AUTOMATICALLY EXPLOITING REGULARITY IN APPLICATIONS TO REDUCE RECONFIGURATION MEMORY REQUIREMENTS

Fatma Abouelella, Karel Bruneel and Dirk Stroobandt

Department of Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
{fmostafa, kbruneel, dstrooba}@elis.ugent.be

ABSTRACT

Partial reconfiguration (PR) of FPGAs is a very promising technique. Applications implemented with PR are smaller and faster than applications that are not reconfigured. However, the overhead emerging from the reconfiguration process can nullify the benefits of PR. Moreover, the lack of automatic tools hinders the widespread use of the PR technique. In previous work, the PR barriers have been tackled by introducing parameterized configurations and a tool flow that exploits these configurations. For regularly structured applications mapped through this tool flow, the memory resources needed to store the parameterized configuration can be significantly reduced when regularity is exploited. In this paper, we propose a front-end to the tool flow that automatically detects regular structures at the HDL level and transfers those regularities into the reconfiguration process. The results show that a reduction factor of 76, 10 and 167 is achieved in the memory resources needed to store the parameterized configuration when the regularity is exploited for an adaptive FIR, a regular expression matcher and a Ternary Content Addressable Memory (TCAM) respectively. The reduction factor will be further increased when applications scale.

1. INTRODUCTION

Many FPGAs provide support for Partial Reconfiguration (PR). PR is a technique used to modify the functionality implemented by a portion of an FPGA while it is working. The FPGA resources are controlled by the configuration memory. In order to change the functionality implemented by certain FPGA resources, the configuration memory bits that control those resources should be updated by sending a new partial configuration to the FPGA. Circuits implemented using the PR technique reuse parts of the available area and therefore occupy less FPGA resources than their generic implementations. Thus, the use of this technique allows for bigger circuits to be implemented on smaller and cheaper FPGAs.

The benefits of the PR technique come at the cost of an extra overhead (reconfiguration overhead). This overhead contains both the time needed to generate (select) and send a new partial configuration to an FPGA (reconfiguration time) and the resources needed to store partial configurations[1] and to build a configuration manager. Furthermore, the complexity of designing for PR designs and the lack of automatic tools hinder the widespread use of the PR technique.

In [1] and [2], the authors have introduced both a new type of FPGA configurations, so-called parameterized configurations and a tool flow that automatically generates these configurations. A parameterized configuration is a multi-valued Boolean function that expresses a specialized FPGA configuration as a function of some parameters. Each time the parameter values change at run-time, the parameterized configuration is evaluated according to the new parameter values and a specialized configuration is generated.

Although the amount of memory needed to store a parameterized configuration is much less than that needed to store specialized configurations for all possible combinations of the parameter values, the memory needed to store parameterized configuration will still be considerably large as the application scales.

Many applications contain regular structures such as adaptive FIR filters, Ternary Content-Addressable Memory (TCAMs) and Regular Expression (RegExp) matchers. These applications are commonly used in various systems. For instance, TCAMs are used in routers of computer networking devices [3] and Network Intrusion Detection Systems (NIDS) [4] for packet classification. FIR filters are used in most Digital Signal Processing (DSP) systems. RegExp matchers are used in NIDS and DNA sequence alignment systems [5].

In such regularly structured applications, the parameterized configuration memory requirements can be drastically reduced. By exploiting regularity, the parameterized configuration that corresponds to the repetitive module of the

[1] In case all possible partial configurations are generated off-line to avoid long generation times taken by conventional tools.

regular structure needs to be stored only once.

Therefore, we propose an automatic method that detects regular structures in HDL designs and transfers this regularity to the reconfiguration process. This method is integrated in the tool flow explained in [2]. The results show that we gain a factor of 76, 10 and 167 less memory when the regularity is exploited in an adaptive FIR, a RegExp matcher and a TCAM respectively. Moreover, the time needed to generate specialized configuration is also reduced when the regularity is exploited in the three applications.

