Dynamic Data Folding with Parameterizable FPGA Configurations

KAREL BRUNEEL, WIM HEIRMAN, and DIRK STROOBANDT, Ghent University

In many applications, subsequent data manipulations differ only in a small set of parameter values. Because of their reconfigurability, FPGAs (field programmable gate arrays) can be configured with a specialized circuit each time the parameter values change. This technique is called dynamic data folding. The specialized circuits are smaller and faster than their generic counterparts. However, the overhead involved in generating the configurations for the specialized circuits at runtime is very large when conventional tools are used, and this overhead will in many cases negate the benefit of using optimized configurations.

This article introduces an automatic method for generating runtime parameterizable configurations from arbitrary Boolean circuits. These configurations, in which some of the configuration bits are expressed as a closed-form Boolean expression of a set of parameters, enable very fast run-time specialization, since specialization only involves evaluating these expressions. Our approach is validated on a ternary content-addressable memory (TCAM). We show that the specialized configurations, produced by our method use 2.82 times fewer LUTs than the generic configuration, and even 1.41 times fewer LUTs than the implementation generated by Xilinx Coregen. Moreover, while Coregen needs hand-crafted generators for each type of circuit, our toolflow can be applied to any VHDL design. Using our automatic and generally applicable method, run-time hardware optimization suddenly becomes feasible for a large class of applications.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Automatic synthesis*; B.7.1 [**Integrated Circuits**]: Types and Design Styles; B.7.2 [**Integrated Circuits**]: Design Aids; J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design (CAD)*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Automatic hardware synthesis, dynamic data folding, FPGA, run-time reconfiguration

ACM Reference Format:
Bruneel, K., Heirman, W., and Stroobandt, D. 2011. Dynamic data folding with parameterizable FPGA configurations. ACM Trans. Des. Autom. Electron. Syst. 16, 4, Article 43 (October 2011), 29 pages.
DOI = 10.1145/2003695.2003703 http://doi.acm.org/10.1145/2003695.2003703

1. INTRODUCTION

The inherent reconfigurability of SRAM-based FPGAs makes it possible to dynamically optimize the configuration of the FPGA for the situation at hand. Since optimized configurations are smaller and faster than their generic counterparts, this may result in a more efficient use of FPGA resources.

If the number of possible situations is limited, a dynamically reconfiguring system can easily be implemented with a conventional FPGA tool flow. We simply generate an FPGA configuration optimized for each possible situation and store these in a configuration database. At run-time, a configuration manager loads the appropriate configuration from the database in the FPGA, depending on the situation at hand.

Author's address: K. Bruneel, ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium; email: karel.bruneel@ugent.be.

DOI 10.1145/2003695.2003703 http://doi.acm.org/10.1145/2003695.2003703

However, in most cases, the number of possible configurations is very large. This is especially the case for dynamic data folding (DDF). DDF is a technique to implement applications where some of the input data, called the parameters, changes only once in a while. Each time the parameter values change, the FPGA is reconfigured with a configuration that is specialized for the new parameter values. It's easy to see that the number of possible configurations grows exponentially with the number of parameter bits. In realistic situations, this makes it impossible to store all possible configurations. An alternative is to generate the specialized configurations on demand. Yet, a conventional FPGA tool flow is too slow to be executed at runtime. Hence, DDF cannot be implemented with a conventional tool flow.

To the best of our knowledge, this article presents the first *automatic* tool flow that builds DDF implementations starting from parameterized HDL designs. These are RT-level HDL designs in which a distinction is made between regular inputs and parameter inputs. The result of the tool flow is a parameterizable configuration. This is an FPGA configuration in which some of the configuration bits are expressed as a closed-form Boolean expression of the parameters. At run-time, the required configuration is generated on-the-fly by evaluating the closed-form Boolean expressions for the new parameter values.

This article starts with a detailed description of DDF and an overview of related work (Section 2). In Section 3, we give a high-level overview of our staged mapping tool flow. The tool flow uses the same steps as a conventional tool flow: synthesis, technology mapping, place, and route. The main difference between our tool flow and the conventional tool flow lies in the technology-mapping step. Therefore, we revisit conventional technology mapping in Section 4. In Section 5, we generalize this technology mapper to obtain a new technology mapper called TMAP, which is suited for our DDF tool flow. The generation of specialized configurations by evaluating the closed-form Boolean expressions is discussed in Section 6. Further, we apply TMAP on adaptive FIR filters and ternary content-addressable memory in Section 7. Finally, we conclude in Section 8.

2. DYNAMIC DATA FOLDING

Dynamic data folding (DDF) [Foulk 1993; Wirthlin and Hutchings 1997] is a technique that can be used to implement applications where some of the input data changes only once in a while. We call these slowly changing inputs the *parameters*. The general idea of DDF in reconfigurable devices is that each time the parameters change value, the device is reconfigured with a configuration that is specialized for the new parameter values. Since specialized configurations are smaller and faster than their generic counterparts, the hope is that the complete system will be smaller and faster when using DDF.

In this work we target DDF to hybrid systems that contain at least some dynamically reconfigurable FPGA fabric and an instruction set processor (ISP). The ISP is responsible for detecting parameter changes, and in response to these events generating specialized configurations for the FPGA fabric and reconfiguring it. To enable the ISP to reconfigure the fabric, it must have an interface with the configuration memory of the FPGA fabric. Figure 1 shows a simple example of a system that can be targeted for DDF.

A DDF implementation consists of both a software procedure that runs on the ISP and an FPGA configuration that is used to configure the FPGA at startup. This FPGA configuration is called the *template configuration*. It contains all the static functionality that is needed on the FPGA and has the dynamic parts, the configuration bits that will change at run-time, blanked out. The software procedure is called the *reconfiguration procedure*. It describes how the FPGA configuration has to be adapted, given new parameter values.

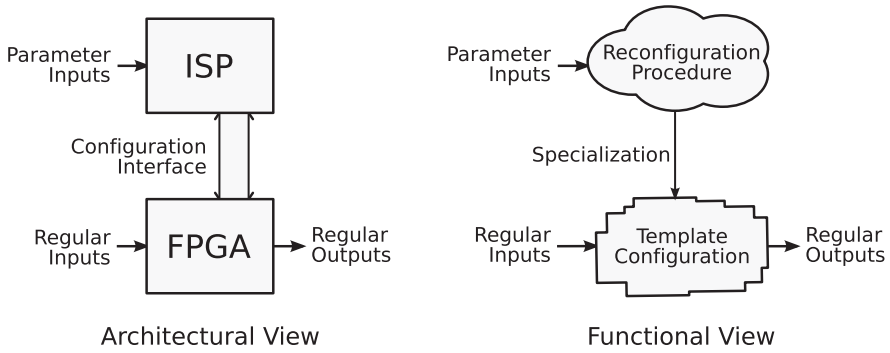


Fig. 1. Architectural and functional view of a DDF system.

Solving one specific problem using DDF requires us to execute the reconfiguration procedure and to run the specialized hardware generated during the first step. Hence, the cost of solving the problem is the sum of the costs of both steps. We call the cost of generating specialized configurations and reconfiguring the FPGA the *specialization overhead*. When comparing the cost of a DDF implementation to a generic one, we must take the specialization overhead into account.

2.1. Implementing DDF

First of all, we would like to point out that it is infeasible to implement DDF with the tool flows for dynamic reconfiguration proposed by FPGA vendors [Xilinx 2006]. These tool flows produce all specialized configurations off-line and store them in a database. At run-time they select the required configuration from the database and use it to reconfigure the FPGA. This works fine if the number of configurations is small. However, in DDF, the number of configurations grows exponentially with the number of bits needed to represent the parameter data. Hence, generating all configurations off-line and storing them in a database is infeasible for real-life applications.

Since, in most cases, the specialized configurations cannot be generated off-line, the only option is to have the reconfiguration procedure generate the configurations at run-time. A specialized configuration can be generated at run time in many Pareto-optimal ways, each with their own trade-off between the quality (area, speed, power etc.) of the resulting specialized configuration and the specialization overhead. Which option is best, depends highly on the rate at which the parameters change.

If the parameters change very fast, the use of DDF is not recommended, because the hardware does not run long enough to amortize the specialization overhead. A generic FPGA implementation which has the parameter inputs as physical inputs is, in this case, a much better option. Changing the parameter values now only involves forcing the new parameter values on the parameter inputs of the generic hardware. This does not require any computation, and thus the overhead is very small; but the quality of the FPGA implementation is low, since it is not specialized at all.

If the parameters change very slowly, a valid and obvious choice to implement the reconfiguration procedure is to use a conventional FPGA tool flow. In this case, the procedure takes the new parameter values and merges them with an HDL description of the generic functionality in order to form an HDL description for the specialized functionality. This last HDL description is then used as input for a conventional FPGA tool flow, which results in a specialized configuration. On the one hand, this leads to a specialized configuration of very high quality, because all options for optimization are available. On the other hand, running a conventional FPGA tool flow

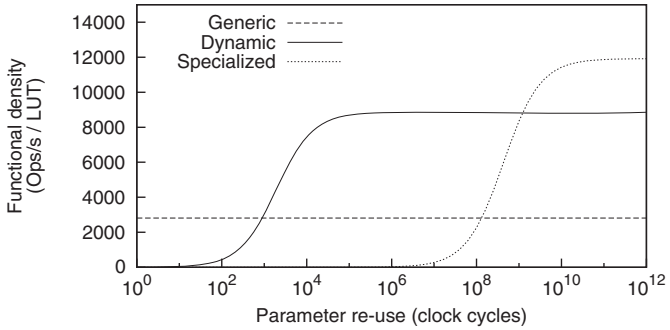


Fig. 2. Illustration of the functional density (operations per second and per number of required LUTs) as a function of the rate of change of the parameter inputs, for a number of DDF implementation techniques.

is computationally very expensive, making the specialization overhead so high that only a few applications can benefit from this type of DDF. This is, for example, the case in logic emulation [Hauck and Dehon 2007].

Figure 2 illustrates these extremes, using data from an experiment on a 16-tap, 8-bit FIR filter [Bruneel and Stroobandt 2008a]. The filter coefficients are treated as the parameter inputs. For a number of DDF implementation techniques, the functional density is plotted as a function of the rate of change of the parameters. Functional density is defined here as the number of operations that can be performed per second, including specialization overhead, per area usage (in number of LUTs). The *generic* implementation is relatively big, and thus has a low functional density, but it can be reconfigured in a single clock tick. It therefore has no specialization overhead, making its functional density independent of the rate of change of the parameters. The *specialized* circuit is 66% smaller and 29% faster, yielding a functional density (before specialization overhead) that is, over four times higher.[1] Yet, generation of a specialized circuit using a conventional FPGA tool chain took over 30 seconds. This overhead can only be amortized when the parameters stay constant for more than 10^8 iterations of the FIR filter. Also, a 30-second delay for parameter changes to take effect may not be acceptable for many applications.

To extend the applicability of DDF, many techniques have been developed that lie in between the two extreme cases described above. Each of these techniques trade some of the hardware quality for a reduction in specialization overhead. Some researchers have implemented generic tool flows that can be used for a variety of applications [Sankar and Rose 1999; Lysecky et al. 2006; Bruneel et al. 2007], but most took an application-specific hand-crafted approach: constant multiplication [Chapman 1993; Wirthlin 2004]; pattern matching [Lemoine and Merceron 1995; McGregor and Lysaght 1999]; content addressable memory (CAM) [Brelet and New 1999]. While the generic tool flows are an improvement over conventional FPGA tool flows in terms of computational effort, the improvements made in the circuits they produce are nowhere near the orders of magnitude improvement achieved in hand-crafted designs.

In contrast to other generic tool flows, the tool flow presented in this article combines a very low specialization overhead with good hardware quality. Figure 2 (*dynamic*) shows the functional density of an FIR filter implemented with our tool flow. Because this implementation requires more LUTs than the specialized implementation, its functional density is not as good for very long reuse of coefficients. However, since the generation of a specialized FIR now only takes 160 μs, the specialization overhead is

[1]On average, since the exact specialized circuit's size and clock speed depend on the parameter values.

