How Parameterizable Run-time FPGA Reconfiguration can Benefit Adaptive Embedded Systems

Dirk Stroobandt and Karel Bruneel
Ghent University, ELIS Department, Gent, Belgium, Dirk.Stroobandt@UGent.be

Abstract—*Adaptive embedded systems are currently investigated as an answer to more stringent requirements on low power, in combination with significant performance. It is clear that runtime adaptation can offer benefits to embedded systems over static implementations as the architecture itself can be tuned to the problem at hand. Such architecture specialisation should be done fast enough so that the overhead of adapting the system does not overshadow the benefits obtained by the adaptivity. In this paper, we propose a methodology for FPGA design that allows such a fast reconfiguration for dynamic datafolding applications. Dynamic Data Folding (DDF) is a technique to dynamically specialize an FPGA configuration according to the values of a set of parameters. The general idea of DDF is that each time the parameter values change, 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 counterpart, the hope is that their corresponding system implementation will be more cost efficient. In this paper, we show that DDF can be implemented on current commercial FPGAs by using the parameterizable run-time reconfiguration methodology. This methodology comprises a tool flow that automatically transforms DDF applications to a runtime adaptive implementation. Experimental results with this tool flow show that we can reap the benefits (smaller area and faster clocks) without too much reconfiguration overhead.*

Keywords: Automatic hardware synthesis, Dynamic Data Folding, FPGA, Run-time reconfiguration

1. Introduction

In order to keep up with more stringent requirements on power usage along with performance, current embedded systems are increasingly made adaptive. In a first step towards full adaptivity, task scheduling on embedded systems has been changed from static (off-line) to dynamic (on-line) scheduling so as to cope with dynamism in the applications. Generally, application scenarios are detected at run-time and for each scenario, the proper schedule is chosen from the set of statically derived schedules for all application scenarios on the architecture at hand [1], [2], [3]. In more advanced embedded systems, also the architecture itself is made adaptive [4], [5]. In this way, not only the schedule can change but also the resource allocation can be altered depending on the application scenario at hand. It is clear that such runtime architecture adaptation can offer benefits to embedded systems over static implementations as the architecture itself can be tuned to the problem at hand. Such architecture specialisation should be done fast enough so that the overhead of adapting the system does not overshadow the benefits obtained by the adaptivity.

One hardware component that is extremely well fit for combining performance with adaptivity is the FPGA. 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 [5]. Therefore, dynamically reconfiguring FPGAs is a good way of introducing the architecture adaptivity in the context described above.

If the number of possible application scenarios is limited, a dynamically reconfiguring system can easily be implemented with a conventional FPGA tool flow. One simply generates an FPGA configuration optimized for each possible situation and stores 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.

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, change only once in a while. Each time the parameters change value, 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. This makes it impossible to store all possible configurations. On the other hand, a conventional FPGA tool flow is too slow to be executed at run-time. DDF can therefore not be implemented with a conventional tool flow.

Our research group at Ghent University is the first to present an *automatic* tool flow that builds DDF implementations, thus bringing the FPGA architecture to the level where it could be useful in a dynamic run-time adaptive embedded system [5]. Our methodology and tool flow starts from parameterized HDL designs. These are RT-level HDL designs in which a distinction is made between regular inputs