The paper is organized as follows: in Section 2 some related work is presented. In Section 3 some background on the parameterized configuration and the tool flow presented in [2] is given. In Section 4, we present a concrete description of the problem and our contribution. In Section 5, we present our method for detecting applications' regular structures on the HDL level and for transferring this regularity to the reconfiguration process. In Section 6, the results that confirm our work are shown through three applications. Finally, we conclude in Section 7.

2. RELATED WORK

In FPGAs, high-quality applications can be achieved if researchers have the opportunity to test and proof their ideas on commercial FPGAs. Since free tools from FPGA vendors are not open- source, some researchers have tried to create custom CAD tools for commercial FPGAs. In [6], RapidSmith tool is introduced. This tool is an open-source tool platform for rapidly creating FPGA CAD tools. This tool enables researchers to implement new ideas and algorithms on FPGAs. In [7] a tool flow that automatically generates homogeneous hard macros for Xilinx FPGAs starting from a high-level description is introduced. They designed a homogeneous placer and a suitable router to maintain homogeneity in the generated hard macros. Homogeneous hard macros are very useful in timing critical applications such as time-to-digital converters. In this application, the regularity of the routing structure is used to measure the time difference between two electrical pulses with a certain precision.

3. BACKGROUND

3.1. Parameterized configurations

A parameterized configuration [1] is a multivalued Boolean function that expresses a specialized FPGA configuration as a function of some parameters. These parameters are the slowly varying inputs of the application. Once the parameterized configuration is found, generating a specialized FPGA configuration for given parameter values requires the evaluation of this parameterized configuration for those parameter values. This specialized configuration is used afterwards to reconfigure the FPGA. Obviously the evaluation of Boolean functions is several orders of magnitude faster than the generation of a regular FPGA configuration from scratch using conventional tools. Moreover, specialized configurations result in implementations that are smaller and faster than those resulting from the regular FPGA configurations.

3.2. TLUT tool flow

In order to exploit the concept of parameterized configurations, an automatic tool flow has been introduced in [2]. This tool flow maps parameterized applications to a self-reconfiguring platform. This platform contains a Configuration Manager (CM) and an FPGA. The task of the CM is to generate a specialized configuration each time the parameter values change and to configure the FPGA with that configuration. In [2] the authors assumed that the CM is a process running on an instruction set processor. However, the CM can also be implemented inside the FPGA fabric [8].

The tool flow takes a parameterized HDL design as input and outputs both a template FPGA configuration that configures the FPGA at start-up and a set of reconfiguration functions, which are used by the CM to generate specialized configurations at run-time.

The parameterized HDL design is a regular HDL design with annotated parameter inputs. At design-time, the designer has to select and annotate the slowly changing inputs in the HDL design. In the first step of the tool flow, the parameterized HDL design is synthesized in a conventional way. Afterwards, the design is mapped with a new technology mapper, called TMAP. A conventional mapper maps the design into a LUT circuit, though, TMAP maps the design into a TLUT circuit. The TLUT is like a regular LUT, except its truth table is expressed as functions of parameter inputs [1]. Since parts of the design functionality depending on parameter inputs, are incorporated in truth table bits of the TLUTs, the size of the TLUT circuit is much smaller than the regular LUT circuit for the same design. The next step is to place and route the TLUT circuit. This step can be performed using conventional placement and routing tools when a static LUT circuit is obtained by ignoring that the TLUT truth tables depend on the parameter values. The placed and routed circuit is used to generate the template configuration. On the other hand, the parameterized configuration associated to the TLUT circuit is extracted. This configuration represents the truth table bits of each TLUT in the TLUT circuit. Finally, the parameterized configurations are used to generate the reconfiguration functions.

4. PROBLEM DEFINITION

Although implementations generated by the TLUT tool flow are smaller and faster than these generated by conventional tool flows [1], the reconfiguration overhead can nullify those gains. A main portion in the reconfiguration overhead is the memory needed to store the reconfiguration functions, other overhead portions are discussed elsewhere [9].