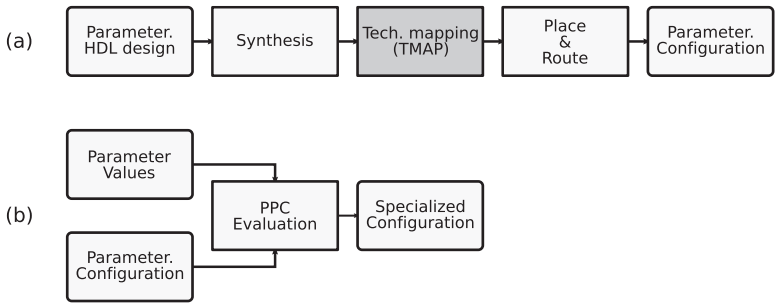


Fig. 3. Overview of our staged mapping tool-flow; (a) the generic stage; (b) the specialization stage.

already amortized once the coefficients stay constant for 10^3 iterations of the filter. Therefore, our DDF implementation has the best functional density if the periods in which the coefficients stay constant lies in the range of 10^3 to 10^9 iterations. The specialization overhead and the hardware quality are comparable to those of hand-crafted implementations. Yet, our tool flow can obtain these results in an automatic way, starting from RT-level HDL code.

3. STAGED MAPPING TOOL-FLOW

In DDF, the specification of the specialized configuration becomes available in two stages. At compile time, the generic functionality is available, but the parameter inputs are not yet bound. Only at run-time, the parameters get bound and the full specification is available. A conventional tool flow needs a full specification from the start. Therefore, the complete mapping process needs to be executed at run-time in order to generate the specialized configuration. Generating the specialized configuration from scratch each time the parameters change results in a large specialization overhead. However, since a large part of the specification (the generic functionality) is available at compile-time, we would expect that it should be possible to complete a large part of the mapping process at compile-time, which can then be refined at run-time when the parameter values become available. In this case, we would expect a large reduction in specialization overhead, since only the refinement step needs to be executed at run-time. Our tool flow uses this technique, which we call *staged mapping*. A similar concept has been used in software compilation, where it is called staged compilation. It has also been used in FPGA mapping e.g., in Derbyshire et al. [2006], where part of the synthesis process was moved from run-time to compile-time.

Figure 3 gives an overview of our tool flow. The final result, a *specialized configuration* for the FPGA, is generated in two steps or stages: the *generic stage* and the *specialization stage*. The generic functionality is presented to the generic stage in the form of a *parameterizable HDL design*, while the parameter values are only known at the beginning of the specialization stage. The generic stage produces a *parameterizable configuration* (PC). The specialization stage combines this with the parameter values to produce the specialized configuration each time the parameters change.

A *parameterizable HDL description* is an HDL description in which we make a distinction between *regular input ports* and *parameter input ports*.[2] The parameter inputs will not be inputs of the final specialized configurations; they will be bound to a constant value during the specialization stage.

[2]One should be careful not to confuse a parameter input with a generic as defined in VHDL or a parameter as defined in Verilog. A parameter input is a special kind of input port.

A PC is a function that takes parameter values as arguments and produces a specialized configuration. Since both parameters and FPGA configurations are bit vectors, the parameterizable configuration is a multioutput Boolean function. Since many of the output bits of the PC are independent of the parameter inputs, we can reduce the number of configuration bits that need to be reconfigured by splitting up the PC in a *template configuration* (TC) and a *partial parameterizable configuration* (PPC). The TC contains all static bits and is used to configure the FPGA once, when the system is started. Just like the PC, the PPC is a multioutput Boolean function. The PPC will be used by the reconfiguration procedure to generate a new partial configuration for the FPGA. In previous work [Bruneel and Stroobandt 2008a, 2008b] we have represented the PPC as a vector of closed-form single-output Boolean expressions of the parameter inputs. In this article, we represent the PPC as a Boolean network (Section 6). This enables the use of combined logic optimization, and thus leads to a more compact representation and faster evaluation.

In the parameterizable configurations generated by the tool flow presented in this work, only the truth tables of the LUTs are expressed as a function of the parameter inputs. All other configuration bits are static, and will hence be part of the TC. In other work [Bruneel and Stroobandt 2010], we have built a tool flow where the routing bits can also be expressed as a function of the parameter inputs. This tool flow can in some cases further reduce the number of FPGA resources. However, in this article we focus on the reconfiguration of LUTs.

The steps needed in the generic stage of our two-stage approach are similar to those used in conventional FPGA mapping: synthesis, technology mapping, place, and route. In Section 3.1 we will explain these algorithms in more detail. It is important to note here that these algorithms are computationally hard and so have a long run time. The specialization stage, on the other hand, generates a specialized FPGA configuration by evaluating the PPC, which is represented as a Boolean network. In Section 6 we show that the number of Boolean gates in this network scales linearly with the number of gates in the generic implementation. The specialization stage is thus not computationally hard and will run a lot faster than the generic stage. Therefore, the staged mapping tool-flow is much more efficient in generating specialized configurations than a conventional tool flow. This is because our staged flow can reuse the parameterizable configuration for each parameter value. Hence, the effort spent in the generic stage is divided over all invocations of the specialization stage. For large sets of parameter values, the average mapping effort asymptotically reaches the effort spent in the specialization stage.

3.1. Overview of the Generic Stage

The problem faced by the generic stage of our tool flow is to produce a parameterizable configuration given a parameterizable HDL description, while minimizing some cost function. Similar to conventional FPGA mapping, we divide the mapping problem into four subproblems: synthesis, technology mapping, placement, and routing. In what follows, we give an overview of these algorithms and the data structures used in our tool flow. It is important to note that, since both the input and output of the tool flow are parameterizable, the internal data structures need to be able to express parameterizability, and the algorithms that transform the data structures must preserve the parameterizability.

3.1.1. Synthesis. The synthesis step converts the parameterizable HDL description into a gate-level circuit. As we described in Section 3, a parameterizable HDL description distinguishes regular inputs from parameter inputs. This distinction has to be preserved in the gate-level circuits. This can easily be done by allowing both types of

```
entity multiplexer is
port (
  I   : in    std_logic_vector(3 downto 0);
  S   : in    std_logic_vector(1 downto 0); --PARAM
  O   : out   std_logic
);
end multiplexer;

architecture behavior of multiplexer is
begin
  O <= I(conv_integer(S));
end behavior;
```

Fig. 4. The parameterizable VHDL code of a 4-to-1 multiplexer where the select inputs are marked as parameters.

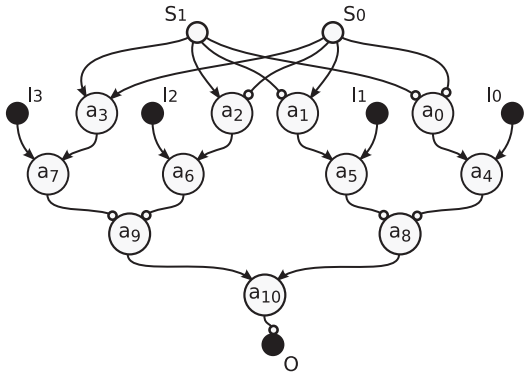


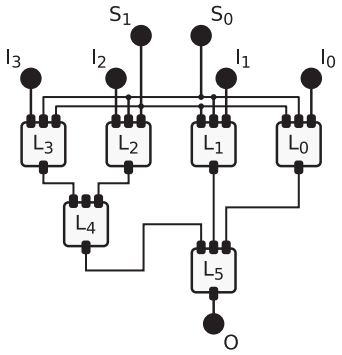
Fig. 5. AIG of a 4-to-1 multiplexer. Large circles represent internal nodes; smaller circles represent combinational inputs and output; inverted edges have a circular arrowhead, while noninverted edges have a standard arrowhead.

inputs in the gate-level circuit data structure. The synthesis tool simply has to pass on this information about the inputs.

In our tool-flow implementation, the distinction between regular inputs and parameter inputs is made by adding the comment line annotation `--PARAM` at the end of the input declaration in the entity declaration. Figure 4 shows the parameterizable VHDL code of a 4-to-1 multiplexer where the select inputs are marked as parameters. We will use this multiplexer example, throughout this article. Processing this code with a slightly adjusted synthesis tool results in the gate-level circuit, which is represented as a graph in Figure 5. In this figure, regular inputs are represented by filled circles (I_0, I_1, I_2, and I_3), while the parameter inputs (S_0 and S_1) are not filled.

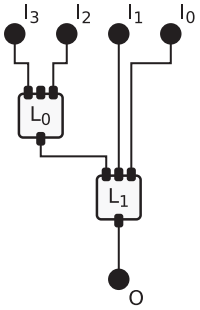
3.1.2. Technology Mapping. During technology mapping, the gate-level circuit produced by the synthesis step is mapped onto the resources available in the target FPGA architecture, while minimizing a cost function.

The result of a conventional technology mapper is a circuit containing LUTs and flip-flops connected by nets. Each LUT is associated to a fixed functionality represented by a static truth table. The result of mapping the gate-level circuit in Figure 5 to 3-input LUTs is shown in Figure 6. In Section 4 we review the conventional LUT mapping process.



Entry	L_0	L_1	L_2	L_3	L_4	L_5
000	0	0	0	0	1	1
001	1	0	0	0	0	1
010	0	0	0	0	1	1
011	0	1	0	0	0	1
100	0	0	0	0	0	0
101	0	0	1	0	0	1
110	0	0	0	0	0	1
111	0	0	0	1	0	1

Fig. 6. The final result of mapping the AIG represented in Figure 5 with a conventional mapper ($K = 3$). The LUT structure (left) and associated truth tables (right).



Entry	L_0	L_1
000	1	1
001	$\overline{S_1} + S_0$	1
010	1	1
011	$\overline{S_1} + S_0$	1
100	$\overline{S_1 S_0}$	0
101	$\overline{S_1}$	$\overline{S_1 + S_0}$
110	$\overline{S_1 S_0}$	$\overline{S_1} S_0$
111	$\overline{S_1}$	$\overline{S_1}$

Fig. 7. The final result of mapping the AIG represented in Figure 4 with a TLUT mapper (K = 3). The LUT structure (left) and associated truth tables (right).

Since we want to express the configuration memory of the FPGA as a Boolean function of the parameter inputs, our technology mapper will need to map to LUTs with truth tables that are expressed as a function of the parameter inputs. We call these LUTs *tunable LUTs* or *TLUTs*. A circuit containing TLUTs instead of regular LUTs is called a TLUT circuit. A TLUT is an abstract element that represents a regular LUT from which the truth table will be reconfigured upon a parameter change. It also shows how the truth table can be calculated given the new parameter value. In the final DDF implementation, the functionality represented by a TLUT will be partially implemented by the ISP and partially by a physical LUT on the FPGA. The ISP will be responsible for calculating a new static truth table for the physical LUT upon a parameter change, and the physical LUT implements the specialized Boolean function represented by that truth table in between parameter changes. In what follows, we will use the term TLUT to refer to both the abstract element and the physical LUT that partially implements it.

Figure 7 shows a possible TLUT circuit for our 4-to-1 multiplexer example. In Section 5, we explain in detail how to extend the conventional LUT mapping process to tunable LUT mapping. The result is an algorithm called TMAP.

3.1.3. Placement and Routing. A placement tool associates every abstract LUT in the input LUT circuit to a physical LUT on the FPGA while minimizing the routing cost. The routing tool calculates which of the switches in the configurable interconnect should be closed and which should be opened in order to realize the connections represented by the nets in the input LUT circuit. Since placement and routing do not depend on the

truth tables of the LUTs, but only on the topology of the input circuit, the conventional tools can be adopted without change to place and route TLUT circuits. We will therefore focus the remainder of this article on the mapping step. In Bruneel et al. [2009], we explain in detail how TMAP can be incorporated in the Xilinx tool-flow.

