Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 716984, 13 pages
doi:10.1155/2012/716984

Research Article

Dynamic Circuit Specialisation for Key-Based Encryption Algorithms and DNA Alignment

Tom Davidson, Fatma Abouelella, Karel Bruneel, and Dirk Stroobandt

ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium

Correspondence should be addressed to Tom Davidson, tom.davidson@ugent.be

Received 30 April 2011; Revised 30 August 2011; Accepted 3 September 2011

Academic Editor: Marco D. Santambrogio

Parameterised reconfiguration is a method for dynamic circuit specialization on FPGAs. The main advantage of this new concept is the high resource efficiency. Additionally, there is an automated tool flow, *TMAP*, that converts a hardware design into a more resource-efficient run-time reconfigurable design without a large design effort. We will start by explaining the core principles behind the dynamic circuit specialization technique. Next, we show the possible gains in encryption applications using an AES encoder. Our AES design shows a 20.6% area gain compared to an unoptimized hardware implementation and a 5.3% gain compared to a manually optimized third-party hardware implementation. We also used *TMAP* on a Triple-DES and an RC6 implementation, where we achieve a 27.8% and a 72.7% LUT-area gain. In addition, we discuss a run-time reconfigurable DNA aligner. We focus on the optimizations to the dynamic specialization overhead. Our final design is up to 2.80-times more efficient on cheaper FPGAs than the original DNA aligner when at least one DNA sequence is longer than 758 characters. Most sequences in DNA alignment are of the order 2^{13}.

1. Introduction

Parameterised configurations are a new concept for dynamic circuit specialization that uses FPGA reconfiguration. It was developed to use run-time reconfiguration (RTR) to dynamically specify the design [1]. This concept is implemented in the *TMAP* tool flow, an alternative to the normal FPGA tool flow. The *TMAP* tool flow allows us to automatically make a run-time reconfigurable design, based on the original design. Its principles and advantages are discussed in section 2. Because this is a new technique, there is still a lot of exploration needed to fully understand how it should be used and what the potential gains are for different applications. This paper shows that in at least two fields, key-based encryption and DNA alignment, substantial gains can be made using parameterised configurations. The *TMAP* tool flow allows us to check very quickly whether or not a certain implementation is suitable for dynamic circuit specialization or not. In Section 3 we will discuss the similarities between this tool flow and hardware/software partitioning that does not use run-time reconfiguration. There, we will also show that using parameterised configurations extends the hardware/software boundary.

To explain how parameterised configuration can be used in encryption applications, we start with a straightforward implementation of the Advanced Encryption Standard. AES is an encryption algorithm detailed by the NIST in [2] and is explained in more detail in Section 4. In this section we will also detail the design decisions for our own AES implementation, k_AES. Next, we will explain how exactly the *TMAP* tool flow is used on this application to make the design run-time reconfigurable. In Section 5 we will discuss the result of applying *TMAP* to k_AES. We show a 20.9% area gain for the k_AES implementation, compared to the normal FPGA tool flow. In addition, we will compare this result with four other, manually optimized, designs, Crypto-PAn [3], Avalon_AES [4], AES128 [5], and AES_core [6]. We will show results. We will also use *TMAP* on the manually optimized designs, and discuss why this does not result in significant gains. Finally, allowing a design to be dynamically specialized will always introduce a specialization overhead, we will discuss this overhead for the AES implementation in this section.

In Section 6 we will expand our results for the AES algorithm to two other key-based encryption algorithms. For the TripleDES [7] encryption algorithm we see a 27.2% LUT-area gain for using parameterisable configuration. The second algorithm is the RC6 [8] encryption algorithm, here we have a 72.7% LUT-area gain.