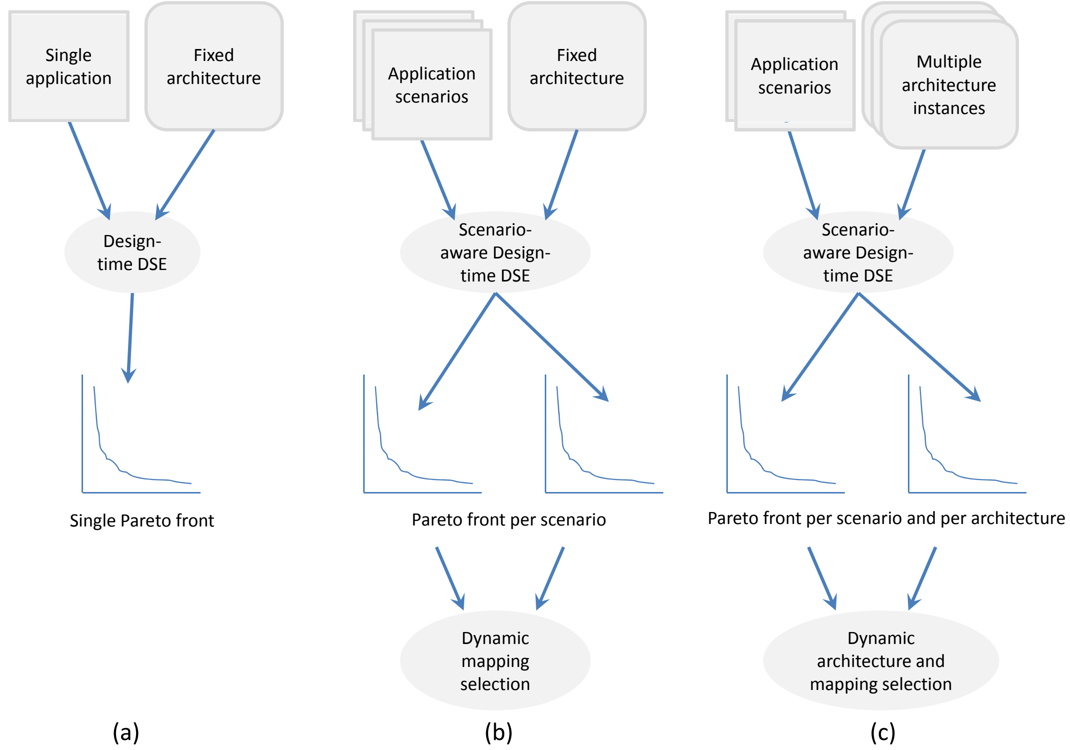

Fig. 1: Different forms of system-level DSE, with increasing dynamism from (A) to (C).

and parameter inputs. The parameter inpus will define the reconfiguration intervals. 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 configuration manager does not fetch the specialized configuration from a large configuration database. Instead, it generates the required configuration on the fly by evaluating the closed-form Boolean expressions for the new parameter values.

This paper starts with a brief discussion on the context of adaptive embedded systems (Section 2), as well as a description of DDF and an overview of related work in Section 3. In Section 4, 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, which we generalize to obtain a new technology mapper called TMAP, suited for our DDF tool flow. We show that our new method for parameterizable run-time reconfiguration can be implemented on current FPGA devices and without too many changes in the FPGA implementation tool flow (Section 5). Furthermore, we apply TMAP on adaptive FIR filters and Ternary Content-Addressable Memory in Section 6. Experimental results show that the use of self-reconfiguration with our tool flow improves the resource demands of the application by 39% for a 32-tap adaptive filter and 66% for a Ternary Content Addressable Memory implementation, without introducing a prohibitively large reconfiguration generation overhead. Finally, we conclude in Section 7.

2. Adaptive Embedded Systems

Today, modern embedded computing systems are not only rapidly evolving towards MultiProcessor System-on-Chîps (MPSoC), they are also increasingly dynamic and adaptive. The application workload can change dramatically over time. For this reason, the notion of application scenarios has gained interest in the past years [1], [2], [3]. An application scenario describes the evolution of system use cases, i.e., the combinations of applications that can be active at the same time. This has important implications on the scheduling process that maps tasks to the computing nodes. A careful trade off has to be made of non-functional requirements such as performance, power, cost, etc. Design Space exploration (DSE) is therefore a very important aspect in this mapping process. DSE is a multi-objective optimization problem that searches through the space of different mapping alternatives in order to find Pareto-optimal design instances. A design instance is said to be Pareto-optimal when it is optimal for at least one of the optimization objectives (e.g., per-

formance, power, cost, etc.). The traditional, design-time DSE is illustrated in Figure 1(A). With the occurrence of different application scenarios, this traditional DSE has to be extended. The mapping decisions can no longer be made at design time. A run-time system configuration manager will be needed to dynamically map and re-map applications onto the underlying architectural resources. The current state-of-the art in this field has only very recently started to address this situation (Figure 1(B)) [1], [6], [7], [8], [9], [10].

However, even more flexibility and optimization options are available when also the underlying architecture of embedded systems is adaptive. One way of achieving this is to include reconfigurable hardware components such as Field Programmable Gate Arrays (FPGAs), now popular architectural elements that enable to accelerate specific computational kernels in applications by means of run-time hardware reconfiguration (e.g. [4], [5]). Making not only the applications but also the hardware adaptive, poses additional challenges to DSE. Ideally, a system configuration manager in such a system continuously optimizes the system for non-functional requirements (performance, power consumption, etc.) at run-time, both by means of mapping of application tasks and by reconfiguraing architectural processor and network components (Figure 1(C)). This type of DSE is still only in the research planning phase. However, the optimization possibilities clearly outnumber the ones in previous DSE frameworks.