4. BACKGROUND: CONVENTIONAL MAPPING

In this section we review the basics of LUT mapping. We first give some definitions and introduce the data structures that are used in technology mapping (Section 4.1); followed by the problem definition (Section 4.2), and a description of a simple structural mapping algorithm (Section 4.3). The information in this section will be used in Section 5 to construct an efficient TLUT mapping algorithm.

4.1. Definitions and Data Structures

The input of our structural mapper is the combinational part of the circuit produced by the synthesis step. It is represented as an and-inverter graph (AIG) [Kuehlmann et al. 2002]. This is a directed acyclic graph (DAG) $G = (N, E)$. A node in the graph $n \in N$ represents a two-input AND gate, a combinational input (CI) or a combinational output (CO). A directed edge in the graph $(u \in N, v \in N, i \in \{0, 1\}) \in E$, represents a connection in the logic circuit with an input of gate v as sink and the output of gate u as source; i indicates whether the connection is inverted (1) or not (0).

Figure 5 depicts the AIG for the 4-to-1 multiplexer example. The internal nodes that represent two-input AND gates are represented by large circles. The smaller circles represent the CIs and COs of the AIG. Inverted edges have small circles as arrowheads, while noninverted edges, have a standard arrowhead. The AIG shown in the figure is not the minimal size AIG for a 4-to-1 multiplexer, but it is better suited to illustrate the concepts and reasoning in what follows.

A cone of node n, C_n, is a subgraph of the AIG consisting of n and some of its predecessors that are not CIs, such that any node $u \in C_n$ has a path to n that lies entirely in C_n [Manohararajah et al. 2006]. The set of input edges, $iedge(C_n)$ of a cone C_n, is the set of edges with a head in C_n and tail outside C_n, and the set of output edges $oedge(C_n)$ is the set of edges with n as a tail. The set of input nodes $inode(C_n)$ of a cone C_n, is the set of distinct nodes that are the tails of $iedge(C_n)$. A cone C_n is called K-feasible if the number of input nodes $|inode(C_n)|$ is smaller than or equal to K.

A cone C_n of a node n is uniquely specified by the ordered pair $(n, inode(C_n))$; $inode(C_n)$ is called the cut of cone C_n. When n and $inode(C_n)$ are given, the nodes that are part of C_n can be found by backtracking from n towards the nodes in $inode(C_n)$. All the nodes that are visited during this process are part of C_n.

In Figure 5, the subgraph consisting of nodes a_8 and a_{10} is a cone of node a_{10}. The input edges of this cone are $(a_4, a_8, 1)$, $(a_5, a_8, 1)$, and $(a_9, a_{10}, 0)$. The output edge of the cone is $(a_{10}, O, 1)$. The cut of this cone is $\{a_4, a_5, a_9\}$. This cone of node a_{10} is uniquely specified by the ordered pair $(a_{10}, \{a_4, a_5, a_9\})$. Because the number of its input nodes is smaller than or equal to 3, the cone is 3-feasible.

4.2. Problem Definition

During technology mapping, the gate-level circuit produced by the synthesis step is mapped on the resources (K-LUTs) available in the target FPGA architecture.

Since a K-input LUT can implement any Boolean function with up to K arguments, the conventional structural mapping problem reduces to selecting a set of K-feasible cones to cover the input graph. This means that every edge of the input DAG lies entirely within one of the selected cones or is an output edge of one of the selected cones [Manohararajah et al. 2006]. There are many possible coverings for each input DAG. The task of the mapper is to find a covering with near to minimum cost. In

Table I. Intermediate results of mapping AIG in Figure 5 to an FPGA architecture containing 3-input LUTs with a conventional mapper.

Node n	Cut set $\Phi(n)$	Best cut $bc(n)$	Depth $depth(bc(n))$	Area flow $af(bc(n))$
a_0	$\{\{a_0\}, \{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_1	$\{\{a_1\}, \{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_2	$\{\{a_2\}, \{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_3	$\{\{a_3\}, \{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_4	$\{\{a_4\}, \{a_0I_0\}, \{I_0S_0S_1\}\}$	$\{I_0S_0S_1\}$	1	1
a_5	$\{\{a_5\}, \{a_1I_1\}, \{I_1S_0S_1\}\}$	$\{I_1S_0S_1\}$	1	1
a_6	$\{\{a_6\}, \{a_2I_2\}, \{I_2S_0S_1\}\}$	$\{I_2S_0S_1\}$	1	1
a_7	$\{\{a_7\}, \{a_3I_3\}, \{I_3S_0S_1\}\}$	$\{I_3S_0S_1\}$	1	1
a_8	$\{\{a_8\}, \{a_4a_5\}, \{a_0a_5I_0\}, \{a_1a_4I_1\}\}$	$\{a_4a_5\}$	2	3
a_9	$\{\{a_9\}, \{a_6a_7\}, \{a_2a_7I_2\}, \{a_3a_6I_3\}\}$	$\{a_6a_7\}$	2	3
a_{10}	$\{\{a_{10}\}, \{a_8a_9\}, \{a_4a_5a_9\}, \{a_6a_7a_8\}\}$	$\{a_4a_5a_9\}$	3	6

depth-oriented mapping, this cost is the longest path through the cones of the covering. In area-oriented mapping, the cost is the number of cones in the covering. In the rest of this article we will focus on depth-oriented mapping, but the techniques described can be easily extended to other mapping criteria.

4.3. The Algorithm

Just like most structural mapping algorithms [Hauck and Dehon 2007], the mapper described here is based on a dynamic programming technique. These algorithms typically consist of the following steps:

cone enumeration. Calculates all K-feasible cones of the input circuit;
cone ranking. Selects the lowest-cost cone for each node;
cone selection. Selects a subset of the best cones to form the final covering; and
mapping solution generation. Generates the LUT structure and the truth tables.

4.3.1. Cone Enumeration. During cone enumeration, all K-feasible cones of every node in the DAG are enumerated. This is done with a dynamic programming algorithm [Pan and Lin 1998] that traverses all nodes of the input circuit in topological order, starting from the CIs and going towards the COs.

The set of K-feasible cones of node n is denoted $\zeta(n)$. Since all cones in $\zeta(n)$ have the same root, it is not efficient to store $\zeta(n)$ as a set of cones. Hence we store $\zeta(n)$ as an ordered pair $(n, \Phi(n))$, where $\Phi(n)$ is the cut set of the cones.

$\Phi(n)$ is generated by combining the cut sets of n's left and right predecessors (n_l and n_r), adding the trivial cut and retaining only the K-feasible cuts. The trivial cut of a node n is equal to $\{n\}$. The cone enumeration process is formally represented in Equations (1) and (2);

$$\Phi(n) = \begin{cases} \{\{n\}\} & \text{if } n \in \text{CI} \\ \{\{n\}\} \cup M(n) & \text{otherwise.} \end{cases} \tag{1}$$

$$M(n) = \{c_l \cup c_r | c_l \in \Phi(n_l), c_r \in \Phi(n_r), |c_l \cup c_r| \leq K\}. \tag{2}$$

The intermediate results of applying conventional structural mapping to the AIG, shown in Figure 5, to an FPGA architecture containing 3-input LUTs are given in Table I,[3] The second column shows the cuts of all 3-feasible cones found by the cone enumeration process described above.

[3]In the remainder of this article we assume, for the examples, that we have an FPGA architecture with 3-input LUTs, for the sake of an easier explaination and without loss of generality.

4.3.2. Cone Ranking. In the cone-ranking step, the nodes are again processed in topological order from the CIs to the COs. For each visited node n, the best nontrivial cone $bc(n)$ is selected from $\Phi(n)$ according to the mapping, criterion. In depth-oriented mapping, the cone with the lowest depth (Eq. (3)) is selected. If several cones have the same depth, the one with the lowest area flow (Eq. (4)) is selected. This mapping criterion leads to a depth, optimal mapping solution [Cong and Ding 1994].

$$depth(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \\ 1 + \max\limits_{u \in inode(C_n)} (depth(bc(u))) & \text{otherwise.} \end{cases} \tag{3}$$

$$af(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \\ 1 + \sum\limits_{u \in inode(C_n)} \frac{af(bc(u))}{|oedge(u)|} & \text{otherwise.} \end{cases} \tag{4}$$

The best cones and their associated depth and area flows for each of the nodes in the multiplexer example of Figure 5 are shown, respectively, in the third, the fourth, and the fifth columns of Table I.

4.3.3. Cone Selection. In the cone selection step, a subset of the best cones is selected as the final covering of the graph. This is done by traversing the nodes of the graph in topological order, but now starting from the COs and moving towards the CIs. First, the best cones of the nodes driving the COs are selected and then the best cones of the input nodes, $inode(.)$, of the selected cones are added until the CIs are reached. The selected cones form a cone covering of the input circuit. The selected cones for our example are $(a_{10}, \{a_4 a_5 a_9\})$, $(a_9, \{a_6 a_7\})$, $(a_7, \{I_3 S_0 S_1\})$, $(a_6, \{I_2 S_0 S_1\})$, $(a_5, \{I_1 S_0 S_1\})$, and $(a_4, \{I_0 S_0 S_1\})$.

4.3.4. Generating the Mapping Solution. During this last step, the LUT structure and the truth tables for the LUTs are generated. The LUT structure is found by introducing one LUT for each of the cones in the covering, and connecting these LUTs in the same way that their corresponding cones are connected. The truth tables are found by analyzing the functionality of the cone. An entry in the truth table can be found by evaluating the functionality for the entry's corresponding input value. The LUT structure for our example is found in Figure 6, together with the truth tables for each of the LUTs.

5. TMAP: TUNABLE LUT MAPPING

In this section we describe TMAP, a structural mapping algorithm that is capable of mapping the combinational part of a Boolean circuit, in which a distinction is made between regular inputs (RI) and parameter inputs (PI), to Tunable LUTs or TLUTs. Again, we use the 4-to-1 multiplexer of Figure 5 as an example, but now the select inputs of the multiplexer (S_0 and S_1) are parameter inputs. In the figure, regular inputs are filled circles, while parameter inputs are not filled.

We start by describing an obvious, but naive, implementation in Section 5.1. The main problem with the naive implementation is that it considers many uninteresting or redundant cones, which can lead to very long run times. In Section 5.2, we identify these redundant cones, called *incomplete cones*, and describe how to optimize the algorithm so that it avoids enumerating these cones. An interesting side effect of not considering incomplete cones is that all remaining cones, called the *complete cones*, can be uniquely specified by a subset of their input node set. We call this subset the *reduced cut* of the cone (Section 5.3). As will be explained, using reduced cuts decreases the complexity of operations on cones and reduces the memory needed to store a cone. In Section 5.4, we show that the final optimized algorithm has the same computational complexity as a conventional LUT mapper.

Table II. Intermediate results of mapping AIG in Figure 5 to an FPGA architecture containing 3-input LUTs with a naive implementation of a TLUT mapper.