In addition to the encryption algorithms we will also discuss an example of a specialised string matching algorithm, a DNA aligner. DNA alignment is discussed in Section 7. In this case, we followed a different path than with the AES application. We started with an existing DNA aligner, changed the control flow slightly, and applied the *TMAP* tool flow. The resulting RTR aligner does not show a reduction in the number of LUTs, but does reduce the number of used BRAMs significantly. It changes the resource tradeoff of the design. The RTR aligner can be implemented on a much cheaper FPGA. We will discuss the details of the RTR aligner and its advantages in comparison with the original design in Section 8. In Section 8.3 we will discuss the test-setup and experimental results for the RTR aligner, with concrete numbers on reconfiguration time as measured on an actual FPGA.

In the case of the DNA aligner, the specialisation overhead has a large impact on the design. We will discuss this overhead in detail in Section 9. In this section we will also discuss how to reduce this specialisation overhead using several optimizations.

2. Parameterised Configurations and *TMAP*

FPGAs can be configured to implement any function, as long as there are enough FPGA resources available. The functions on the FPGA are completely controlled by memory. This memory is called the configuration memory. An FPGA configuration of a design describes what values the configuration memory should have to implement the design. Since the configuration memory consists of SRAMs, this memory can be overwritten at run time, and thus the functionality on an FPGA can be changed at run time. This is why FPGAs can be used for run-time reconfigurable implementations of applications and why they are a useful platform for dynamic circuit specialization.

Dynamic circuit specialisation strives to specialise a circuit, at run time, for the specific inputs at that time. This is a form of constant propagation in hardware. This specialisation results in a circuit that is both faster and smaller than the original circuit. Each specialisation takes time, both to generate the specialised configuration and to configure the FPGA. This means that in practice, dynamic circuit specialisation is only useful when some of the inputs of the circuit change less frequently than others. We will call these slowly changing inputs parameters from now on.

There has already been a lot of research on run-time reconfiguration on FPGAs. Most vendors even have their own tool flow for run-time configuration. The Xilinx Modular design flow is one example [9]. This design flow allows the user to use the FPGA hardware in a time-multiplexed way. If one wants to implement two or more applications that never need to be available at the same time, then this design flow can be used to implement both applications on the same FPGA area. This can be done for completely different applications or for different specialisations of the same circuit. At run time, when a new application is needed, the appropriate configuration is retrieved from memory and the FPGA will be (partially) reconfigured.

This design flow could be used for dynamic circuit specialisation, but only in some very specific cases. The problem with this approach is that it scales exponentially with respect to the number of parameter bits. A solution for this could be to generate the specialised circuit at run time. However running the full FPGA tool flow at run time introduces an overhead of seconds to hours, a much too large overhead for most run-time reconfigurable applications.

Parameterised configuration is not the only method or concept for dynamic circuit specialisation. The existing techniques, for both dynamic circuit specialisation and run-time reconfiguration in general, fall in two main categories based on how they approach placement and routing.

The first group avoids placement and routing at run time and so has trouble realising LUT-area gains. Reference [10] is an example of this. They bypass logic cells that become unnecessary after run-time constant propagation. This results in a circuit with better timing behaviour, but this does not decrease the number of LUTs the application needs. Our technique, parameterised configurations does show good area gains.

The second group implements some form of placement and routing at run time. This group of solutions does show LUT-area gains. The problem with this approach is that placement and routing stay NP-complete problems, even if they are simplified. Solving NP-complete problems at run time is very hard because of the limited time available. Our approach takes care of all NP-complete algorithms at compile time. Reference [11] shows work in developing a JIT hardware compiler, aimed at solving simplified place and route problems at run time. Reference [12] is a paper that also falls in this category. They simplify the routing problem by reducing the complexity of the routing architecture. The paper presents a global dynamic router for connecting reconfigurable modules. This global dynamic router allows the connections between modules to be dynamically changed, not the connections within these modules.

The parameterisable configuration concept strives to solve the problems with the existing techniques. A conventional FPGA configuration consists of LUT truth table bits and routing bits. A parameterised configuration is an FPGA configuration where some of the LUT truth table bits are expressed as Boolean functions instead of Boolean values. The Boolean functions are functions of the parameters we discussed earlier. If the parameters change, then the new configuration is generated by evaluating the Boolean functions, based on the current parameter values. By evaluating the Boolean functions a fully specified FPGA configuration can be generated very quickly. This generation only involves evaluating Boolean expressions, and can be done fast enough for use at run time. Actual measurements of this evaluation time will be discussed in Section 9.