Currently the size of the reconfiguration functions is directly proportional to the amount of FPGA resources (TLUTs) that will be reconfigured every time the parameter values change. However, for applications that contain regular structures, we can exploit this regularity in the structure by observing that repetitive modules are similar to each other except for the inputs. This similarity also exists in the reconfiguration functions. Thus, the reconfiguration functions[2] associated to one repetitive module (modular reconfiguration functions) could be used to generate a complete specialized configuration for the whole regular structure. At run-time when a new specialized configuration is needed, those modular reconfiguration functions will be executed a number of times according to the repetition scheme of the repetitive module with a different subset of parameter values each time. This way the memory needed to store the modular reconfiguration functions can be significantly reduced.

In the previously described TLUT tool flow, there is no way to automatically detect the regularity in a design and hence two possible scenarios are expected. In the first scenario, the entire regular structure is mapped in the TMAP stage of the tool flow and the size of the output reconfiguration functions will be large. In the second scenario, if the designer realizes a regularity in the design, he has to manually indicate that the repetitive module should only be mapped in TMAP in order to generate modular reconfiguration functions. In addition, the repetition scheme of the repetitive module in the regular structure has to be manually extracted and combined with the modular reconfiguration functions.

Contribution

The main target of this paper is to automatically exploit regularities existing in applications to reduce the memory needed to store the reconfiguration functions. This will be done by automatically detecting regular structures in applications and transferring these regularities to the reconfiguration process. Since our target is to only map the repetitive module of the regular structure, the detection process should be done before TMAP.

In the TLUT tool flow, synthesis of the parameterized HDL design is the first step as well as the step just before TMAP. Synthesis tools take a design in a high-level description language and generate a flattened netlist (an optimized netlist of gates). During the synthesis of regularly structured applications, repetitive modules within the same regular structure might be optimized in dissimilar ways which makes the detection of the regular structure after synthesis a complex process. It is more logical and easier to detect these regularities on the high-level description of the design where the hierarchy concept is usually used by designers for simplicity and readability.

[2]For the sake of clarity, in the remainder of this paper we will call the reconfiguration functions that correspond to a repetitive module of a regular structure, *modular reconfiguration functions*.

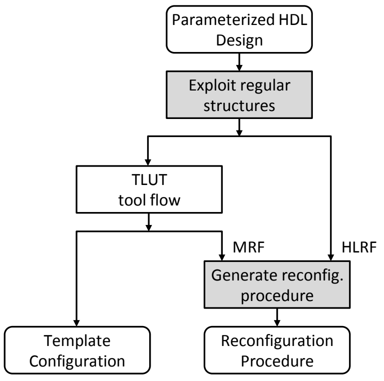
Parameterized HDL Design
Exploit regular structures
TLUT tool flow
MRF
HLRF
Generate reconfig. procedure
Template Configuration
Reconfiguration Procedure

Fig. 1. The TLUT tool flow with regularity exploitation front-end

Fig. 1 shows the integration of exploiting regular structures in the TLUT tool flow. The detection of regular structures is done on the HDL level. Regularities are transferred to the reconfiguration process by the introduction of High-Level Reconfiguration Functions (HLRFs). These functions describe the repetition scheme of the regular structures' repetitive modules. In the TLUT tool flow, since we will only map the repetitive module in TMAP, the output of the tool flow will be a template configuration and a set of Modular Reconfiguration Functions (MRFs). The final step is to generate a reconfiguration procedure. This reconfiguration procedure is code that is executed by the CM every time a specialized configuration is needed. This code is built by combining the high-level reconfiguration functions and the modular reconfiguration functions.

5. METHOD

5.1. Detection of regular structures in HDL designs

Most of the Hardware Description Languages (HDL) such as VHDL [10] and Verilog [11] provide a mechanism to express regular structures. In Verilog, this mechanism is represented by the *generate block* capability, which enables to repetitively (using for-loop statements) and conditionally (using if-else and case statements) generate modules. In VHDL that mechanism is represented by a *generate statement* with two schemes: the repetitive and the conditional scheme. Fig. 2 shows the repetitive scheme in VHDL. This scheme, commonly called the FOR-scheme, allows to replicate a set of concurrent statements (such as component instantiations) a number of times. This number is specified by the range of the loop variable. Similarly, the conditional

```
name    :    FOR variable IN range GENERATE
        --Concurrent-statements
END GENERATE name;
```