In such a new type of DSE optimizations, this increased flexibility poses an additional problem. Application scenarios can be considered a given (and can be measured or modelled). However, the use of FPGAs offers a seemingly infinite architecture implementation space and the best architecture has to be found for each application scenario instance. The main challenge in this kind of framework is hence the right choice of FPGA implementations to consider in DSE. In the next section, we will show that dynamic data folding applications offer an interesting perspective that offers the possibility to limit the implementation choices while retaining the flexibility needed to implement an almost optimal architecture for each application scenario.

3. Dynamic Data Folding

Dynamic data folding (DDF) applications have two types of inputs that are treated differently: fast changing inputs (regular inputs) and slow changing inputs (parameter inputs). Instead of building generic circuitry where both types of inputs are normal input signals, we build a dynamic data folding system where only the regular inputs are inputs to a reconfigurable module implemented in the FPGA fabric. The parameters are inputs to a second subsystem, the configuration manager (CM), in our case an instruction set processor (ISP). Every time the parameters change, the CM specializes the reconfigurable module for the new parameter values. Once specialized, the module is ready to process the fast changing input data. The reason to build a DDF system is that the reconfigurable module can be implemented more efficiently in the FPGA fabric than the generic circuitry. With convential FPGA tools only handcrafted DDF systems are possible [11], [12]. The TMAP tool flow on the other hand (see Section 4) automatically maps dynamic data folding applications to a self-reconfiguring system [13]. The input of the tool chain is a behavioral description of the functionality in which a distinction is made between regular inputs and the parameter inputs. The output is a Tunable LookUp Table (TLUT) circuit that consists of a fixed LUT-structure and a Boolean circuit we call the Partial Parameterizable Configuration (PPC). The PPC describes the Boolean dependency of the truth table bits on the parameters as a Boolean circuit that consists of AND and inverter gates. This is also called an AND-Inverter Graph (AIG) [14]. As an example we chose the selection bits of a 4-input multiplexer as parameters and mapped it to 3- LUTs.[1] The resulting fixed LUT structure and AIG of the PPC are shown in Figure 2. We note that making a generic 4-input multiplexer with 3-LUTs takes 6 LUTs, while this datafolded version only takes 2 LUTs. The fixed LUT circuit can be placed and routed on the FPGA fabric using conventional tools. The PPC is compiled to an evaluation function that has to be carried out by the CM. More specifically, the evaluation function consists of C-code that can run on an instruction set processor (ISP). From the locations of the LUTs on the FPGA and the evaluation function of the PPC the specialization procedure is synthesized. The specialization procedure takes the parameters as arguments, generates new truth tables for the reconfigurable module and writes them in the configuration memory. The specialization procedure thus consists of an evaluation of the PPC and a reconfiguration of the truth tables of the fixed LUT circuit.

4. 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 every 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, one 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, one would expect a large reduction

[1] K-LUTs are LUTs with K inputs.

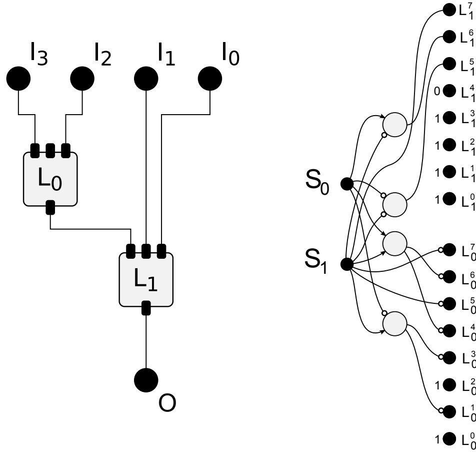

Fig. 2: The fixed LUT structure and AIG of the PPC for the 4-input multiplexer example.

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 [15] 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.

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 multi-output 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 multi-output Boolean function. The PPC will be used by the reconfiguration procedure to generate a new partial configuration for the FPGA. In previous work [16], [5] we have represented the PPC as a vector of closed-form single-output Boolean expressions of the parameter inputs (called *Tuning functions*). In this paper, we represent the PPC as a Boolean network (Figure 2). 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 thus be part of the TC. In other work [17], 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 paper 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. It is important to note here that these algorithms are computationally hard and thus 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. It can be shown [18] 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. The effort spent in the generic stage thus 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.

5. Practical Tool Flow Instance

In the previous section, we presented the general tool flow to map an application to a self-reconfiguring platform. However, to enable a commercial introduction of this tool flow without too many hurdles, we have searched for a practical tool flow that uses current commercial tools as much as possible and only needs a very limited amount of additional tools. The tool flow presented in this section targets Xilinx components and reuses many Xilinx tools.

The self-reconfiguring platform (Fig. 4) targeted by our tool flow is implemented on a Xilinx Virtex-II Pro FPGA. The configuration manager is implemented on an embedded

[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.