A parameterised configuration consists of two parts. A template, the part of the configuration that has fixed Boolean values, and the Boolean functions, the part of the design that is dependent on the parameters. In [1, 13] the *TMAP* tool flow is presented. This tool flow is able to generate parameterised configurations automatically, based on VHDL. This tool flow is an adaptation of the conventional FPGA tool flow for the generation of configurations. Both tool flows are shown in Figure 1.

The conventional tool flow is shown in Figure 1(a). The synthesis tool converts a VHDL description of the design to a gate level circuit. The technology mapper maps this circuit on an LUT-circuit. This LUT-circuit is then given to the placer and the router to place the design on an actual FPGA.

There are several changes for the *TMAP* tool flow, as seen in Figure 1(b). First, the VHDL code is now annotated. These annotations show which input signals should be considered parameters. This annotated VHDL file is then given to an adapted technology mapper, the *TMAP* technology mapper. This mapper also maps the design to an LUT-circuit, but the truth tables of these LUTs can now be dependent on the parameter values, instead of being fixed in a conventional FPGA tool flow. The result is that logic that is only dependent on parameter values will now be expressed as Boolean functions. These Boolean functions determine the truth tables of some of the LUTs. This means the total number of LUTs will be smaller because some of the functionality, originally expressed in LUTs, will now be incorporated in the Boolean functions. This will lead to LUT-area gains. Both the placement and the routing is done with the same tools as the conventional FPGA tool flow.

It is clear that parameterised configurations can be used to implement dynamic circuit specialisation. As said before, the parameterised configuration consists of two parts, the template and the Boolean functions. The template contains all the LUTs that need to be placed and routed. The area needed to implement this template is smaller than using the original toolflow without parameters. All the logic that is only dependent on the parameters is now expressed in Boolean functions and does not require separate LUTs. However, a template in itself does not contain all the information necessary for a fully working implementation. Some of the truth tables are expressed in the Boolean functions. To get a working specialised implementation, these Boolean functions need to be evaluated, based on the actual parameter values.

The above means that all the NP-complete problems, like place and route, can be done at compile time, because they determine the template which does not change at run time. At run time we only need to evaluate Boolean functions, based on the current parameter values, to specialise the circuit. When a parameter changes, the Boolean functions need to be evaluated and the FPGA needs to be reconfigured according to the new results. This introduces an overhead.

A much more detailed overview of parameterised configurations is given in both [1, 13]. Important for us is that the input of the *TMAP* tool flow is annotated VHDL and its output is a working parameterised configuration. This output also includes the code that needs to be executed to evaluate the Boolean functions. The annotations added to the VHDL are very simple and their only role is to tell the tool which of the input signals are parameters. Any of the input signals of a design or parts of a design can be selected as parameters. It is still the job of the designer to chose the parameters, from that point on no user intervention is needed.

The reconfiguration platform, regardless of reconfiguration technique, always consists of two elements, the FPGA itself and a configuration manager. This configuration manager is responsible for evaluating the Boolean functions and supervising the reconfiguration of the FPGA. The configuration manager is generally a CPU. We will use a PowerPC on the FPGA but that is not the only option.

A different perspective on parameterisable configurations is given in the next section. There we will discuss how using *TMAP* can be seen as executing a hardware/software partitioning. This partitioning extends the hardware/software boundary, because run-time reconfiguration is used.

3. *TMAP* as HW/SW Partitioning

The *TMAP* tool flow is an extended hardware/software partitioning method. It is based on the distinction between the slowly changing inputs, the *parameters*, and swiftly changing inputs (regular inputs). Once we have selected the parameters, the tool (*TMAP*) generates a parameterised configuration of the design. This parameterised configuration can then be split up in a hardware and a software part. The hardware part corresponds to the bits in the parameterised configuration that have Boolean values (0 or 1). This will give us an incomplete FPGA configuration, the template. The software part then consists of Boolean expressions in the parameterised configuration that are dependent on the parameters. Evaluating these Boolean expressions will generate the values to complete the FPGA configuration, and is done in software by the configuration manager.