Fig. 2. FOR-Scheme generate statement in VHDL

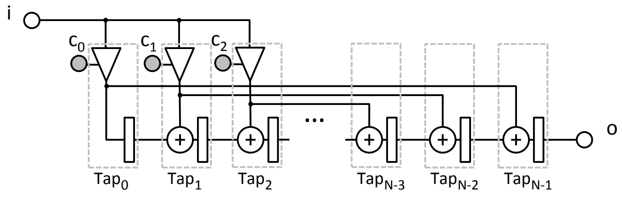

Fig. 3. linear phase FIR filter with N taps

scheme, commonly called IF-scheme, allows variations in the regular structure.

For the sake of simplicity and without loss of generality, in the remainder of the paper we will use VHDL examples and we will assume only one type of concurrent statements (the component instantiation). Thus, we can assume each module in the regular structure consists of a number of component instances, which we will call basic blocks.

An adaptive FIR filter with N taps is a perfect example of an application with a regular structure. The regularity in this type of filters arises from the repetition of the tap structure. In pipelining FIR filters, each tap consists of a multiplier and an adder followed by a register. For example, consider the pipelining linear phase FIR with N taps shown in Fig. 3. Each tap in this filter is highlighted by a dashed border. Since the coefficients of the linear phase FIR filter are symmetrical around the center coefficient (the first coefficient is the same as the last), the multipliers in the last *N-Ceil(N/2)* taps are omitted. Therefore, a variation in the tap construction can be easily noticed in the last *N-Ceil(N/2)* taps. The structural description of this application in VHDL is shown in Fig. 4. This design is built based on tree multipliers. The regularity of the design is expressed by the FOR-scheme of the generate statement. In each loop, the regular structure's repetitive module (tap) is expressed by two basic blocks: the multiplier instance and the seqadder instance (represents an adder followed by a register). The first tap is different from the rest. Variations in the taps' structure are handled by IF-schemes of the generate statement inside the main FOR-scheme. The first IF-scheme expresses the variation in the first tap by introducing the *init* block, which loads the register of the first module by the multiplier output directly. The second IF-scheme describes the regular structure's repetitive module that includes a multiplier and an adder. This IF-scheme generates taps ranging from two to *Ceil(N/2)*. Finally the third IF-scheme describes the last *N-Ceil(N/2)* taps in which the multipliers are omitted.

```
entity FIR is
port (clk : in std_logic ;
  i : in std_logic_vector (DIW-1 downto 0);
  c : in arrayOfStdlogicVectors ; --PARAM
  o : out std_logic_vector (DOW-1 downto 0));
end FIR;
architecture struct of FIR is
  type inter_t is array (0 to N) of std_logic_vector (DOW-1
       downto 0);
  signal inter : inter_t := (others => (others => '0'));
  type mult_t is array (0 to Ceil(N/2)-1) of std_logic_vector
       (2*DIW-1 downto 0);
  signal mult : mult_t;
begin

GS_FOR : for T in 0 to N-1 generate

  GS_IF1 : if (T = 0) generate
    MULTIPLIER0 : entity work.multiplier(rtl )
     port map (i, c(T), mult(T));
    LOAD : entity work.init (rtl )
      port map (clk, mult(T), inter (T+1));
  end generate GS_IF1;

  GS_IF2 : If (0 < T < Ceil(N/2)) generate
    MULTIPLIER1 : entity work.multiplier(rtl )
     port map (i, c(T), mult(T));
    ADDER0 : entity work.seqadder(rtl )
      port map (clk, mult(T), inter (T), inter (T+1));
  end generate GS_IF2;

  GS_IF3 : If (T >= Ceil(N/2)) generate
    ADDER1 : entity work.seqadder(rtl )
      port map (clk, mult(N-T-1), inter(T), inter (T+1));
  end generate GS_IF3;

end generate GS_FOR;
o <= inter(N);
end struct ;
```

Fig. 4. VHDL for N taps linear phase FIR filter. The type arrayOfStdlogicVectors is array (0 to Ceil(N/2)-1) of std_logic_vector(DIW-1 downto 0)

In our method, we detect the regularity in applications by traversing the Abstract Syntax Tree (AST) of the HDL design. Fig. 5 shows the AST of the VHDL representing the linear phase FIR. The *ForGn* node represents the FOR-scheme generate statement named GS_FOR in the VHDL represented in Fig. 4. The body of the *ForGn* illustrates the regular structure module (tap) and contains three *IfGn* nodes. Each *IfGnx* node represents the IF-scheme generate statement named GS_IFx in the VHDL.