Node n	Cut set $\Phi(n)$	Best cut $bc(n)$	Depth $depth(bc(n))$	Area flow $af(bc(n))$
a_0	$\{\{a_0\}, \{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_1	$\{\{a_1\}, \{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_2	$\{\{a_2\}, \{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_3	$\{\{a_3\}, \{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_4	$\{\{a_4\}, \{a_0I_0\}, \{I_0S_0S_1\}\}$	$\{I_0S_0S_1\}$	1	1
a_5	$\{\{a_5\}, \{a_1I_1\}, \{I_1S_0S_1\}\}$	$\{I_1S_0S_1\}$	1	1
a_6	$\{\{a_6\}, \{a_2I_2\}, \{I_2S_0S_1\}\}$	$\{I_2S_0S_1\}$	1	1
a_7	$\{\{a_7\}, \{a_3I_3\}, \{I_3S_0S_1\}\}$	$\{I_3S_0S_1\}$	1	1
a_8	$\{\{a_8\}, \{a_4a_5\}, \{a_0a_5I_0\}, \{a_1a_4I_1\}, \{a_4I_1S_0S_1\}, \{a_5I_0S_0S_1\}, \{a_0I_0I_1S_0S_1\}, \{a_1I_0I_1S_0S_1\}, \{I_0I_1S_0S_1\}\}$	$\{I_0I_1S_0S_1\}$	1	1
a_9	$\{\{a_9\}, \{a_6a_7\}, \{a_2a_7I_2\}, \{a_3a_6I_3\}, \{a_6I_3S_0S_1\}, \{a_7I_2S_0S_1\}, \{a_2I_2I_3S_0S_1\}, \{a_3I_2I_3S_0S_1\}, \{I_2I_3S_0S_1\}\}$	$\{I_2I_3S_0S_1\}$	1	1
a_{10}	$\{\{a_{10}\}, \{a_8a_9\}, \{a_4a_5a_9\}, \{a_6a_7a_8\}, \{a_4a_9I_1S_0S_1\}, \{a_6a_8I_3S_0S_1\}, \{a_5a_9I_0S_0S_1\}, \{a_7a_8I_2S_0S_1\}, \{a_9I_0I_1S_0S_1\}, \{a_8I_2I_3S_0S_1\}\}$	$\{a_9I_0I_1S_0S_1\}$	2	2

In the description that follows, we distinguish two types of internal nodes of an AIG: regular nodes and parameter nodes. A *parameter node* is an internal node of the AIG of which the fan-in cone is driven only by parameter inputs. The set of parameter nodes of the input AIG is denoted as PN. In our example, a_0, a_1, a_2, and a_3 are parameter nodes, all other internal nodes are regular nodes, denoted as RN.

5.1. The Naive Implementation

As discussed in Section 3.1.2, a TLUT is a LUT where the truth table is expressed as a Boolean function of the parameter inputs. It is easy to see that such a TLUT can implement any Boolean function with up to K regular signals and any number of parameter signals as its arguments (as the parameter signals only define the truth table entries). In the context of structural mapping, this means that a TLUT can implement the functionality represented by a cone with any number of parameter input nodes, and up to K nonparameter input nodes, as its cut. We call such cones K^**-feasible*. The TLUT mapping problem thus reduces to finding a minimum cost covering of the input AIG with K^*-feasible cones. As the mapping criterion, we will again try to minimize the depth of the LUT structure that will be implemented on the FPGA.

The structural mapping algorithm described in Section 4 can easily be transformed to meet the requirements described in the previous paragraph. Only the cone enumeration step needs to be adapted so that all K^*-feasible cones are enumerated instead of all K-feasible cones. This can easily be done by substituting Eq. (2) by Eq. (5).

$$M(n) = \{c_l \cup c_r | c_l \in \Phi(n_l), c_r \in \Phi(n_r), |(c_l \cup c_r) \setminus PI| \leq K\}. \tag{5}$$

The only difference is that instead of retaining the K-feasible cones, we now retain the K^*-feasible cones after merging the input cone sets of the predecessor nodes of n. This naive implementation was first described in Bruneel and Stroobandt [2008b].

The intermediate results of applying the new mapping process on the multiplexer example are given in Table II. Again we assume the LUTs in the target FPGA fabric to be three-input LUTs. The second column shows the input node sets of the 3^*-feasible cones for each node.

The cone-ranking step (Section 4.3.2) and cone-selection step (Section 4.3.3) can be adopted without change. The results of the cone-ranking step are shown in Table II.

The best cone, its depth, and its area flow are shown respectively in columns three, four, and five. The cones $(a_{10}, \{a_9 I_0 I_1 S_0 S_1\})$ and $(a_9, \{I_2 I_3 S_0 S_1\})$ are selected as the covering by the cone selection step. The LUT structure for the multiplexer example can be found in Figure 7, together with the truth tables for each of the LUTs, expressed as a Boolean function of the parameter inputs. The way these Boolean functions are derived is explained in detail in Section 6.

5.2. Incomplete Cones

An *incomplete cone* C_n has one or more parameter nodes in its cut. In our example, the cone $C_{a_8} = (a_8, \{a_0 a_5 I_0\})$ $(depth(C_{a_8}) = 2; af(C_{a_8}) = 3)$ is an incomplete cone. The cut of this cone contains the parameter node a_0. Incomplete cones are redundant because there is always another K^*-feasible cone C'_n with a better rank. The cone C'_n can be constructed by simply adding the parameter nodes and all their predecessor nodes to cone C_n. If we do this for cone C_{a_8} we find cone $C'_{a_8} = (a_8, \{a_5 I_0 S_0 S_1\})$ $(depth(C'_{a_8}) = 2; af(C'_{a_8}) = 2)$ by adding node a_0 to C_{a_8}.

More formally, if $p \in inode(C_n)$ is a parameter node that depends on parameter inputs P, the cone C'_n with $inode(C'_n) = (inode(C_n) \setminus p) \cup P$ has a higher rank than C'_n. The depth (Eq. (3)) of C'_n is less than or equal to the depth of C_n because the depth of the parameter inputs, P, is zero and the maximum over a subset, $inode(C_n) \setminus p$, is always less than or equal to the maximum over the complete set. The area flow (Eq. (4)) of C'_n is less than the area flow of C_n because $af(C'_n) = af(C_n) - af(bc(p))$ and since p is an internal node $af(bc(p)) \geq 1$.

The enumeration of incomplete cones can be avoided by not adding the trivial cut $\{p\}$ for parameter nodes p during the cone enumeration process, therefore p can never become part of an enumerated cut. The naive algorithm described in Section 5.1 enumerates 47 cones for the multiplexer example. By not adding the trivial cut for the parameter nodes a_0, a_1, a_2, and a_3, we avoid enumerating 16 incomplete cones. It is easy to think of real-life circuits where the number of incomplete cones exceeds by far the number of complete cones.

5.3. Reduced Cuts

An interesting thing to note is that every complete cone C_n is uniquely specified by the ordered pair $(n, rinode(C_n))$, where n is the root of the cone and $rinode(C_n)$ is the set of regular (nonparameter) input nodes of the cone. We call $rinode(C_n)$ the *reduced cut* of cone C_n. The full cut C_n can easily be found by backtracking, starting from the root node n and stopping at nodes that are in the set $rinode(C_n) \cup PI$. Because a complete cone is fully specified by its root and its reduced cut, we don't need to store the full cut of a K^*-feasible cone. Representing a cone using its reduced cut therefore makes the algorithm more memory-efficient.

During cone enumeration, the cut of a cone C_n rooted at a node n, with predecessors n_l and n_r, is calculated by taking the union of the cut of a cone C_{n_l} rooted at n_l and the cut of a cone C_{n_r} rooted at n_r (Eq. (5)). We can easily see that the reduced cut of cone C_n can be calculated by taking the union of the reduced cuts of C_{n_l} and C_{n_r}. Therefore, the cone enumeration process can be executed without ever needing to use costly backtracking in order to calculate the full cut of a cone. Taking the union of two reduced cuts is less expensive than taking the union of the full cuts, since the size of a reduced cut is always smaller than or equal to the size of the corresponding full cut. Also note that the size of the reduced cut is limited by K, while full cuts are limited by $K + |PI|$, which depends on the input circuit.

Since the fan-in cone of a parameter node only contains other parameter nodes, there exists only one complete cone for each parameter node. The reduced cut of each of these

Table III. Intermediate results of mapping AIG in Figure 5 to an FPGA architecture containing 3-input LUTs with an optimized implementation of a TLUT mapper that makes use of reduced cuts.

Node n	Reduced cut set $\Phi^r(n)$	Best cut $bc(n)$	Depth $depth(bc(n))$	Area flow $af(bc(n))$
$a_0 \dots a_3$	$\{\{\}\}$			
a_4	$\{\{a_4\}, \{I_0\}\}$	$\{I_0\}$	1	1
a_5	$\{\{a_5\}, \{I_1\}\}$	$\{I_1\}$	1	1
a_6	$\{\{a_6\}, \{I_2\}\}$	$\{I_2\}$	1	1
a_7	$\{\{a_7\}, \{I_3\}\}$	$\{I_3\}$	1	1
a_8	$\{\{a_8\}, \{a_4a_5\}, \{a_4I_1\}, \{a_5I_0\}, \{I_0I_1\}\}$	$\{I_0I_1\}$	1	1
a_9	$\{\{a_9\}, \{a_6a_7\}, \{a_6I_3\}, \{a_7I_2\}, \{I_2I_3\}\}$	$\{I_2I_3\}$	1	1
a_{10}	$\{\{a_{10}\}, \{a_8a_9\}, \{a_4a_5a_9\}, \{a_6a_7a_8\}, \{a_4a_9I_1\}, \{a_6a_8I_3\}, \{a_5a_9I_0\}, \{a_7a_8I_2\}, \{a_9I_0I_1\}, \{a_8I_2I_3\}\}$	$\{a_9I_0I_1\}$	2	2

cones is the empty set. This means that we know the reduced cut set of a parameter node beforehand, and thus don't need to visit the parameter nodes during cone enumeration. Since the fan-in cone of a nonparameter node is driven by at least one regular input, nonparameter nodes can be visited in topological order by first labeling the parameter (PN) nodes as visited, visiting the regular inputs (RI), and then recursively visiting internal nodes, of which both predecessors are visited.

Given the above information, we can construct a new dynamic programming algorithm that enumerates all complete cones of the input AIG. The algorithm traverses all nonparameter nodes of the input circuit in topological order and generates the reduced cut sets $\Phi^r(n)$ of the complete K^*-feasible cones $\zeta^*(n)$ by combining the reduced cut sets of n's predecessors n_l and n_r. The new cone enumeration process is represented by Eq. (6) and (7).

$$\Phi^r(n) = \begin{cases} \{\{\}\} & \text{if } n \in \text{PN} \\ \{\{n\}\} & \text{if } n \in \text{RI} \\ \{\{n\}\} \cup M^r(n) & \text{otherwise.} \end{cases} \tag{6}$$

$$M^r(n) = \{c_l \cup c_r | c_l \in \Phi^r(n_l), c_r \in \Phi^r(n_r), |c_l \cup c_r| \leq K\}. \tag{7}$$

The result of applying this cone enumeration step on the example of Figure 5 can be found in column 2 of Table III.

In Section 5.1, Eq. (3) and (4) are used to calculate the rank of a cone. The problem with these equations is that they need the full cut of a cone in order to calculate its depth and area flow. This requires the cone-ranking step to use backtracking to generate the full cut for every enumerated cone, which would have a negative impact on the run-time of TMAP. However, a closer look shows that there is a much better solution. The backtracking process will only find parameter inputs in addition to the nodes in the reduced cut. As can be seen from Eqs. (3) and (4), the depth and the area flow of a parameter input are both zero. Since the maximum and the sum over a set of positive numbers does not change by adding zeros to that set, the depth and the area flow of a cone are not influenced by its parameter inputs. We can thus substitute Eqs. (3) and (4) by Eqs. (8) and (9), respectively. From these equations it is clear that the reduced cuts contain sufficient information to determine their rank, making backtracking unnecessary. The best cone, its depth and its area flow for each of the nodes in the example circuit can be found in columns 3, 4, and 5 of Table III, respectively.

$$depth(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \cup \text{PI} \\ 1 + \max\limits_{u \in rinode(C_n)} (depth(bc(u))) & \text{otherwise.} \end{cases} \tag{8}$$

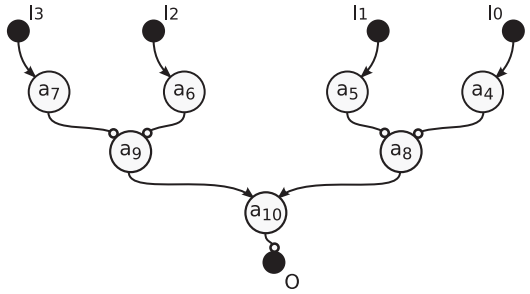


Fig. 8. The reduced circuit of the 4-to-1 multiplexer AIG represented in Figure 5.

$$af(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \cup \text{PI} \\ 1 + \sum_{u \in rinode(C_n)} \frac{af(bc(u))}{|oedge(u)|} & \text{otherwise} \end{cases} \tag{9}$$

It's easy to see that the cone selection step can also be completed without the need of the full cuts of the enumerated cones. The need for backtracking to generate the full cut of a cone is therefore postponed until after the final cover is selected. The number of selected cones is only a fraction of the total number of enumerated cones. Therefore, the impact of backtracking on the total run-time is limited.