The similarities between *TMAP* and hardware/software partitioning are discussed in detail in [14]. However, it is important to realize that because run-time reconfiguration is used, the hardware/software boundary can be extended compared to traditional hardware/software partitioning.

3.1. The Hardware/Software Boundary. Using FPGAs and run-time reconfiguration the hardware/software boundary can be extended so a larger part of the design can be implemented in software. This is possible because run-time reconfiguration allows the use of specialized circuits. In a conventional hardware/software partitioning, the hardware part would have to be generic to accommodate all possible parameter values. Using FPGAs and run-time reconfiguration, the hardware part can be optimized for specific parameter values.

In a conventional approach to hardware/software partitioning without run-time reconfiguration, such as [15], we would select the slowly changing inputs of the design. Next we identify the functionalities that are only dependent on these parameters. This way the hardware/software boundary is identified. In the next step the boundary is replaced by

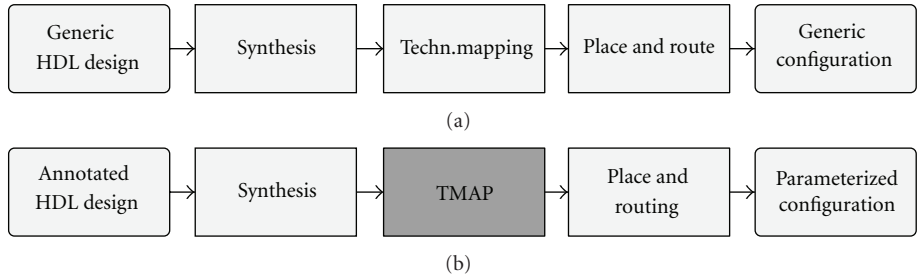

Figure 1: Conventional and *TMAP* FPGA tool flow.

registers and the functionalities that are only dependent on the parameters all get a software equivalent. The actual hardware then consists of the registers and the remainder of the design. The signal values on the boundary will be calculated by the software, and then written to those registers.

When using an approach that includes run-time reconfiguration, several things change. Using the parameterised configuration concept, and the *TMAP* tool flow, we can extend the hardware/software boundary by moving it to the configuration memory instead of adding registers to the design. In addition, because parameterised configurations are used, the hardware will be a specialized circuit that is optimized for specific parameter values. This is in contrast to the generic hardware design of the previous approach. As for the software, in this case it consists of Boolean functions that are generated automatically, based on the hardware functionality that is replaced Figures 2 and 3.

In the next sections of this paper we will discuss in greater detail what the effects are of applying *TMAP* and parameterisable configurations on actual applications. First, in Sections 4 and 5 we will discuss a series of AES implementations, including our own k_AES design. We will explain why using *TMAP* on our design has better gains, compared to other AES implementations. In Section 6 we will also discuss two other encryption algorithms. Finally, we will discuss our tmapped DNA aligner, the RTR aligner in Section 7. Because the specialisation overhead is important in this case, we will give a detailed overview and offer optimisations in Section 9.

4. Advanced Encryption Standard

The complete Advanced Encryption Standard (AES) is described in [2] by the NIST. This document describes the details and mathematical background involved with the different AESs. We will only discuss the hardware implementation of the algorithm, and therefore will only focus on the practical implications of AES.

In its most basic form, AES describes how to encode 128 bits of data, using an encoding scheme based on a specific key value. The Advanced Encryption Standard consists of three separate standards, whose main difference is the length of this key (128, 192, and 256 bits). The three standards are very similar and do not require significantly different hardware implementations. This is why we only discuss the 128-bit AES algorithm in this paper.