The last step in the detection process is to determine the parameterized basic block[3] within the repetitive module. The mapping criteria used in the TLUT tool flow tends to map in TMAP blocks that depend on parameter inputs, more details can be found in [2].

In the linear phase FIR example, the filter coefficients

[3]Basic blocks that depend on parameter inputs.

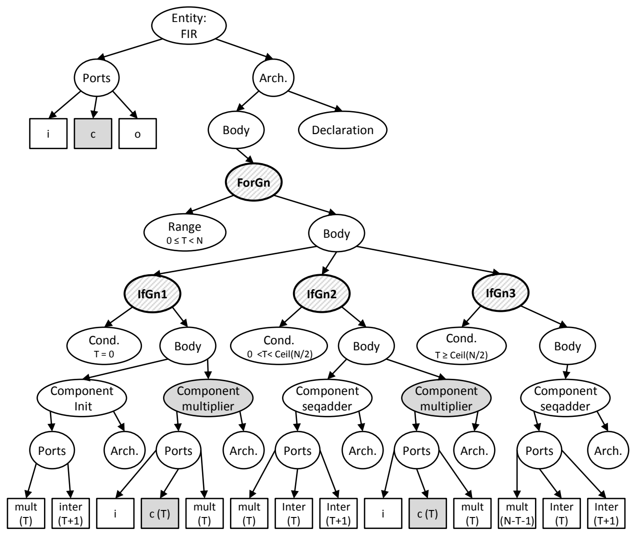

Fig. 5. The AST of the VHDL describing the linear phase FIR. The clk input has not been shown for simplicity

are the parameters. The AST of the FIR (shown in Fig. 5) illustrates that the parameter inputs are in the port list of the multiplier basic block (shown in grey), which indicates the parameterization of the multiplier basic block.

5.2. Generation of a static HDL template

The next step is to generate a static HDL template. This template represents the static parts of the HDL design. It is implemented using conventional tools (in the TLUT tool flow) to generate a template FPGA configuration. In our method, we start from the AST of the original HDL design[4] in order to generate a HW AST, which can be easily converted to a static HDL template. The HW AST is an AST similar to that of the original HDL design except for the parameter inputs and the parameterized basic blocks. In the HW AST, all parameter inputs that exist in the original AST are removed. In addition, each parameterized block, which will be mapped using TMAP (in the TLUT tool flow), is replaced by a TMAP template block. The ports of the TMAP template are compatible with the ports of the corresponding parameterized basic block minus the parameter inputs. The architecture of the TMAP template block will be found after mapping of the parameterized block in TMAP.

For our FIR example, the HW AST can be easily extracted from the original AST shown in Fig. 5. Ports in grey, which represent the parameters (the coefficients of the FIR filter), should be removed. Entities in grey, which represents the parameterized basic block of the regular structure (multiplier) should be replaced by a TMAP template of the multiplier.

[4]Without loss of generality, we assume an application with one regular structure.

5.3. Generation of high-level reconfiguration functions

The last step in our method is to transfer the regularity detected in an application to the reconfiguration procedure. This will be done by generating high-level reconfiguration functions for the modular reconfiguration functions that correspond to the parameterized basic blocks of regularly structured modules. For each regularly structured module a high-level function is generated. This function mainly calls the modular reconfiguration functions associated to the module's parameterized basic blocks. The way the modular reconfiguration functions are called depends on the hierarchy of the parameterized basic blocks in the regular structure module. In our method, we extract this hierarchy from the AST of the original HDL design. We generate a SW AST that describes the regularity in a structure. This SW AST can be easily used to generate the high-level reconfiguration function associated to a certain regular structure.