The reduced cuts $(a_{10}, \{a_9 I_0 I_1\})$ and $(a_9, \{I_2 I_3\})$ are found in our example. After backtracking, this results in cones $(a_{10}, \{a_9 I_0 I_1 S_0 S_1\})$ and $(a_9, \{I_2 I_3 S_0 S_1\})$. This result is exactly the same as for the naive implementation. The TLUT circuit is thus the same as in Figure 7.

5.4. Scalability

Just like in conventional mappers [Manohararajah et al. 2006], most of the execution time of TMAP is spent in the cone enumeration step. It is easy to see that this time is proportional to the total number of enumerated cones. The enumerated cones of a node n are not just the K-feasible cones, but all cones that are considered as, K-feasible cones for node n during the cone enumeration process. The enumerated cones of a node n are denoted $\mathscr{E}(n)$ for the conventional mapper and $\mathscr{E}^r(n)$ for the reduced cut mapper. Their formal definitions are given in Eq. (10) and Eq. (11), respectively.

$$\mathscr{E}(n) = \begin{cases} \{\{n\}\} & \text{if } n \in \text{CI} \\ \{\{n\}\} \cup \{c_l \cup c_r | c_l \in \Phi(n_l), c_r \in \Phi(n_r)\} & \text{otherwise.} \end{cases} \tag{10}$$

$$\mathscr{E}^r(n) = \begin{cases} \{\{\}\} & \text{if } n \in \text{PN} \\ \{\{n\}\} & \text{if } n \in \text{RI} \\ \{\{n\}\} \cup \{c_l \cup c_r | c_l \in \Phi^r(n_l), c_r \in \Phi^r(n_r)\} & \text{otherwise.} \end{cases} \tag{11}$$

Since the number of enumerated cones is highly dependent on the topology of the input circuit, the complexity analysis is very difficult. However, we will show that, for the worst-case circuits, the number of cones enumerated by TMAP is of the same order as the number of cones enumerated by a conventional mapper for a reduced version of that same input circuit. The computational complexity of TMAP is therefore of the same order as the computational complexity of a conventional technology mapper.

The reduced circuit is derived from the original circuit by simply removing the parameter nodes. The reduced circuit of our multiplexer example input circuit is depicted in Figure 8. In Theorem 5.1, we show that when we apply the conventional cone enumeration algorithm, described in Section 4, to the reduced input circuit, the K-feasible

cones found for a node n^* of the reduced circuit are exactly the same as the reduced K^*-feasible cones found for the corresponding node n when the optimized cone enumeration algorithm, described in Section 5.3, is applied to the original input circuit. Given Theorem 5.1, we can easily prove Theorem 5.2, which proves the same property, but for the enumerated cones instead of the K-feasible cones.

THEOREM 5.1. *If $A^* = (N^*, E^*)$ is the DAG that represents the reduced circuit, $A = (N, E)$ is the DAG of the original circuit, and node $n^* \in N^*$ corresponds to node $n \in N$, then*

$$\Phi(n^*) = (\Phi^r(n))^* \quad \forall n^* \in N^*, \tag{12}$$

where $(\cdot)^*$ replaces the nodes n in the cut set by there corresponding node n^*.

PROOF. By induction on n^*. The *base case*, $n^* \in CI^*$, where CI^* is the set of all combinational inputs of N^* can be proven as follows.

$$\Phi(n^*) = \{\{n^*\}\} = (\{\{n\}\})^* = (\Phi^r(n))^*. \tag{13}$$

The first step directly follows from Eq. (1) for combinational inputs. The second step is valid because of the definition of operator $(\cdot)^*$. Since every combinational input in A^* corresponds to a regular input in A, the last step follows directly from Eq. (6).

For the *inductive step* we prove two cases: for nodes n^* with one predecessor (n_r^*) and nodes n^* with two predecessors (n_l^* and n_r^*). In both cases, we assume that Eq. (12) holds for the predecessor(s) of n^*. For the proof of the single predecessor case, we start from the right-hand side of Eq. (12).

$$\begin{aligned}
(\Phi^r(n))^* &= (\{\{n\}\} \cup \{c_{par} \cup c_r | c_{par} \in \Phi^r(n_{par}), c_r \in \Phi^r(n_r), |c_{par} \cup c_r| \leq K\})^* \\
&= (\{\{n\}\} \cup \{c_{par} \cup c_r | c_{par} = \{\}, c_r \in \Phi^r(n_r), |\{\} \cup c_r| \leq K\})^* \\
&= (\{\{n\}\} \cup \Phi^r(n_r))^* \\
&= \{\{n^*\}\} \cup (\Phi^r(n_r))^* \\
&= \{\{n^*\}\} \cup \Phi(n_r^*) = \Phi(n^*).
\end{aligned}$$

The first step follows from Eqs. (6) and (7) and from the fact that in the case of a single predecessor for node n^*, its corresponding node n has two predecessors: a parameter node n_{par} and a regular node n_r. The second step uses Eq. (6) to substitute the cut set of the parameter node n_{par} with $\{\{\}\}$. The third step is justified because the union of a set c_r and the empty set is equal to the set c_r itself, and that a cardinality of a reduced K^*-feasible cut is at most K by definition. The fourth step makes use of the definition of the $(\cdot)^*$ operator; the fifth step uses the induction hypotheses; and the sixth step uses Eqs. (1) and (2) for a single predecessor node.

For the proof of the two predecessors case, we start from the left-hand side of Eq. (12).

$$\begin{aligned}
\Phi(n^*) &= \{\{n^*\}\} \cup \{c_l \cup c_r | c_l \in \Phi(n_l^*), c_r \in \Phi(n_r^*), |c_l \cup c_r| \leq K\} \\
&= \{\{n^*\}\} \cup \{c_l \cup c_r | c_l \in (\Phi^r(n_l))^*, c_r \in (\Phi^r(n_r))^*, |c_l \cup c_r| \leq K\} \\
&= (\{\{n\}\} \cup \{c_l \cup c_r | c_l \in \Phi^r(n_l), c_r \in \Phi^r(n_r), |c_l \cup c_r| \leq K\})^* \\
&= (\Phi^r(n))^*.
\end{aligned}$$

The first step directly follows from Eqs. (1) and (2). In the second step, the induction hypothesis is used. The third step makes use of the definition of operator $(\cdot)^*$. In the final step, Eqs. (6) and (7) are used. □

THEOREM 5.2. *If $A^* = (N^*, E^*)$ is the DAG that represents the reduced circuit, $A = (N, E)$ is the DAG of the original circuit, and node $n^* \in N^*$ corresponds to node $n \in N$,*

then

$$\mathscr{E}(n^*) = (\mathscr{E}^r(n))^* \quad \forall n^* \in N^*, \tag{14}$$

where $(\cdot)^*$ *replaces the nodes n in the cut set by their corresponding node* n^*.

PROOF. The proof follows directly from the definitions of $\mathscr{E}(n)$ (Eq. (10)), $\mathscr{E}^r(n)$ (Eq. (11)), and Theorem 5.1. □

Using Theorem 5.2, we can show that the number of cones enumerated when the reduced cut TMAP algorithm (Section 5.3) is applied to the original circuit is equal to the sum of the number of cones enumerated when the conventional mapping algorithm is applied to the reduced circuit and the number of parameter nodes $|PN|$.

$$\sum_{\forall n \in N} |\mathscr{E}^r(n)| = \sum_{\forall n \in (N \setminus PN)} |\mathscr{E}^r(n)| + \sum_{\forall n \in (PN \subset N)} |\mathscr{E}^r(n)| \tag{15}$$

$$= \sum_{\forall n^* \in N^*} |\mathscr{E}(n^*)| + |PN|. \tag{16}$$

Which of the two terms in Eq. (16) gets the upper hand, depends on the type of circuit. In the extreme case, when all nodes of the input circuit are parameter nodes, the second term of Eq. (16) is equal to $|N|$ and the first term equals zero. This means that in the worst case, the second term is linear in the size of the input circuit. At the other extreme, when none of the nodes of the input circuit are parameter nodes, the TLUT mapping problem reduces to a conventional mapping problem, and thus the second term becomes zero. We know that the number of cones enumerated by a conventional mapper will grow faster than linear for a large group of circuits. For the worst-case circuits, we can ignore the second term when the circuit size grows. Formally, we can write:

$$O\left(\sum_{\forall n \in N} |\mathscr{E}^r(n)|\right) = O\left(\sum_{\forall n^* \in N^*} |\mathscr{E}(n^*)|\right). \tag{17}$$

The computational complexity of TMAP is therefore of the same order of computational complexity as a conventional technology mapper.

Theorem 5.1 also implies that TLUT mapping can be done using a conventional mapper with a front-end to produce the reduced circuit. This method was used in our first TLUT mapper [Bruneel and Stroobandt 2008a]. The reduced circuit can be derived from the original input circuit in linear time. The front-end simply visits all nodes of the original circuit in topological order and removes the parameter nodes during this process. Since the front-end runs in linear time, this mapper scaled equally well as the mapper described in this article. However, the mapper described in this section has the advantage that the cost function can be easily optimized for TLUT mapping. In this article we focused on the minimization of the depth and the area of the resulting LUT structure. However, in TLUT mapping, it might be interesting to also incorporate the reconfiguration effort in the cost function. During the mapping process, we could, for example, try to minimize the number of configuration bits that need to be reconfigured or the complexity of the reconfiguration procedure. This is not possible with the approach described in Bruneel and Stroobandt [2008a].

5.5. Experiment: Multipliers

In this section we use TMAP to map multipliers where one of the inputs is a parameter. The result of the mapping is a hardware constant multiplier where the constant can be changed by means of reconfiguration. This type of multiplier has been hand-designed by several authors [Chapman 1993; Wirthlin 2004]. The main advantage of our technique

Table IV. Comparing mapping results (($K = 4$) of conventional mapping and TMAP of a set of differently-sized multipliers where one of the inputs is a parameter.

	Conventional		TMAP		
size	LUTs	Depth	LUTs	TLUTs	Depth
4×4	33	6	8	8	1
8×8	170	14	52	38	10
16×16	876	28	271	152	23
32×32	3650	55	1104	501	50
64×64	15142	111	4614	1972	105

is of course that TMAP can generate the same result from an architecture-independent HDL description of a multiplier.[4] In the experiment, we mapped multipliers of different sizes, going from a 4×4 multiplier up to a 64×64 multiplier.

The experiments in this section make use of three different technology mappers: a conventional mapper as described in Section 4.3; the optimized implementation of TMAP as described in Section 5.3, and the naive implementation of TMAP; as described in Section 5.1. All these algorithms are implemented in Java running on the Java HotSpotTM 64-Bit Server VM. The computer in the experiments used an Intel Core 2 processor running at 2.13GHz and has 2 GiB of memory.

The conventional mapping algorithm takes an .aig file [Biere 2007] as input and produces a .blif file [University of California Berkeley 2005] that represents the mapped circuit. Both TMAP implementations take an .aig file and a list of the parameter inputs as input, and outputs a LUT structure and an .aig file that represents the PPC (Section 6). The LUT structure can be represented as either a VPR .net file [Betz and Rose 1997] or a VHDL file that directly instantiates Virtex-II pro LUTs [Xilinx 2008b]. The .net file can be further placed and routed using VPR. The VHDL file can be used as input to the ISE tool flow; this is explained in detail in Bruneel et al. [2009].

5.5.1. Conventional Mapping vs. TMAP. In this first experiment, we compare the mapping result of a conventional mapper with the mapping result of TMAP. The mapping results ($K = 4$) are shown in Table IV. The first columns shows the size of the multipliers. The second and the third columns show the mapping results for the conventional LUT mapper. The fourth, fifth, and sixth columns show the mapping results for TMAP.

If we compare the number of LUTs found by the conventional mapper with the number of LUTs found by TMAP, we see a decrease of at least 69%. This gain in FPGA resources is due to the fact that part of the functionality of the multiplier has moved from expensive FPGA resources to the reconfiguration procedure, which is executed in software.