The 128-bit AES application can be split up in two parts: data encoding and key expansion. The data encoding explains how the data will be converted into encoded data. For this process we need several values that are key dependent, the round keys, which are generated by the key expansion.

(1) Data Encoding. In the AES algorithm the input data is encoded per 128-bit blocks. We split this data up in 16 bytes, and assign each byte a location in a 4-by-4 state matrix. The encoding part of the AES algorithm consists of a series of bytetransformations that are applied to the state. These transformations are combined in a specific order and grouped in a round, and each round has a round key input. This round key value is different for each round, and all these values are the result of the key expansion.

(2) Key Expansion. The round keys, necessary for the addRoundkey transformation in the different rounds, are derived from the input key. This key is expanded to generate the different round keys. The first round key is the input key, this key is used in the first `addRoundKey()` that is applied to the data input. From the second round key, the round key generation uses some similar transformations as the data encoding process. A more detailed description of the exact transformations used in both the key expansion and the data encoding can be found in [2].

4.1. k_AES. To explain how the *TMAP* tool flow works, we chose an application where the parameter is easy to identify. As described in Section 4, the AES application consists of two main parts: the data encoding and the key generation. Both parts are clearly separated and work, in general, on a different time scale. In most practical applications the key changes much slower than the input data. So, the key input signal is a good parameter choice, we expect a large part, if not all, of the key expansion will be moved to Boolean functions.

To show how parameterised configuration can significantly optimize a design very quickly by making it run-time reconfigurable, we wrote our own AES implementation, *k_AES*. In the next Section 5, this design will be compared with several other AES designs. *k_AES* is almost a direct reflection of the hardware described in the NIST document [2]. The design was deliberately kept fully parallel, and it was therefore very simple to write and test.

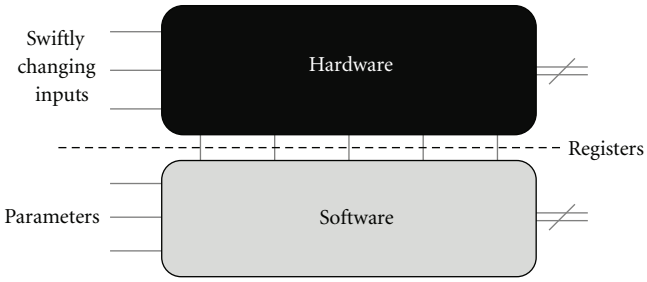

Figure 2: Conventional hardware/software partitioning.

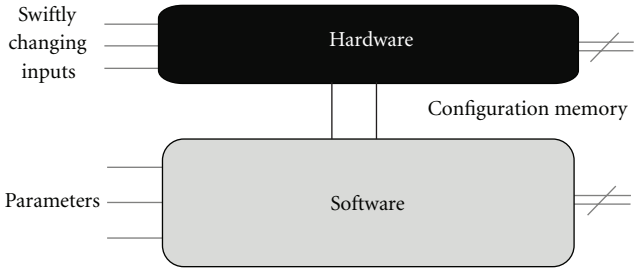

Figure 3: Hardware/software partitioning using *TMAP*.

The design is split up in two main parts, a chain of 10 rounds that implements the encoding process and a chain of 10 k_rounds that take care of the key expansion. The key expansion is combinatorial. The actual transformations are not clock dependent, all the parts are instantiated separately and are only used for calculating one transformation. All this means that we use a lot of area, but conversely encode the data very fast. The throughput is 128 bit of encoded data every clock cycle.

4.2. Making k_AES Run-Time Reconfigurable. Once we have designed our *k_AES* implementation, the next step is to use the *TMAP* tool flow on this design. Since the tool flow is automatic once we have annotated the VHDL code, the main decision designers have to make when using this technique is which signals to select as parameters. Some designs are clearly suited for this approach, like our AES encoder, *k_AES*, and other encryption algorithms that are key based. If a key is used to encode the data, in most cases this key changes very slowly compared to the data.