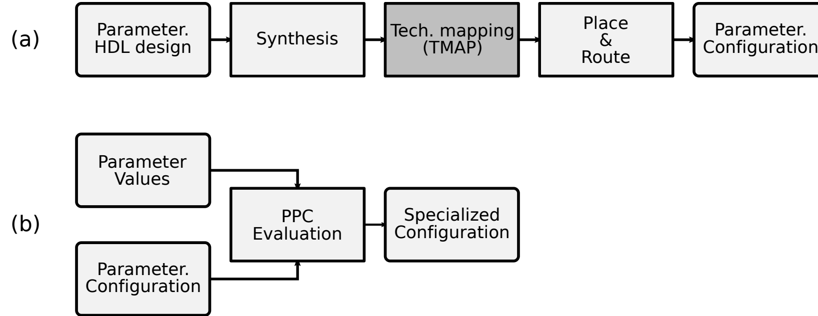

Fig. 3: Overview of our staged mapping tool flow. (a) The generic stage. (b) The specialization stage.

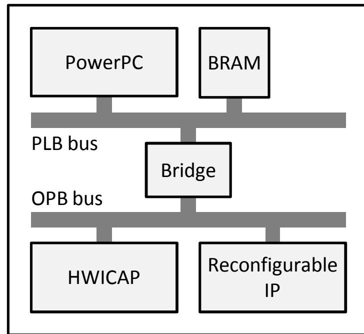

Fig. 4: Self-reconfiguring platform on a Xilinx Virtex-II Pro FPGA.

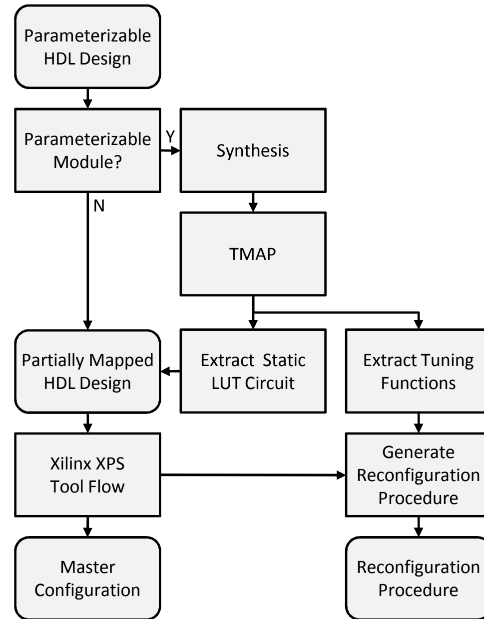

Fig. 5: Practical tool flow for mapping a parameterizable HDL design to a self-reconfiguring platform.

PowerPC of the Xilinx Virtex-II Pro FPGA, which ensures a tight connection to the FPGA fabric [19]. The connection between the configuration manager and the configuration memory is realized through the Xilinx HWICAP module, which provides the interface between the OPB bus and the FPGA's ICAP (Internal Configuration Access Port). To configure parts of the FPGA fabric (LUTs) after a parameter value has changed, the PowerPC evaluates the tuning functions (which are individual PPC instances for every TLUT), generates the new configuration, and sends this new configuration to the FPGA configuration memory through the ICAP port of the FPGA via the HWICAP module. The entire reconfiguration flow is thus executed within the system and no external source is needed to reconfigure the FPGA, nor to take the decision to reconfigure. Therefore, this system is a true *self-reconfiguring* system.

The self-reconfiguring platform shown in Fig. 4 is implemented using Xilinx XPS [20]. The XPS tool flow is implemented in a makefile and it is therefore easy to insert our tools in the flow. The adapted tool flow is shown in Fig. 5.

5.1 Generating a Master Configuration

We assume that the parameterizable HDL design contains a number of parameterizable modules and a number of non-parameterizable modules. A parameterizable VHDL module is nothing more than a regular VHDL description with annotations indicating which of the inputs are the parameter inputs. The parameterizable module of the 6:1 multiplexer example we will use in this section is shown in Fig. 6. The annotation `--PARAM` indicates that the select inputs are parameters. As the annotations are in a comment line, any conventional synthesis tool can be used to synthesize the circuit. We used Altera Quartus II because it can dump a .blif file that can then be used as input for our mapper TMAP [21], which maps the circuit to a TLUT circuit.