Regularity in structures is expressed mainly by a FOR-scheme generate statement (in VHDL). Nested FOR-scheme generate statements and IF-scheme generate statements inside FOR-scheme generate statements may exist. In our method, we start from the outer FOR-scheme generate statement. Each FOR-scheme generate statement (represented by ForGn in the original AST) is mapped to a for-statement in the SW AST. The variable range of this loop is identical to the range of the corresponding FOR-scheme generate statement in the original AST. The order of manipulating the variable range is similar to that in the corresponding FOR-scheme generate statement. In addition, each IF-scheme generate statement (represented by IfGn in the original AST) is mapped to an if-statement in the SW AST. The condition of the if-statement is identical to that of the corresponding if-scheme generate statement. Parameterized basic blocks are only mapped to the SW AST by calling the corresponding modular reconfiguration functions. The parameter input ports in the parameterized basic blocks are used as arguments for the high-level reconfiguration functions.

Fig. 6 shows the SW AST of the linear phase FIR. This AST is very simple since there is only one parameterized block (multiplier) in each tap. The parameters (the coefficients of the filter) are the arguments of the high-level reconfiguration function represented by the SW AST. One for-statement and two if-statements, which correspond to *ForGn*, *IfGn1* and *IfGn2* nodes respectively, are used to represent the regular structure. The third IF-scheme generate statement (*IfGn3*) has not been mapped to the SW AST since it does not enclose parameterized blocks.

Finally, Fig. 7 shows the high-level reconfiguration function that corresponds to taps of the linear phase FIR filter example in C. The modular reconfiguration function of the multiplier, generated by the TLUT tool flow, is called in two

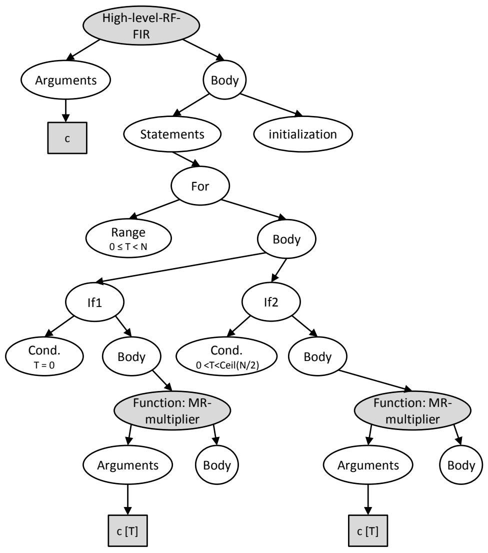

Fig. 6. The SW AST of the linear phase FIR

```
void High−Level−RF−FIR(int c[Ciel(N/2)][DIW]){
        for  (T = 0;  T< N; T++){
             if   (T == 0)
                  MR−multiplier (c[T]);
             if (  T>0 && T<Ceil(N/2))
                  MR−multiplier (c[T]);
        }
}
```

Fig. 7. The high-level function for the linear phase FIR in C

if-statements. [5] In the function, it is easy to notice the similarity to the HDL of the FIR shown in Fig. 4.

6. EXPERIMENTS

In this section, we show the results of applying our method. Three applications: a Finite Impulse Response (FIR) filter, a Ternary Content-addressable Memory (TCAM) and a Regular Expression matcher (RegExp) are used. For each application, different implementation methodologies are applied. The first methodology implements the entire application using Xilinx conventional tools without the use of the PR technique. The second methodology implements the application in the TLUT tool flow explained in [2]. The third methodology implements the application in the TLUT tool flow with the front-end (TLUT w.f.e.) explained in this paper.

The self-reconfiguring platform targeted by the TLUT tool flow is implemented on a Xilinx Virtex-II Pro (XCVP30) FPGA. The configuration manager is implemented on the embedded PowerPC of the Xilinx Virtex-II Pro FPGA. The PowerPC is running at 300Mhz with caches (instruction and data) enabled. More details regarding the platform can be found in [2]. In these experiments we focus on both the reconfiguration area overhead and the time needed to generate specialized configuration, the optimizations of the reconfiguration time overhead can be found elsewhere [9]. For each methodology applied to a certain application, we will measure the design area, which is represented by the number of LUTs, of the entire application, the memory needed to store the reconfiguration procedure, the maximum operating frequency and finally the time needed to generate a specialized configuration (the execution time of the reconfiguration procedure).