Once the parameter is selected the *TMAP* tool flow can be used as any other FPGA tool flow is used. It will generate a parameterised configuration in approximately the same amount of time a traditional FPGA tool flow needs to generate an FPGA configuration from a VHDL description. These kind of optimizations are a lot faster than manually optimizing a VHDL implementation by, for example, reducing parallelism in the key expansion. We will discuss the result of applying *TMAP* to *k_AES* in the next section, where we will also discuss several other AES implementations. Applying *TMAP* to these implementations yields much smaller gains. We will briefly discuss the reasons for this observation.

Table 1: The resource usage, in 4 LUTs, of the different AES designs.

	fpga	TMAP	Gain	Throughput (Gbps)
k_AES	45178	35843	20.3%	17.68
Crypto-PAn	37874	37750	0.3%	11.26
Avalon_AES	8448	8448	0%	1.31
AES128	5111	5071	0.7%	0.92
AES_core	5973	5973	0%	0.12

5. Comparison of AES Implementations

In this section we will discuss the results of applying *TMAP* to different AES implementations. Quartus was used as the synthesis tool. As the technology mapper when *TMAP* was not used, we chose the mapper available in ABC [16], called *fpga*. Also, to provide a good basis to compare the different designs, we will only compare LUT-only implementations. This was decided because it is entirely dependent on external factors if you would rather use more BRAMs and less LUTs or the other way around. The DNA aligner in the next section is a good example of *TMAP* influencing the resource tradeoff. This tradeoff will be discussed in much more detail in Section 8.3.

5.1. Area Optimization through Parameterised Configurations. In Section 2 we discussed the concept of parameterised configuration and the *TMAP* tool flow. The result of the *TMAP* tool flow can be seen in Table 1.

The first important result in Table 1 is that, as suggested in Section 4, *TMAP* succeeds in optimizing our original *k_AES* design significantly. The version of *k_AES* mapped by *TMAP* is 20.3% smaller than the design mapped by the original mapper. Basically, the *TMAP* tool flow has removed all the parts of the design that are solely dependent on the parameter, in this case the key input, and has converted those parts to Boolean functions. For the *k_AES* design this means that almost the full key expansion has been moved to software.

However, we do see a difference in clock speeds between both designs. The original *k_AES* has a maximum clock speed of 146.63 Mhz while the tmapped version only has a clock speed of 138.17 Mhz. This is still significantly better than the other AES implementations below, but does mean a 5.7% clock speed decrease. This means we offer up a 5.7% of throughput to reduce the area by more than 20%.

One comment is needed here: the usage of dynamic circuit specialization will introduce a specialization overhead. Each time the key changes, the new FPGA configuration has to be generated and the FPGA itself has to be reconfigured, the total of both types of overhead is called the specialization overhead. The impact of this overhead will be discussed in Section 5.3.

5.2. Comparison to Opencore AES Implementations. To compare our design to other designs, we looked at the publicly available AES implementations found on the opencores.org website. For applying *TMAP* the designs need to be written

in VHDL. We only looked at the "stable" cores that were finished. We found 5 AES implementations that fit these criteria. We had to discarded one implementation because its results with the *fpga* technology mapper generated errors. This leaves us with four AES implementations, the *Avalon_AES* [4], the *Crypto-PAn* [3], the *AES128* [5], and the *AES_core* [6] implementation. The original goals of these applications are less important, and we will only discuss the AES implementations within these applications.

In the first application, "*Crypto-PAn*", an AES-encoding module implements the rounds in a similar way as in our AES implementation, *k_AES*. The main difference is in the key expansion, where the *Crypto-PAn* AES implementation uses a much more complex and more sequential design, involving a state machine. This contrasts to the more parallel *k_AES* implementation, where the key expansion is spread out into 10 k_rounds. For this design, the throughput is the same as for the *k_AES* design, 128 encoded bits every clock cycle. This design can be clocked at 88.043 Mhz.