We make a distinction between parameterizable modules and non-parameterizable modules. Indeed, the Virtex-II Pro architecture is a very heterogeneous architecture compared to the homogeneous LUT architecture that TMAP targets. Therefore, using TMAP to map the full design would result in a very inefficient use of the Virtex-II Pro architecture. We thus limit the use of TMAP to the parameterizable modules,

```
entity mux6 is
port(
  s : in  std_logic_vector(2 downto 0); --PARAM
  i : in  std_logic_vector(5 downto 0);
  o : out std_logic);
end mux6;

architecture behavior of mux6 is
begin
  o <= i(conv_integer(s));
end behavior;
```

Fig. 6: Parameterizable VHDL module of the 6:1 multiplexer example.

as is shown in Fig. 5. The static LUT circuit of these modules is expressed in VHDL by directly instantiating LUTs in the VHDL module. Combining these modules with the non-parameterizable VHDL modules of the original design forms the partially mapped HDL design. This VHDL design can now be efficiently mapped to the Virtex-II Pro architecture by the Xilinx tools without corrupting the mapping done by TMAP. The result of this last mapping is the master configuration. This workaround could of course be avoided if the ability to map to TLUTs would be incorporated in the Xilinx mapper.

It is important to note that every LUT instantiated in VHDL is given a unique name. This enables our tools to find the LUT's location after place and route, see Section 5.2. Although it is not strictly necessary, we also lock the pins of the LUTs with the `lock_pins` attribute so that the router does not interchange the pins during routing. This greatly simplifies generating the reconfiguration procedure.

5.2 Generating the Reconfiguration Procedure

The reconfiguration procedure reconfigures all the TLUTs instantiated in a parameterizable module according to the parameter values that are passed as arguments to the procedure.

We need both the tuning functions of each TLUT and the location of each TLUT in order to do the reconfiguration upon a parameter change. The tuning functions for each TLUT are provided by TMAP, this is explained in detail in [21]. The LUT locations are harder to come by. On the Virtex-II Pro a LUT location is specified by a slice row, a slice column and whether it's the F or the G LUT of the slice [22]. Finding these locations for each instantiated LUT is done in the following way. The Xilinx tool flow generates a .NCD file that contains all the information on the mapped circuit including the location of the LUTs. This .NCD file is first converted to a .XDL file, a clear-text representation of the .NCD file, using the Xilinx XDL program [23]. We find the LUT locations in this .XDL file by searching the unique names given to the LUTs when they were instantiated in VHDL, as explained in Section 5.1.

A reconfiguration procedure is then generated as follows. For each of the TLUTs in a parameterizable module we

```
void L1( XHwIcap *hwIcap,
         Xuint8 S0, Xuint8 S1, Xuint8 S2) {

  Xuint8 truthTable[LUT_SIZE];
  truthTable [0] = !(0);
  truthTable [1] = !(S0 && S1);
  truthTable [2] = !(!S0 && S1);
  truthTable [3] = !(S1);
  truthTable [4] = !(S0 && !S1);
  truthTable [5] = !(S0);
  truthTable [6] = !((!S0 && S1) || (S0 && !S1));
  truthTable [7] = !( S0 || S1);
  truthTable [8] = !(!S0 && !S1);
  truthTable [9] = !((S0 && S1) || (!S0 && !S1));
  truthTable [10]= !(!S0);
  truthTable [11]= !(!S0 || S1);
  truthTable [12]= !(!S1);
  truthTable [13]= !(S0 || !S1);
  truthTable [14]= !(!S0 || !S1);
  truthTable [15]= !(1);

  XHwIcap_SetClbBits( hwIcap, 31, 45, G_LUT,
                      truthTable, LUT_SIZE);
}
```

Fig. 7: The TLUT reconfiguration procedure for LUT L_1 of our 6:1 multiplexer example. We assume that LUT L_1 is located in the G LUT of the slice at row 31 and column 45.

generate a TLUT reconfiguration procedure that takes the module parameter values as inputs, evaluates the tuning functions generated by TMAP and reconfigures the LUT. The TLUT reconfiguration procedure for LUT L_1 of our 6:1 multiplexer example is shown in Fig. 7. The code that evaluates the tuning functions of a TLUT is generated by simply translating the expressions produced by TMAP into C-style expressions.[3] When executed, these expressions result in a new truth table for the LUT. The reconfiguration of the LUT is then done by calling the procedure `XHwIcap_SetClbBits`, which is provided by Xilinx in the HWICAP module driver. This procedure takes the LUT location and the new truth table to reconfigure the LUT. In our example we assume that LUT L_1 is located in the G LUT of the slice at row 31 and column 45. The reconfiguration procedure for a module simply calls the TLUT reconfiguration procedure for each of the TLUTs of a module. The reconfiguration procedure for our 6:1 multiplexer example is shown in Fig. 8.