[5]The compiler will take care of the optimization.

6.1. Adaptive Finite Impulse Response (FIR) filter

Table 1. 64-tap FIR (8-bit input and 8-bit coefficients)

Methodology	Design area (LUTs)	Recon.P. memory (KByte)	Freq. (MHz)	Generation time (us)
Conventional	7578	-	85.44	-
TLUT	5080	742.67	195.72	21731.92*
TLUT w.f.e.	5080	9.85	185.61	982.22

Throughout the paper we have explained the linear phase FIR filter, which is a special case of the general FIR filter. However, in the experiments we will focus on the general FIR filter. In this experiment, we select the filter coefficients to be the parameters.

Table 1 shows the design area of the FIR filter, reconfiguration procedure memory size, the maximum frequency of the design and the time needed to generate a complete specialized configuration each time the parameter values change. Each row in the table represents results obtained by applying a certain methodology.

The table shows both a 32% reduction in number of LUTs representing the design area and a ∼50% increase in the maximum frequency of the FIR when the parameterized configuration concept is applied in the last two methodologies. The memory needed to store the reconfiguration procedure is significantly reduced (by a factor of 76) when the new methodology is applied and the regularity is exploited. Moreover, the generation time of the specialized configuration is reduced by factor of 22. However, we should note that the maximum size of the local memory available for the PowerPC in the XC2VP30 device is limited to 128 KByte, therefore, for the amount of reconfiguration procedure memory that exceeds 128 KByte an external memory (slower than the local memory) is used. This explains the large reduction in the generation time when the regularity is exploited and the reconfiguration procedure fits inside the local memory of the PowerPC. The generation time associated to a reconfiguration procedure that is stored in an external memory is annotated by an * symbol in the table.

6.2. Regular expression (RegExp) matcher

In this experiment, we will make use of the Regular Expression (RegExp) matcher design presented in [12]. In this design the RegExp is built by cascading Generic Blocks (GB). These GBs are able to implement regular expressions and are designed to be reconfigured by changing some of their inputs, called the configuration inputs. In this experiment, we select those configuration inputs as the parameter inputs. The regularity in this application arises from the repetition of the GBs.

Table 2 shows the results of a RegExp matcher with 13 generic blocks[6] when the three methodologies are applied. The table shows that the design resources are reduced by 44-46% when the last two methodologies are applied. The memory needed by the reconfiguration procedure is decreased by a factor of 10. The reason for the decrease in the clock frequency when the new methodolgy is applied is due to the longest path passing through the 13 generic block. Thus mapping the GBs separately in TMAP prevents the possible optimizations in the longest path delay. Finally, we can notice that although the reconfiguration procedures in the last two methodologies fit inside the local memory of the PowerPC, the generation time is also reduced by a factor of 3. This reduction results from the fact that if the reconfiguration procedure is small enough, it can fit entirely in the instruction cache and hence the execution time is reduced.

Table 2. Regular expression matcher with 13 generic blocks

Methodology	Design area (LUTs)	Recon.P. memory (KByte)	Freq. (MHz)	Generation time (us)
Conventional	709	-	14.8	-
TLUT	392	37.29	78.55	95.44
TLUT w.f.e.	377	3.4	49.21	26.58

6.3. Ternary Content-addressable Memory (TCAM)

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.

*This generation time is associated to a reconfiguration procedure that is stored in an external memory.

[6]In the conventional implementation, RegExp matcher with 13 Generic blocks is the largest possible choice that can fit into the XCVP30 FPGA without exceeding the available IOB components.

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 this experiment, we have built the read operation of the TCAM in the conventional way, while the write operation is built by means of reconfiguration. To handle the reconfiguration, an array of pattern entries is added as an input to the read module. Those pattern entries are selected to be the parameters. The regularity in this application arises from the repetition of the submodule that compares the input data to one pattern entry.