As Table 1 indicates, we see that, the unoptimized *k_AES* implementation is quite a bit larger than the *Crypto-PAn* design, 45178 LUTs compared to 37874 LUTs. The optimized version of k_AES, however, shows a 5% area gain, compared to the manually optimized *Crypto-PAn* design. From Table 1 it is also clear that both the throughput and the size of the design are worse than the optimized *k_AES* implementation. When comparing *k_AES* and *Crypto-PAn*, the design time and complexity should also be taken into account. The *k_AES* key expansion is significantly more simple and easier to implement and test. As the results show, using *TMAP*, *k_AES* was optimized automatically to the point where it is smaller than more complex and time-consuming implementations.

The second application, "*Avalon_AES*," is even more sequential. Not only in the key-expansion, but also in the encoding part of the AES algorithm. This implementation only consists of one round that is used sequentially to run the full AES algorithm. This design is significantly more complex than the *k_AES*. It also has a much lower throughput. This design needs 10 clock cycles to generate 128 bits of encoded data. The clock speed of this design is 102.76 Mhz. There are significant differences between k_AES and Avalaon_AES. It is very hard to directly compare both designs, because they are both Pareto efficient. Which one is better is decided by external factors.

The third and fourth application, *AES128* and *AES_core*, are both similar to the *Avalon_AES* implementation of the AES algorithm. The *AES128* is a fully sequential AES implementation. It outputs encoded 128 bits of data every 13 clock cycles. This design clocks at 94.20 Mhz. The *AES_core* design is even more sequential, as it works on a byte level instead of the full 128-bit data length. The design size is small, but encrypting 128 bits of data takes more clock cycles. The clock speed of the *AES_core* design is 156.6Mhz. This is the fastest design, but also the one which will take the largest number of clock cycles to encode 128 bits of data.

If we look at the "*TMAP*" column of Table 1, we see that the gains to the other designs for using *TMAP* are a lot less or even nonexistent. The reason is closely linked to how parameterised configurations work. Since we chose the key input as a parameter, all the signals that are dependent on the key input, and not on any other signals, are removed from the design. In the case of a fully parallelised design, like *k_AES*, this results in almost the complete key expansion being moved to software as Boolean functions. In more sequential designs, however, the key expansion is not only dependent on the key input, but also on swiftly changing signals. In these designs, internal timing and hardware reuse make sure that the signals change at a higher rate than the key input. This means a smaller part of the design will be only dependent on the parameters. In essence, a (much) smaller part of the key expansion is moved to software, because most of the hardware used for the key expansion is reused, and thus no longer only dependent on the key input. This is clearly the case in the opencore designs.

These results were attained by comparing 128-bit key AES implementations.

5.3. Specialization Overhead. In the case of AES and similar scenarios the impact of the specialisation overhead is fully dependent on external factors and individual-use cases. In most user-initiated parameter changes an overhead of milliseconds, which is very attainable using *TMAP*, will be unnoticeable to the human user. In this case, dynamic circuit specialization can be used without a large or noticeable impact on the user experience.

In other cases, any delay will be too large. For example, one can easily image a encryption scheme in which the AES key changes for every new 128 bit of input data. In this case the specialization overhead will have a huge impact on the actual operating speed of the design because it will need to be reconfigured very frequently. In most of those cases a delay in the order of milliseconds will never be acceptable.

In Section 9 we will discuss the specialisation overhead further, for the case of the DNA aligner. We will also discuss optimizations to reduce this overhead. But first, in the next section, we will show our results with other key-based encryption algorithms.

6. Other Encryption Algorithms

To check whether our results are transferable to other key-based encoding algorithms, we looked at two other encoding algorithms, "*TripleDES*" [7] and "*RC6*" [8].

The TripleDES algorithm is based on a Data Encryption Standard (DES) encoding/decoding scheme. The data is first encoded with a specific key, then decoded with another key, and finally encoded again with a third key. These separate encoding/decoding steps are done using the DES algorithm. We select the three input keys as parameters for similar reasons as in the AES application. Since a suitable opencore implementation was available for TripleDES [17], we annotated their VHDL description and ran the design through the *TMAP* tool flow without any further adaptations. The result is a 28.7% area gain, from 3584 to 2552 LUTs.