We also compare the depth, that is, the number of LUTs in the longest path, of the circuits. The depth of a circuit is a measure for the maximum delay in a circuit, or, in other words, the maximum operating frequency of the circuit. We see a decrease in depth of at least 4 LUTs. This decrease is independent of the size of the multiplier, and thus is more prominent for multipliers with a small data width.

Finally, column five shows that only part of all LUTs in the design (column four) are TLUTs. Many LUTs do not depend on the parameter inputs, and thus have a static truth table. For FPGAs that are partially reconfigurable, this may further decrease the reconfiguration time.

5.5.2. Scalability. In a second experiment, we compare, the run-times of the naive implementation of TMAP, described in Section 5.1, and the optimized TMAP algorithm, described in Section 5.3. The results are shown in Table V. The table shows the

[4]The multiplier in our experiment is implemented as a tree of adders.

Table V. Comparing execution time of the naive and the optimized implementation of TMAP ($K = 4$) for a set of different-sized multipliers where one of the inputs is a parameter.

	Enumerated Cones			Execution time [s]			Time per cone [μs]	
size	naive	optimized	ratio	naive	optimized	speedup	naive	optimized
4×4	61216	20848	2.94	0.53	0.64	1.21	8.69	30.75
8×8	388521	118966	3.27	1.66	0.93	1.68	4.27	7.85
16×16	3737049	608560	6.14	13.79	2.38	5.47	3.68	3.92
32×32	46070046	2513673	18.33	465.08	6.97	61.45	10.10	2.77
64×64	602449833	10571999	56.99	87705.32	24.96	3514.26	145.58	2.36
128×128	–	44096346	–	–	114.03	–	–	2.59

run-times and the number of enumerated cones for both algorithms. The naive implementation for the 128×128 multiplier ran out of memory after more than four days of run-time. In that time, only 88,173 of the 256,313 AIG nodes were processed. Since the processing time per node grows fast for nodes with higher rank, the computation was far from finished. The missing data is indicated with dashes in the table.

If we look at the speed-up of the optimized algorithm compared to the naive algorithm (column seven), we clearly see that as the input multiplier grows larger, the speed-up increases. The complexity of the optimized algorithm is hence of a lower order than the complexity of the naive algorithm. There are two reasons for this. First, the number of cones enumerated by the optimized algorithm is of a lower order than the number of cones enumerated by the naive algorithm. This can be seen in column four, where we give the ratio of the enumerated cones of both algorithms. The number of enumerated cones is lower for the optimized algorithm because it avoids enumerating incomplete cones (Section 5.2). Second, since the size of a reduced cut (Section 5.3) is limited by K, the complexity of operations on reduced cuts is independent of the input circuit. On the other hand, the complexity of operations on full cuts is limited by $K + |PI|$, where PI is the set of parameter inputs. Thus, the complexity of operations on full cuts grows with the number of parameter inputs. This effect can be observed in column eight and column nine, which show the average time needed to enumerate one cone for the naive algorithm and the optimized algorithm, respectively. In column nine, we see that for the optimized algorithm, the time needed to enumerate one cone is independent of the size of the input circuit for large multiplier sizes. For small multipliers, the overhead of starting the JVM distorts this trend. In other words, this means that the execution time of the optimized algorithm is, at least for multipliers, proportional to the number of enumerated cones, as we claimed in Section 5.4. In column eight, on the other hand, we see that this is not the case for the naive algorithm. There, the time needed to enumerate one full cone grows as the multiplier size increases.

6. GENERATING THE PPC

As we explained in Section 2, a DDF system solves a given problem in two steps. In the first step, a configuration specialized for the problem at hand is generated and loaded into the FPGA. In the second step, this specialized configuration is executed by the FPGA. The cost of solving the problem is the sum of the costs of both steps. Up until now, we have focused on minimizing the cost of the second step by optimizing the area and the length of the longest path of the LUT structure. In this section we discuss the cost of generating a specialized configuration.

In Section 3 we explained that the partial parameterizable configuration (PPC) has to be evaluated in order to generate a partial configuration that may be used to specialize the current FPGA configuration. This PPC is a multioutput Boolean function that maps the parameter inputs to the truth table entries of the LUTs. We assume that the cost of generating a specialized configuration is proportional to the number of Boolean operations that need to be performed in order to evaluate the PPC.

```
function constructPPC(ConeSet cover)
  AIG PPC = new AIG();
  for cone in cover:
    for entry in 0 to 2^K−1:
      Cone copy = cone.copy();
      copy.connectRegularInputs(entry.bits());
      copy.add(new Output(copy.root()));
      PPC.add(copy);
  return PPC;
```

Fig. 9. Pseudocode for constructing the PPC as an AIG from the input AIG and the cover found by the mapper.

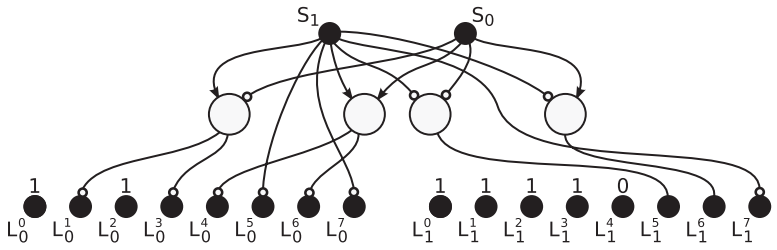


Fig. 10. The optimized PPC of the 4-to-1 multiplexer example represented as an AIG.

In Bruneel and Stroobandt [2008a, 2008b], we treated every truth table entry of the TLUTs as an independent single-output Boolean function (see Figure 7). However, representing a multioutput Boolean function as a set of independent single-output Boolean functions prevents the use of combined logic optimization, which could further minimize the total number of Boolean operations needed to generate a specialized configuration. Therefore, we will represent the PPC as a Boolean network, more specifically, an AIG. This allows us to use well-established Boolean minimization techniques, like those implemented in ABC [Berkeley Logic Synthesis and Verification Group], to minimize the number of Boolean operations needed to evaluate the PPC.

A Boolean network that generates one entry of the truth table of a TLUT can be constructed by copying the cone of the TLUT and connecting the regular inputs of that cone to constants associated to the entry. In that case, the root of the cone produces the truth table entry as a function of the parameter inputs, and thus needs to be connected to an output of the network. If this is repeated for every entry of every cone in the covering, the result is a Boolean network with, as inputs, the parameter inputs and, as outputs, the entries of the truth tables of the TLUTs. The pseudo-code for constructing the PPC is given in Figure 9. At each moment during the construction, the PPC is structurally hashed [Kuehlmann et al. 2002]. This means that constants are propagated, and there is at most one node with a given pair of input edges. If we apply this process to our multiplexer example, we get the PPC shown in Figure 10.

As can be derived from the preceding description, each node in the input AIG is copied 2^K times for each cone in the cover it is part of. Without considering any optimization, the maximum number of nodes in the PPC can be expressed as shown in Eq. (18), where $dup(n)$ is equal to the number of cones in the cover it is part of, and $\widehat{dup}$ is the average of $dup(n)$ over N.

$$\sum_{n \in N} 2^K dup(n) = 2^K \widehat{dup} |N| \tag{18}$$

Table VI. Study of scalability of the PPC for a set of differently-sized multipliers where one of the inputs is a parameter.

size	LUTs	TLUTs	$\widehat{dup}$	$\|N\|$	eq. (18)	$\|PPC\|$	$\frac{\|PPC\|}{\|N\|}$
4×4	8	8	2.8	109	4,883	156	1.43
8×8	52	38	4.4	545	38,368	470	0.86
16×16	271	152	8.3	2608	346,343	1918	0.74
32×32	1104	501	14.3	10839	2,479,963	8503	0.78
64×64	4614	1972	27.0	45111	19,487,952	33783	0.75

For duplication-free mapping, $dup(n)$ equals one. In that case, the equation can be simplified to $2^K|N|$. In other words, in duplication-free mapping, the complexity of generating a specialized configuration given a set of parameter values is linear in the number of nodes in the input AIG.

In general, Eq. (18) cannot be written in a closed form, but we can find an upper bound. If we note that $dup(n)$ is limited by the number of cones in the cover and that this number is limited by the number of nodes in the subject graph, we find that the number of nodes in the PPC is $O(|N|^2)$.

This upper bound is very pessimistic because it does not consider any of the many optimization opportunities (constant propagation, structural hashing) for the PPC. To strengthen this claim, we measured the number of operations in the PPC for the multiplier example in Section 5.5. The results can be found in Table VI. Column four shows the average duplication of a node. As can be seen, duplication of nodes is high for the multipliers, up to 27 for the 64×64 multiplier, and, as expected, it rises with $|N|$ (column eight), but less than linear. Column six predicts the number of nodes in the PPC according to Eq. (18), while column seven shows the actual number of AND-gates in the PPC after optimization with the *resyn3* command of ABC [Berkeley Logic Synthesis and Verification Group]. When comparing these numbers, it is clear that Boolean optimization can greatly reduce the number of AND-gates in the PPC. In the last column, we show the ratio of AND-gates in the PPC and AND-gates in the input AIG. This number first drops if the multiplier size rises and then stabilizes for large multipliers at about 0.76. This means that the size of the PPC grows linear with $|N|$ for these multipliers and not quadratic, as the upper bound we found earlier predicted. From Eq. (18), we expect the constant factor of $2^K\widehat{dup}$ to be larger than 16 because $K = 4$, but we find only 0.76. This means that for real-life multipliers, the complexity of calculating new truth tables is less than the complexity of evaluating one multiplication.

7. EXPERIMENTAL RESULTS

In this section, we apply tunable LUT mapping on more complex circuits. In Section 7.1, we create adaptive FIR filters that are adapted by means of reconfiguration, and in Section 7.2, we create TCAMs, whose content is written by means of reconfiguration. Both designs are implemented on a Virtex-II Pro XC2VP30 using Xilinx ISE 9.2.

7.1. Adaptive FIR filter

Adaptive FIR filters that are adapted by means of reconfiguration can be created with TMAP by simply choosing the filter coefficients as the parameters of the design. In what follows we create this sort of adaptive filter for different numbers of taps and input widths, and we compare them with conventional adaptive filters. The filters used are fully pipelined FIR filters, as shown in Figure 11 for a 32-tap filter.

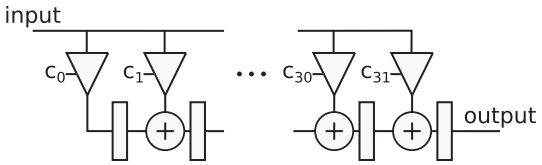


Fig. 11. 32-tap fully pipelined adaptive FIR filter.

Table VII. Hardware properties for a set of differently-sized adaptive FIR Filters implemented without using dynamic reconfiguration (conventional), and with dynamic reconfiguration (TMAP). The numbers between brackets are relative compared to the conventional implementation.

Size		Conventional		TMAP			
Width	Taps	LUTs	f_{max} [MHz]	LUTs		f_{max} [MHz]	
8 bit	32	2641	80.84	1520	(0.58)	123.82	(1.53)
8 bit	64	5298	72.41	3056	(0.58)	89.06	(1.23)
8 bit	96	7954	65.41	4592	(0.58)	87.96	(1.34)
8 bit	128	10611	53.81	6128	(0.61)	74.23	(1.38)

On the one hand, the filters are implemented using the conventional ISE 9.2 tool flow stating from RTL descriptions of the filters. Synthesis is done using Xilinx XST 9.2 with the default settings, except for the multiplier style which we set to LUT. This way the multipliers are implemented using LUTs, rather than the hardwired multipliers available in the Virtex-II Pro. This is necessary to allow a fair comparison between the conventional implementation and the TMAP implementation. Technology mapping (Xilinx MAP 9.2) and place and route (Xilinx PAR 9.2) are done using default settings. The number of LUTs and the maximum clock frequency for the filters implemented using ISE can be found in columns three and four of Table VII.

