DESCRIBED ELSEWHERE

Although adaptive circuits use 28% fewer LUTs than non-adaptive circuits, this is only one of the factors in the area comparison between adaptive circuits. To determine the true area benefits of AFPGAs, two other factors need to be quantified. First, the area of the extra circuitry required to implement adaptive circuits needs to measured. Second, the structural changes that a circuit underwent to become adaptive may produce an associated increase or decrease in the demand for routing resources, and this effect needs to be measured. A detailed area study considering these effects is presented in [10].

8. SUMMARY

We presented a high-level overview of the AFPGA architecture and detailed the adaptation step which was used to exploit the architectural features present in an AFPGA. Two algorithms that were part of the adaptation step were described. The first algorithm generated implication sets which were then used by the second algorithm to generate adaptive subcircuits. Following adaptation, a technology mapping step was run to reduce both the area and depth of the resulting circuits. The entire procedure reduced the area of the 20 largest MCNC circuits by 28%.

9. REFERENCES

[1] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. In *Proceedings of the ACM International Symposium on FPGAs*, Monterey, CA, February 2006, pp. xx–xx.

[2] V. Betz and J. Rose. Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs Input Sharing and Size. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, Santa Clara, CA, 1997, pp. 551–554.

[3] A. Marquardt, V. Betz and J. Rose. Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density. In *Proceedings of the ACM International Symposium on FPGAs*, Monterey, CA, February 1999, pp. 37–46.

[4] V. Betz, J. Rose and A. Marquardt. *Architecture and CAD for Deep Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[5] J. Rose, R. Francis, D. Lewis and P. Chow. Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency. *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 5, October 1990, pp. 1217–1225.

[6] E. Ahmed and J. Rose. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. In *Proceedings of the ACM Symposium on FPGAs*, Monterey, CA, February 2000, pp. 3–12.

[7] J. Rose and S. Brown. Flexibility of Interconnection Structures for Field-Programmable Gate Arrays. *IEEE Journal of Solid-State Circuits*, Vol. 26, No. 3, March 1991, pp. 277–282.

[8] V. Betz and J. Rose. FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density. In *Proceedings of the ACM International Symposium on FPGAs*, Monterey, CA, February 1999, pp. 59–68.

[9] M. Sheng and J. Rose. Mixing Buffers and Pass Transistors in FPGA Routing Architectures. In *Proceedings of the ACM International Symposium on FPGAs*, Monterey, CA, February 2001, pp. 75–84.

[10] V. Manohararajah. *Area Optimizations in FPGA Architecture and CAD*, Ph.D. Thesis, University of Toronto, 2005.

[11] Altera. *Altera Device Handbook*. 2005.

[12] Xilinx. *Xilinx Device Handbook*. 2005.

[13] Actel Corporation. *Axcelerator Family FPGAs Product Brief*. January 2005.

[14] Actel Corporation. *ProASIC3 Product Brief*. July 2005.

[15] Altera Corporation. *Max II Device Brochure*. v1.0, March 2004.

[16] Xilinx Corporation. *CoolRunner-II CPLD Family Data Sheet*. v2.5, June 2005.

[17] M. H. Schulz, E. Trischler and T. M. Sarfert. SOCRATES: A Highly Efficient Automatic Test Pattern Generation System. *IEEE Transactions on Computer-Aided Design*, Vol. 7, No. 1, January 1988, pp. 126–136.

[18] H. Fujiwara and T. Shimono. On the Acceleration of Test Generation Algorithms. *IEEE Transactions on Computers* Vol. C-32, No. 12, December 1983, pp. 1137–1144.

[19] J. Rajski and H. Cox. A Method to Calculate Necessary Assignments in Algorithmic Test Pattern Generation. In *Proceedings of the IEEE International Test Conference*, Semptember 1990, pp. 25–34.

[20] S. T. Chakradhar and V. D. Agrawal. A Transitive Closure Algorithm for Test Generation. *IEEE Transactions on Computer-Aided Design*, Vol. 12, No. 7, July 1993, pp. 1015–1028.

[21] W. Kunz and D. Stoffel. *Reasoning in Boolean Networks*. Kluwer Academic Publishers, 1997.

[22] J. Zhao, M. Rudnick and J. Patel. Static logic implication with application to fast redundancy identification. In *Proceedings of the VLSI Test Symposium*, 1997, pp. 288–293.

[23] Collaborative Benchmarking Laboratory. *LGSynth93 Benchmark Suite*. Available from `http://www.cbl.ncsu.edu/www/`.

[24] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. L. Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuit Synthesis*. Technical Report, University of California at Berkeley, 1992, Memorandum No. UCB/ERL M92/41.

[25] V. Manohararajah, S. D. Brown, and Z. G. Vranesic. Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping. In *Proceedings of the International Workshop on Logic and Synthesis*, Temecula Creek Inn, CA, USA, June 2004, pp. 14–21.