Table 3 shows a reduction of 62% in the design area of the TCAM when the parameterized configuration concept is applied in the last two methodologies. The memory needed by the reconfiguration procedure is reduced by a factor of 167 when the new methodolgy is applied. In this case, the frequency of the design even increases using our new methodology. The generation time is reduced by a factor of 31.

Table 3. TCAM dataWidth = 32 and entries = 256

Methodology	Design area (LUTs)	Recon.P. memory (KByte)	Freq. (MHz)	Generation time (us)
Conventional	10871	-	65.47	-
TLUT	4036	543.39	67.38	15123.66*
TLUT w.f.e.	4034	3.25	78.98	480.44

7. CONCLUSION

In [2], the TLUT tool flow is introduced. This tool flow maps applications that have some slowly varying inputs (called parameters) to a self-reconfiguring platform. The tool flow takes a parameterized HDL design as input and outputs both a template FPGA configuration that is used to configure the FPGA at start-up and a set of reconfiguration functions, which are used to generate specialized configurations each time parameter values change at run-time.

In this paper, we introduced an automatic method that exploits the regularity existing in the structure of some applications. The regularity exploitation reduces significantly the memory needed to store the reconfiguration functions generated by the TLUT tool flow. The method starts by detecting parameterized regular structures at the HDL level.

By transforming the Abstract Syntax Tree (AST) of the HDL design the repetition scheme of the regular structure's repetitive module can be extracted. This repetition scheme is combined with the reconfiguration functions generated by the TLUT tool flow to form the reconfiguration procedure. The results have shown that the memory resources needed to store the reconfiguration procedure is reduced by a factor of 76, 10 and 167 when the regularity is exploited in the FIR, the RegExp matcher and the TCAM respectively. Moreover, the time needed to generate specialized configuration is reduced by at least a factor of 3 when the regularity is exploited. Finally, we should note that when regularly structured applications grow larger, the memory use of the technique described in [2] grows very large. The technique described in this paper solves this problem by automatically exploiting the regularity in the original design.

Acknowledgment

This work was supported by the European Commission in the context of the FP7 FASTER project (#287804).

8. REFERENCES

[1] 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. Heidelberg: Kirchhoff Institute for Physics, 9 2008, pp. 361–366.

[2] 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, 4 2009, pp. 964–969.

[3] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[4] M. Gao, K. Zhang, and J. Lu, "Efficient packet matching for gigabit network intrusion detection using TCAMs," *International Conference on Advanced Information Networking and Applications*, vol. 1, pp. 249–254, 2006.

[5] B. Brown and Y. Yin, M.L.and Cheng, "DNA sequence matching processor using FPGA and JAVA interface," *IEMBS '04. 26th Annual International Conference of the IEEE*, vol. 2, 2004.

[6] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "RapidSmith: Do-it-yourself CAD Tools for Xilinx FPGAs." in *Proceedings of the 2011 International Conference on Field Programmable Logic and Applications*. IEEE, 2011, pp. 349–355.

[7] S. Korf, D. Cozzi, M. Koester, J. Hagemeyer, M. Porrmann, U. Rückert, and M. D. Santambrogio, "Automatic HDL-based generation of homogeneous hard macros for FPGAs," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 2011, pp. 125–132.

[8] F. Abouelella, K. Bruneel, and D. Stroobandt, "Efficiently generating FPGA configurations through a stack machine," in *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, Milano, 2010.

[9] B. Al Farisi, K. Heyse, K. Bruneel, and D. Stroobandt, "Memory-efficient and fast run-time reconfiguration of regularly structured designs," in *Field Programmable Logic and Applications, 21st International conference, Proceedings*, 2011, pp. 171 – 176.

[10] *IEEE Standard VHDL Language Reference Manual (1076-2000)*, IEEE, 2000.

[11] *IEEE Standard Verilog Hardware Description Language (1364-2001)*, IEEE, 2001.

[12] T. Davidson, M. Merlier, K. Bruneel, and D. Stroobandt, "Dynamically reconfigurable pattern matcher for regular expressions on FPGA," in *ParCo*, 2011, p. 8.