On the other hand, the filters are implemented using TMAP, again starting from RTL descriptions of the filters. This RTL description is first synthesized using Quartus II 7.2. Quartus is set to dump a .blif file after synthesis. This is possible due to the Quartus II University Interface Program (QUIP). Next, the .blif file is converted to a .aig file using ABC [Berkeley Logic Synthesis and Verification Group]. Together with a list of parameter inputs (the filter coeficients in this case) this .aig file is used as input for a Java implementation of TMAP, which produces both a PPC represented as an .aig file and a LUT structure. In this experiment, the LUT structure is represented as a VHDL file that directly instantiates Virtex-II Pro LUTs and FFs [Xilinx 2008b]. Finally, the LUT structure is implemented on the Virtex-II Pro using the Xilinx ISE 9.2, tool flow (XST 9.2, MAP 9.2, and PAR 9.2) with default settings. For more information on how to integrate TMAP with the ISE tool flow we refer to Bruneel et al. [2009].

The number of LUTs and the maximum clock frequency for the dynamically reconfigurable filters can be found in columns five and six of Table VII. If we compare the number of LUTs in both implementations, we see that the TMAP implementations need at least 39% fewer LUTs than the conventional implementations. We also see that the TMAP implementation can be clocked at least 23% and up to 53% faster than the conventional implementation.

Of course, the gain in area and speed of the FPGA hardware of our dynamically reconfigurable adaptive filters comes at the cost of a larger adaptation time. While the filter coefficients of the conventional adaptive filter can be changed by simply rewriting the registers that store the coefficients, the TMAP implementation requires us to both generate a specialized configuration for the FPGA by evaluating the PPC and write this configuration in the configuration memory of the FPGA. The total time needed to change the coefficients of the filter is called the specialization overhead, $t_{special}$. It

Table VIII. Evaluating the PPC for differently-sized adaptive FIR filters on the hardwired PowerPC of the Virtex-II Pro.

Size		Evaluation Time			Program Size
Width	Taps	$\|PPC\|$	t_{eval} [μs]	$\frac{t_{eval}}{\|PPC\|}$ [$\frac{ns}{AND}$]	S_{eval} [B]
8 bit	32	28672	317	11.06	14150
8 bit	64	57344	634	11.06	14918
8 bit	96	86016	951	11.06	15686
8 bit	128	114688	1268	11.06	16454

contains both the time needed to evaluate the PPC, t_{eval}, and the time to reconfigure the FPGA, t_{reconf}. In what follows we discuss the specialization overhead in detail.

As shown in Figure 1, the evaluation of the PPC is done on an instruction set processor (ISP). In our case, we use the PowerPC which is hardwired on the Virtex-II Pro FPGA. Efficiently evaluating a Boolean network on an ISP is an area of research by itself, and is beyond the scope of this article. However, in order to give an estimate of the evaluation time, we have implemented a simple compiled evaluation technique. In compiled evaluation, a dedicated function is created that takes the input values of the network as its arguments and returns the output values of the network. In our case, the network is the PPC created by TMAP, the input values are the parameter values (the coefficients of the filter), and the output values are the truth tables for the TLUTs.

Starting from the PPC, an evaluation function is created by first generating a C function and then compiling it for the PowerPC. We generate the C code of the evaluation function by traversing the PPC in topological order from the inputs towards the outputs. For every node, a statement is added to the C code. The statement calculates the output value of the node from the output value of its predecessors and stores the output in a local array (`node`). The size of the local array is minimized by freeing its elements when they are no longer needed and by always storing a node output value at the smallest available index. For example. if the left predecessor is inverted, the smallest available index is 3 and and the output values of the left and right predecessors are stored respectively, at indices 9 and 6, the expression would be `node[3] = !node[9] && node[6];`.

Unfortunately, when we generate an evaluation function, as described above, for the complete flattened FIR filter, this leads to very large evaluation function and poor evaluation time. However, many designs, including our FIR filters, contain hierarchy and have a repetitive nature that can be used to build a more compact evaluation function. In our filters, one multiplier is instantiated for every tap. Instead of generating one flat evaluation function for the complete FIR filter, we generate an evaluation function for one multiplier, as described above, and build the FIR evaluation function by calling this function for each instantiation of the multiplier. We could even optimize this further by calculating up to 32 of the multiplier evaluation functions at a time by using bitwise logic operations and packing 32 Boolean values in each 32-bit word. Although we have built the FIR evaluation function manually for our experiments, it could easily be synthesized automatically from the hierarchy found in the HDL design.

In our experiment we created the evaluation function, as described above, for each of the FIR filters and executed it on the PowerPC. The PowerPC was clocked at 300 MHz, and both the instruction and data caches were enabled. The evaluation time, t_{eval}, and the size of the compiled evaluation function S_{eval} are shown in Table VIII. We see that the evaluation time for our adaptive filters takes on the order several hundreds μs,w depending on the size of the filter. The ratio of the evaluation time and the number of AND nodes in the PPC shows that the evaluation time of the filters is very linear in the size of the PPC. The size of the evaluation functions is about 15 kB, and slowly grows as the number of taps increases. The program size is almost independent of the

Table IX. Reconfiguration for differently-sized adaptive FIR Filters via ICAP of the Virtex-II Pro.

size		Reconfiguration Time			
Width	Taps	TLUTs	Frames	S_{bit} [B]	t_{reconf} [μs]
8 bit	32	768	52	66573	1009
8 bit	64	1536	88	111357	1687
8 bit	96	2304	92	115493	1750
8 bit	128	3072	91	114669	1737

Table X. Specialization overhead for differently-sized adaptive FIR filters implemented on the Virtex-II Pro.

size		Specialization Overhead		
Width	Taps	t_{eval} [μs]	t_{reconf} [μs]	$t_{special}$ [μs]
8 bit	32	317	1009	1326
8 bit	64	634	1687	2321
8 bit	96	951	1750	2701
8 bit	128	1268	1737	3005

size of the PPC because one multiplier evaluation function is created, which is reused to generate the truth tables for each of the multipliers in the design.

After evaluating the PPC, the PowerPC needs to write the calculated truth tables in the configuration memory of the FPGA. The PowerPC can access the configuration memory of the Virtex-II Pro from within the FPGA fabric via ICAP (internal configuration access port), which we connected to the bus of the PowerPC. To reconfigure the FPGA, the PowerPC needs to send a partial bitstream to the ICAP, which can be done at a maximum rate of 66 MB/s. The size of the bitstreams, S_{bit}, and the reconfiguration time, t_{reconf}, are shown in column five and six of Table IX. The reconfiguration time ranges from 1 ms to a maximum of 1.75 ms, depending on the size of the filter. As can be seen, the reconfiguration time is not linear in the number of TLUTs, as we could expect, but linear in the number of frames that need to be reconfigured. This is because the atom of reconfiguration of the Virtex-II Pro is not a LUT truth table, but a frame [Xilinx 2007]. All the truth tables of a column of CLBs (configurable logic blocks) are stored in only two frames. If only one LUT in a CLB column changes, half of the LUTs in that column need to be reconfigured. Because its frames are smaller, the importance of this overhead is reduced for the Virtex-5 [Xilinx 2010].

Finally, the total specialization overhead, which is the sum of the evaluation time and the reconfiguration time, is shown in Table X. As can be seen, the specialization time is the order of a few ms, depending on the size of the filter. Thus we can exploit the area and clock frequency benefits of our adaptive filters (Table VII) as long as the time between coefficient changes is a few orders of magnitude higher than the specialization overhead.

7.2. Ternary Content-Addressable Memory

In conventional memories, the read operation returns the data associated with a given address. The read operation of a content addressable memory (CAM) does the opposite: it finds the address associated to a given data value. In both cases, the write operation stores a given data value at a given address. CAMs have many applications [Altera 2001]. The most important commercial application is packet-forwarding in network routers [Pagiamtzis and Sheikholeslami 2006].

A TCAM (ternary CAM) is a special kind of CAM that stores ternary patterns instead of pure data. Each digit in a ternary pattern can either be zero, one, or “don’t care”. The digits are represented by two bits: the data bit and the mask bit. A full pattern entry in the TCAM is represented by two bit vectors (the data and the mask) and one bit, which indicates whether the entry contains a pattern or not. When new input data is provided to the TCAM, it simultaneously compares this data to all stored patterns.

```
entity tcam is
  generic (
    DATA_W   : integer := 32;
    ADDR_W   : integer := 8 );
  port (
    clk      : in    std_logic;
    entry    : in    entry_array; --PARAM
    datain   : in    std_logic_vector(DATA_W-1 downto 0);
    addrout  : out   std_logic_vector(ADDR_W-1 downto 0);
    match    : out   std_logic  );
end tcam;

architecture rtl of tcam is
begin
READ: process(clk) is
  variable local  : std_logic;
begin
  if rising_edge(clk) then
    match  <= '0';
    addrout <= (others => '0');
    for i in 2**ADDR_W-1 downto 0 loop

      local := entry(i).used;
      for j in 0 to DATA_W-1 loop
        local := local and ((entry(i).data(j) xnor datain(j)) or entry(i).mask(j));
      end loop;

      if (local = '1') then
        match  <= '1';
        addrout <= std_logic_vector(to_unsigned(i, ADDR_W));
      end if;

    end loop;
  end if;
end process READ;
end architecture rtl;
```

Fig. 12. Parameterizable HDL code for a TCAM, where the 256 entries of width 32-bits are the parameters; changing the content of the TCAM is done via reconfiguration.

The incoming data, matches a pattern if all bits of the incoming data, for which the corresponding mask bit of the pattern is zero, are equal to the corresponding value bit of the pattern.

The VHDL code of Figure 12 shows a behavioral description of the read operation of a TCAM. It takes *datain* as input and outputs the matching address, if any, on *addrout*. The output *match* indicates whether a match is found or not. The READ process in Figure 12 describes the hardware that compares the input data to all stored patterns. The code takes the array of pattern entries as input. In a conventional TCAM implementation, the entries will be provided by flip-flops (FFs) arranged as a memory. Each memory element uses a FF in the FPGA because all data needs to be accessed in every clock cycle. In our reconfigurable implementation, these inputs are the parameters of the design, and thus will be provided by means of reconfiguration.

Table XI. Comparing different implementations of a TCAM on a Virtex-II Pro. (ISE) synthesis from behavioral VHDL using ISE 9.2. (SRL16) generated with Xilinx Coregen. (TMAP) synthesis from behavioral VHDL (Figure 12) using TMAP.

Design		ISE			SRL16			TMAP		
Width	Entries	LUT	FF	f_{max} [MHz]	LUT	FF	f_{max} [MHz]	LUT	FF	f_{max} [MHz]
16	128	2516	4302	86.72	1504	127	79.26	1095	56	88.72
16	256	5100	8577	74.36	2886	130	68.24	2217	85	83.51
32	128	4569	8419	79.97	2664	223	68.13	1735	237	95.57
32	256	10441	16874	69.01	5070	226	59.69	3497	259	90.00

In important applications such as Internet core routers, this approach is feasible, since the update rate is usually rather limited (at the very most a few hundred updates per second [Labovitz et al. 1998]), while the read rate is orders of magnitude higher (up to several millions of packets per second).

The problem with TCAMs is that their implementation requires many FPGA resources, even for small TCAMs. If the code in Figure 12 is extended by making `entry` an internal signal and adding a write port and a write process, we get a VHDL description of a full TCAM (256 entries of 32-bits). When we synthesize this description using ISE for a Virtex II Pro, the implementation requires 16,874 FFs and 10,441 4-input LUTs, and can be maximally clocked at 69 MHz. This is true for different sizes of TCAM (see Table XI).