On a last practical note, we should warn the reader that, in the Virtex-II Pro family, reconfiguring a LUT will cause corrupted data in the SRL16s and LUT RAMs that are located in the same column. Therefore, placing TLUTs in the same columns as SRL16s or LUT RAMs must be avoided. This can be done using `AREA_GROUP` constraints. This is no longer an issue in the Virtex-5 family.

[3] It must be noted that, since the Virtex-II Pro family LUT configurations are stored in an inverted way, the configuration data must be inverted before configuring the LUTs [24].

```
void mux2w1 ( XHwIcap *hwIcap,
              Xuint8 S0, Xuint8 S1, Xuint8 S2) {
  L0(hwIcap, S0, S1, S2);
  L1(hwIcap, S0, S1, S2);
}
```

Fig. 8: The reconfiguration procedure for our 6:1 multiplexer example.

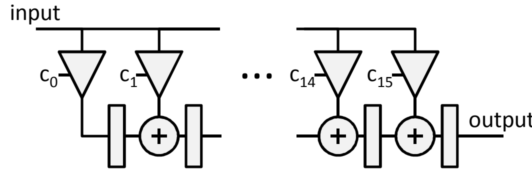

Fig. 9: 32-tap fully pipelined adaptive FIR filter.

6. Experiments and results

In this section, we apply tunable LUT mapping on more complex circuits. In Section 6.1 we create adaptive FIR filters that are adapted by means of reconfiguration and in Section 6.2 we create TCAMs of which the content is written by means of reconfiguration. Both designs are implemented on a Virtex-II Pro XC2VP30 using Xilinx ISE 9.2.

6.1 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 9 for a 32-tap filter.

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 3 and 4 of Table 1.

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 an .aig file using ABC [14]. Together with a list of parameter inputs (the filter coeficients in this case) this .aig file is used as input for

Table 1: Hardware properties for a set of differend-sized adaptive FIR filters implemented without using dynamic reconfiguration (Conventional) and with using dynamic reconfiguration (TMAP). The numbers between brackets are relative compared to the conventional implementation.

Size		Conventional		TMAP			
Width	Taps	LUTs	f [MHz]	LUTs		f [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)

a Java implementation of TMAP which produces both a PPC represented as a .aig file and a LUT structure. In this experiment, the LUT structure is represented as VHDL file that directly instantiates Virtex-II Pro LUTs and FFs [25]. 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 [13].

The number of LUTs and the maximum clock frequency for the dynamically reconfigurable filters can be found in columns 5 and 6 of Table 1. 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 implementation. 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 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 2, 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 paper. 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 an 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. E.g. if left predecessor is inverted, the smallest available index is 3 and and the output values of the left and right predecessors are respectively stored 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 further optimize this 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 2. We see that the evaluation time for our adaptive filters takes in the order several hundreds of μs 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 size of the PPC because one multiplier evaluation function is created, that 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 through the 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 5 and 6 of Table 3. 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 one 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 [26]. 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 it's frames are smaller, the importance of this overhead is reduced for the Virtex-5 [27].

Table 2: Evaluation of the PPC of different-sized adaptive FIR filters on the hardwired PowerPC of the Virtex-II Pro.

Size		Evaluation Time			Program Size
Width	Taps	$\lvert PPC\rvert$	t_{eval} [μs]	$\frac{t_{eval}}{\lvert PPC\rvert}$ [$\frac{ns}{AND}$]	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

Table 3: Reconfiguration of different-sized adaptive FIR filters through the 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

Finally, the total specialization overhead which is the sum of the evaluation time and the reconfiguration time, is shown in Table 4. As can be seen, the specialization time is of the order of a few ms, depending on the size of the filter. We can thus exploit the area and clock frequency benefits of our adaptive filters (Table 1) as long as the time in between coefficient changes is a few orders of magnitude higher that the specialization overhead.

6.2 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

Table 4: Specialization overhead of different-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

a given address. CAMs have many applications [28]. The most important commercial application is packet forwarding in network routers [29].

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. 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.

In a conventional TCAM implementation, the pattern 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 will thus be provided by means of reconfiguration. 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 [30]), 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. When we synthesize a description of a full TCAM (256 entries of 32-bit) 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 the TCAM (see Table 5).

These resource requirements can be drastically reduced with the use of TMAP. In the TMAP design, 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 5). This reduction in FFs is possible because the pattern information is no longer stored in 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. Only a few FFs are left for some output registers.

Because of the importance of TCAMs and the high resource usage of architecture independent HDL implementations, FPGA vendors offer TCAM constructor software which constructs TCAM structures that are highly optimized for a specific architecture [28], [31]. The designer can generate such a structure using the software and then instantiate it in his design. A good example of such a generator is the SRL16 TCAM generator [11] embedded in Xilinx Coregen, which generates TCAM structures that are very similar to the TCAMs that TMAP synthesizes. The results for the SRL16 TCAM are shown in Table 5. As can be seen the SRL16 TCAM (256 entries of 32-bit) 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 of 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 is explained in Section 6.1, the evaluation of the PPC is done on the embedded PowerPC and the reconfiguration is done using the ICAP of the Virtex-II Pro. We did the measurement both in the case only one entry needs to be rewritten and in the case all entries are rewritten. The results are shown in Table 6. If only one entry is written, the reconfiguration time depends on the way the TLUTs of the entry are placed on the FPGA, because the placement determines the number of frames that need to be reconfigured. Table 6 shows the reconfiguration time for the worst case entry. For the largest TCAM (256 entries of 32 bit), reconfiguring one entry takes 245 μs in 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 as its VHDL code 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 our code to architectures with different LUT sizes. The LUT usage for $K = 3$, $K = 4$ and $K = 5$ can be found in Table 7. In the table one can clearly see that for the TCAMs the relative area gain improves when the LUT size increases.

Table 5: Comparison of 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 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

Table 6: Specialization overhead of different-sized TCAMs implemented on the Virtex-II Pro. (One Entry) Only one of the TCAM entries is written. (All Entries) All the 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 7: Several TCAMs mapped to different-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)

7. Conclusions

Run-time hardware reconfiguration provides ample opportunities for optimizations of an implementation in time intervals in between two parameter changes. In this paper 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 of which the truth table is expressed as function of the parameter inputs. We have effectively integrated our tool flow in the Xilinx XPS tool flow that targets Virtex-II Pro FPGA devices. We used the embedded PowerPC of the Virtex-II Pro device as reconfiguration manager. On top of this, in our DDF architecture, specialized configurations are also 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. Our approach is validated by implementing adaptive FIR filters and Ternary Content-Addressable Memories (TCAMs) on a Virtex-II Pro.[4] We show large reductions in the number of LUTs (39% for the FIR filter and 66% for the TCAMs) and significant improvements of 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. Because of 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 [32], regular expression matching [33], DNA aligning [34], [35], serial fault emulation [36] and many others.

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

References

[1] S.V. Gheorghita et al., *System-scenario-based design of dynamic embedded systems.*, ACM Transactions on Design Automation of Electronic Systems, 14(1):1–45, 2009.

[2] G. Palermo, C. Silvano, and V. Zaccaria. *Robust optimization of SoC architectures: A multi-scenario approach.* In Proc. of the IEEE Workshop on Embedded Systems for Real-Time Multimedia, 2008.

[3] J. Paul, D. Thomas, and A. Bobrek. *Scenario-oriented design for single-chip heterogeneous multiprocessors.* In IEEE Trans. on Very Large Scale Integration Systems, 14(8), PP. 868–880, 2006.

[4] K. Compton and S. Hauck. *Reconfigurable computing: a survey of systems and software.* ACM Computing Survey, 34(2): pp. 171–210, 2002.

[5] K. Bruneel and D. Stroobandt. *Reconfigurability-aware structural mapping for LUT-based FPGAs*. In 2008 International Conference on Reconfigurable Computing and FPGAs (ReConFig). IEEE, Piscataway, NJ, USA, 223–8.

[6] L. Benini, D. Bertozzi, and M. Milano. *Resource management policy handling mulitple use-cases in MPSoC platforms using constraint programming.* In Logic Programming, vol. 5366, pp. 470 – 484, 2008.

[7] E.W. Brião, D. Barcelos, and F.R. Wagner. *Dynamic task allocation strategies in MPSoC for soft real-time applications.* In Proc. of the conference on Design, Automation and Test in Europe (DATE), pp. 1386–1389, 2008.

[8] P. Hölzenspies, J. Hurink, J. Kuper, and G. Smit. *Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (MPSoC).* In Proc. of the Conf. on Design, Automation and Test in Europe (DATE), 2008.

[9] V. Nollet. *Run-time Management for Future MPSoC platforms.* Ph.D. thesis, TU Eindhoven, 2008.

[10] P. Yang and F. Catthoor; *Pareto-optimization-based run-time task scheduling for embedded systems.* In Proc. of the Int. Conference on Hardware/Software Codesign and System Synthesis, 2003.