These resource requirements can be drastically reduced with the use of TMAP. In the TMAP design (Figure 12), we chose the entry array of the TCAM as the parameter input of the design, by adding the `PARAM` annotation. This means that the patterns stored in the TCAM will be changed by means of reconfiguration. When we map this design using TMAP, it only requires 3,497 LUTs (a reduction by 67%); the maximum clock frequency rises from 69 MHz to 90 MHz (a gain of 30%); and the number of FFs is reduced dramatically from 16,874 to only 226 (see Table XI). This reduction in FFs is possible because the pattern information is no longer stored in the FFs that are part of the FPGA fabric, but in the memory elements of the configuration memory that stores the truth tables of the LUTs. The only FFs left are the output registers for `addrout` and `match`. We have also buffered `datain` in order to measure the maximum clock frequency. These last FFs are duplicated several times by ISE for speed optimization.

Due to the importance of TCAMs and the high resource usage of architecture-independent HDL implementations, FPGA vendors offer TCAM constructor software that constructs TCAM structures which are highly optimized for a specific architecture [Altera 2001; Xilinx 2008a]. Designers can generate such a structure using the software, and then instantiate it in their design. A good example of such a generator is the SRL16 TCAM generator [Brelet and New 1999] embedded in Xilinx Coregen, which generates TCAM structures that are very similar to the TCAMs that TMAP synthesizes from the code in Figure 12. The results for the SRL16 TCAM are shown in Table XI. As can be seen, the SRL16 TCAM (256 entries of 32-bits) is 45% larger and clocks 34% slower than the TMAP design. This is mainly due to the infrastructure needed to write new entries in the TCAM.

Again, the gain in area and speed of the FPGA hardware for our dynamically reconfigurable TCAM comes at the cost of a larger time to write an entry. While in the ISE implementation an entry can be rewritten in one clock cycle and in the SRL16 implementation in 16 clock cycles, the TMAP implementation requires us to both generate a specialized configuration for the FPGA by evaluating the PPC and write this configuration in the configuration memory of the FPGA. In the next experiment, we measured the specialization overhead for the TMAP implementation. As explained in Section 7.1, the evaluation of the PPC is done on the embedded PowerPC and the

Table XII. Specialization overhead for differently-sized TCAMs implemented on the Virtex-II Pro. (One entry): Only one of the TCAM entries is written. (All entries): All entries of TCAM are written.

Design		One Entry (Worst Case)					All Entries				
Width	Entries	S_{eval} [B]	t_{eval} [μs]	frames	t_{reconf} [μs]	$t_{special}$ [μs]	S_{eval} [B]	t_{eval} [μs]	frames	t_{reconf} [μs]	$t_{special}$ [μs]
16	128	7362	1.50	4	104	106	7446	190	41	795	985
16	256	7362	1.50	4	104	106	7446	380	52	1034	1414
32	128	9750	2.84	9	205	208	9834	360	49	946	1306
32	256	9750	2.84	9	242	245	9834	720	52	996	1716

Table XIII. Several TCAMs mapped to differently-sized LUTs: $K = 3$, $K = 4$, and $K = 5$.

Design		$K = 3$			$K = 4$			$K = 5$		
Width	Entries	Conv.	TMAP		Conv.	TMAP		Conv.	TMAP	
16	128	3637	1604	(0.44)	2516	1095	(0.44)	2471	867	(0.35)
16	256	7278	3197	(0.44)	5100	2217	(0.44)	4863	1724	(0.36)
32	128	6958	3022	(0.43)	4569	1735	(0.38)	4780	1527	(0.32)
32	256	13919	6013	(0.43)	10441	3497	(0.34)	9337	3018	(0.32)

reconfiguration is done using the ICAP of the Virtex-II Pro. We did the measurement both in the case where only one entry needs to be rewritten and in the case where all entries are rewritten. The results are shown in Table XII. If only one entry is written, the reconfiguration time depends on the way the TLUTs of the entry are placed on the FPGA, because this placement determines the number of frames that need to be reconfigured. Table XII shows the reconfiguration time for the worst-case entry. For the largest TCAM (256 entries of 32 bits), reconfiguring one entry takes 245 μs in the worst case and reconfiguring the full TCAM takes 1716 μs. More details can be found in the table.

The disadvantage of using generator software is that it results in architecture dependent designs, because the TCAM structures internally instantiate architecture-specific resources. Our TMAP design does not have that problem. The VHDL code of Figure 12 is architecture-independent. The same design can be mapped to several FPGA architectures by simply changing the mapper. Of course, these mappers must have TLUT capability in order to benefit form the resource reduction. To strengthen this point, we have also mapped the code of Figure 12 to architectures with different LUT sizes. The LUT usage for $K = 3$, $K = 4$, and $K = 5$ can be found in Table XIII, where we can clearly see that, for the TCAMs, the relative area gain improves when the LUT size increases.

8. CONCLUSIONS

In this article we introduced a tool flow that automatically generates a dynamic data-folding implementation starting from an RT-level HDL design. Its main contribution is a novel technology mapper called TMAP. The mapper maps Boolean circuits to Tunable LUTs (TLUTs), these are LUTs where the truth table is expressed as a function of the parameter inputs. We have shown that the complexity of TMAP scales as well as that of a conventional LUT mapper. In our DDF architecture, specialized configurations are generated on-the-fly by evaluating Boolean functions. We expressed these functions as a single Boolean network, which opened up the possibility of using well-known combined Boolean optimization techniques. We have shown that the complexity of the resulting Boolean network, and thus the time required to generate a specialized configuration, scales favorably with the size of the original design.

Our approach is validated by implementing adaptive FIR filters and ternary content-addressable memories (TCAMs) on a Virtex-II Pro.[5] We show large reductions in the number of LUTs (39% for the FIR filter and 66% for the TCAMs) and significant

[5]The FIR filter has 128 taps with 8-bit wide coefficients. The TCAM has 256 entries that are 32 bit wide.

improvements for the maximum clock frequency (38% for the FIR filter and 30% for the TCAM). The specialization of both designs was done using the embedded PowerPC and the ICAP of the Virtex-II Pro. The total time needed to change the coefficients of the filter is 1.74 ms, and the content of the TCAM can be rewritten in 1.72 ms. FIR filters and TCAMs are only two of a large class of applications that can benefit from DDF. Due to its general applicability and the RT-level design, our technique makes designing DDF systems feasible for many applications. Other applications that may benefit from our DDF technique are; encryption algorithms like AES and DES; template matching [Wirthlin and Hutchings 1997]; regular expression matching [Hutchings et al. 2002], DNA aligning [Puttegowda et al. 2003; Yamaguchi et al. 2002]; serial fault emulation [Burgun et al. 1996]; and many others.

REFERENCES

ALTERA. 2001. Application note 119: Implementing high-speed search applications with Altera CAM, Altera.

BERKELEY LOGIC SYNTHESIS AND VERIFICATION GROUP. ABC: A system for sequential synthesis and verification. Berkeley Logic Synthesis and Verification Group.

BETZ, V. AND ROSE, J. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications (FPL'97)*. Springer, Berlin, 213–222.

BIERE, A. 2007. The AIGER and-Inverter Graph (AIG) format. Johannes Kepler University.

BRELET, J.-L. AND NEW, B. 1999. XAPP203: Designing flexible, fast CAMs with Virtex family FPGAs. Xilinx.

BRUNEEL, K., ABOUELELLA, F., AND STROOBANDT, D. 2009. Automatically mapping applications to a self-reconfiguring platform. In *Proceedings of the Design, Automation and Test in Europe*. 964–969.

BRUNEEL, K., BERTELS, P., AND STROOBANDT, D. 2007. A method for fast hardware specialization at run-time. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 35–40.

BRUNEEL, K. AND STROOBANDT, D. 2008a. Automatic generation of run-time parameterizable configurations. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 361–366.

BRUNEEL, K. AND STROOBANDT, D. 2008b. Reconfigurability-aware structural mapping for LUT-based FPGAs. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, Los Alamitos, CA.

BRUNEEL, K. AND STROOBANDT, D. 2010. TROUTE: A reconfigurability-aware FPGA router. In *Reconfigurable Computing: Architectures, Tools and Applications,* Lecture Notes in Computer Science, vol. 5992. Springer, Berlin, 207–218.

BURGUN, L., REBLEWSKI, F., FENELON, G., BERBIER, J., AND LEPAPE, O. 1996. Serial fault emulation. In *Proceedings of the 33rd Annual Design Automation Conference (DAC'96)*. ACM, New York, 801–806.

CHAPMAN, K. 1993. Fast integer multipliers fit in FPGAs. *EDN 39,* 10, 80.

CONG, J. AND DING, Y. 1994. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans. Comput.-Aid. Des. 13*, 1–12.

DERBYSHIRE, A., BECKER, T., AND LUK, W. 2006. Incremental elaboration for run-time reconfigurable hardware designs. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*, ACM, New York, 93–102.

FOULK, P. 1993. Data-folding in SRAM configurable FPGAs. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*. 163–171.

HAUCK, S. AND DEHON, A. 2007. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann.

HUTCHINGS, B., FRANKLIN, R., AND CARVER, D. 2002. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, Los, Alamitos, CA, 111–120.

KUEHLMANN, A., PARUTHI, V., KROHM, F., AND GANAI, M. K. 2002. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. Comput.-Aid. Des. Integrat. Circuits Syst. 21,* 12, 1377–1394.

LABOVITZ, C., MALAN, G., AND JAHANIAN, F. 1998. Internet routing instability. *IEEE/ACM Trans. Network. 6,* 5, 515–528.

LEMOINE, E. AND MERCERON, D. 1995. Run-time reconfiguration of FPGA for scanning genomic databases. In *Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* IEEE, Los Alamitos, CA.

LYSECKY, R., STITT, G., AND VAHID, F. 2006. WARP processors. ACM *Trans. Des. Autom. Electron. Syst. 11,* 3, 659–681.

MANOHARARAJAH, V., BROWN, S., AND VRANESIC, Z. 2006. Heuristics for area minimization in LUT-based FPGA technology mapping. *IEEE Trans. Comput.-Aid. Des. Integrat. Circuits Syst. 25,* 11, 2331–2340.

MCGREGOR, G. AND LYSAGHT, P. 1999. Self-controlling dynamic reconfiguration: A case study. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications (FPL'99)*. Springer, Berlin, 144–154.

PAGIAMTZIS, K. AND SHEIKHOLESLAMI, A. 2006. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE J. Solid-State Circuits 41,* 3, 712–727.

PAN, P. AND LIN, C.-C. 1998. A new retiming-based technology mapping algorithm for LUT-based FPGAs. In *Proceedings of the ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA'98)*. ACM, New York, 35–42.

PUTTEGOWDA, K., WOREK, W., PAPPAS, N., DANDAPANI, A., ATHANAS, P., AND DICKERMAN, A. 2003. A run-time reconfigurable system for gene-sequence searching. In *Proceedings of the 16th International Conference on VLSI Design*. 561–566.

SANKAR, Y. AND ROSE, J. 1999. Trading quality for compile time: Ultra-fast placement for FPGAs. In *Proceedings of the ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays (FPGA'99)*. ACM, New York, 157–166.

UNIVERSITY OF CALIFORNIA BERKELEY. 2005. Berkeley Logic Interchange Format (BLIF). University of California, Berkeley.

WIRTHLIN, M. J. 2004. Constant coefficient multiplication using look-up tables. *J.VLSI Signal Process. 36,* 1, 7–15.

WIRTHLIN, M. J. AND HUTCHINGS, B. L. 1997. Improving functional density through run-time constant propagation. In *Proceedings of the ACM 5th International Symposium on Field-Programmable Gate Arrays (FPGA'97)*. ACM, New York, 86–92.

XILINX. 2010. *Virtex-5 FPGA Configuration User Guide*. Xilinx.

XILINX. 2008a. *DS253: Content-Addressable Memory v6.1*. Xilinx.

XILINX. 2008b. *Virtex-II Pro Libraries Guide for HDL Designs*. Xilinx.

XILINX. 2007. *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*. Xilinx.

XILINX. 2006. *UG208: Early Access Partial Reconfiguration User Guide*. Xilinx.

YAMAGUCHI, Y., MIYAJIMA, Y., MARUYAMA, T., AND KONAGAYA, A. 2002. High-speed homology search using run-time reconfiguration. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL'02)*. Springer, Berlin, 281–291.

Received June 2010; revised October 2010; accepted March 2011