[11] J.-L. Brelet and B. New, B. *XAPP203: Designing Flexible, Fast CAMs with Virtex Family FPGAs*. Xilinx. 1999

[12] M.J. Wirthlin, *Constant coefficient multiplication using look-up tables*. Journal of VLSI Signal Processing, vol. 36, no. 1, PP. 7–15, 2004.

[13] K. Bruneel, F. Abouelella, and D. Stroobandt. *Automatically mapping applications to a self-reconfiguring platform.* In Proceedings of Design, Automation and Test in Europe, K. Preas, Ed. Nice, 964–969. 2009

[14] *ABC: A System for Sequential Synthesis and Verification*. Berkeley Logic Synthesis and Verification Group.

[15] A. Derbyshire, T. Becker, and W. Luk. *Incremental elaboration for run-time reconfigurable hardware designs.* In CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems. ACM, New York, NY, USA, 93–102. 2006

[16] K. Bruneel and D. Stroobandt. *Automatic generation of run-time parameterizable configurations.* In Proceedings of the 2008 International Conference on Field Programmable Logic and Applications, U. Kebschull, M. Platzner, and T. J., Eds. Kirchhoff Institute for Physics, Heidelberg, 361–366. 2008

[17] K. Bruneel and D. Stroobandt. *TROUTE: a reconfigurability-aware FPGA router.* In Lecture Notes in Computer Science. Vol. 5992. Springer Verlag Berlin, Berlin, Germany, 207–218. 2010

[18] K. Bruneel, W. Heirman and D. Stroobandt. *Dynamic Data Folding with Parameterizable FPGA COnfigurations*. To be published in ACM Trans. on Design Automation of Embedded Systems, 2011.

[19] B. Blodget, P. James-Roxby, E. Kelle, S. McMillan, and P. Sundararajan, *A selfreconfiguring platform*, International Conference on Field-Programmable Logic and Applications, pp. 565– 574, 2003.

[20] *Embedded System Tools Reference Manual*, Xilinx.

[21] K. Bruneel and D. Stroobandt, *Automatic generation of run-time parameterizable configurations*, in Proceedings of the International Conference on Field Programmable Logic and Applications, 2008, pp. 361–366.

[22] *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, Xilinx.

[23] J.-B. Note and Éric Rannaud, *From the bitstream to the netlist*, in FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays. New York, NY, USA: ACM, 2008, pp. 264–264.

[24] A. Upegui and E. Sanchez, *Evolving hardware by dynamically reconfiguring Xilinx FPGAs*, in Evolvable Systems: From Biology to Hardware, ser. LNCS, J. M. et al., Ed., vol. 3637. Berlin Heidelberg: Springer-Verlag, 2005, pp. 56–65.

[25] *Virtex-II Pro Libraries Guide for HDL Designs*. Xilinx. 2008.

[26] *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*. Xilinx. 2007

[27] *Virtex-5 FPGA Configuration User Guide*. Xilinx. 2010

[28] *Application Note 119: Implementing High-Speed Search Applications with Altera CAM*. Altera. 2001

[29] K. Pagiamtzis and A. Sheikholeslami. *Content-addressable memory (CAM) circuits and architectures: A tutorial and survey.* IEEE Journal of Solid-State Circuits 41, 3, 712–727. 2006

[30] C. Labovitz, G. Malan, and F. Jahanian. *Internet routing instability*. IEEE/ACM Transactions on Networking 6, 5 (Oct.), 515 –528. 1998

[31] *DS253: Content-Addressable Memory v6.1*. Xilinx. 2008

[32] M. J. Wirthlin and B. L. Hutchings. *Improving functional density through run-time constant propagation.* In FPGA '97: Proceedings of the 1997 ACM Fifth International Symposium on Field-Programmable Gate Arrays. ACM, New York, NY, USA, 86–92. 1997

[33] B. Hutchings, R. Franklin, and D. Carver. *Assisting network intrusion detection with reconfigurable hardware.* In Proceedings of the 10th annual IEEE symposium on field-programmable custom computing machines. 111–120. 2002

[34] K. Puttegowda, W. Worek, N. Pappas, A. Dandapani, P. Athanas, and A. Dickerman. *A run-time reconfigurable system for gene-sequence searching.* VLSI Design, International Conference on 0, 561. 2003.

[35] Y. Yamaguchi, Y. Miyajima, T. Maruyama, and A. Konagaya. *High speed homology search using run-time reconfiguration.* In FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications. Springer-Verlag, London, UK, 281–291. 2002.

[36] L. Burgun, F. Reblewski, G. Fenelon, J. Berbier, and O. Lepape. *Serial fault emulation*. In DAC '96: Proceedings of the 33rd annual Design Automation Conference. ACM, New York, NY, USA, 801–806. 1996.