The second algorithm is the RC6 algorithm, it was designed for the AES competition and was one of the finalists. RC6 has a block size of 128 bits and supports key

sizes of 128, 192, and 256 bits. An opencores implementation of this algorithm was also available [18]. Here, we also chose the key as a parameter. Using *TMAP* on this implementation resulted in a 72.7% LUT-area gain, from 4635 to 1265 LUTS.

7. DNA Alignment

DNA alignment is used here as an example of a string matching algorithm, so not a lot of detail on the biological background will be given. The actual algorithm is the Smith-Waterman algorithm, proposed in [19]. The aim of this algorithm is to find the region within two DNA sequences, sequence A and B, where they share the most similarities.

This is done by filling in a score matrix (F). Each element in the score matrix corresponds to one character in sequence A and one character in sequence B. The rows are associated with sequence A, which we will call the vertical sequence from now on. The columns are associated with sequence B, which we will call the horizontal sequence from now on. The element i, j of the score matrix then corresponds to the score for comparing A_i and B_j. The score matrix is filled by solving (1) for each element. Once the matrix is completely filled in, the regions with the most similarities are found by starting from the maximum value in the score matrix and using a trace-back algorithm to find the start of this region,

$$F_{i,j} = \mathrm{Max}\left(F^D_{i,j}, F^V_{i,j}, F^H_{i,j}\right), \tag{1}$$

where

$$\begin{aligned} F^D_{i,j} &= F_{i-1,j-1} + S(A(i), B(j)), \\ F^V_{i,j} &= F_{i-1,j} + \omega, \\ F^H_{i,j} &= F_{i,j-1} + \omega. \end{aligned} \tag{2}$$

F^D takes the matrix element diagonally above and adds a value S. This value S is positive if $A(i)$ and $B(j)$ are the same and negative in case of a mismatch. The values for all possible pairs $A(i)$ and $B(j)$ are described in a substitution matrix. The values in this substitution matrix are based on biological considerations, so we will not go into further detail here. The value taken from this matrix is called the substitution cost.

F^V takes the matrix element directly above and adds a gap penalty, ω. F^H does the same but with the element to the left. The maximum of these three values $(F^D_{i,j}, F^V_{i,j}, F^H_{i,j})$ is then selected and assigned to $F_{i,j}$.

8. RTR Aligner

To get a run-time reconfigurable DNA aligner, we started with an already existing DNA aligner, that was developed for a project (the FlexWare project) funded by the IWT in Flanders and therefore called the Flexware aligner [20]. We will first discuss the structure of this Flexware aligner briefly, then we will discuss the changes and decisions made to turn it into an RTR aligner.

8.1. Flexware Aligner. The Flexware aligner, like most DNA aligners, is based on the observation that the data dependencies in the Smith-Waterman algorithm are west, northwest, and north, as you can see from Figure 4. This means that the algorithm can be HW accelerated by using a systolic array, as shown in Figure 5. Vertical sequence (A) is streamed through the array and each PE calculates an element of the score matrix each clock tick. Once the full vertical sequence has been streamed through the array, the maximum score and its location are known.

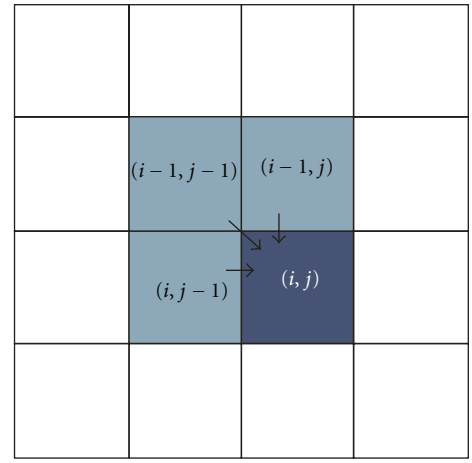

Figure 4: The data dependencies in the Smith-Waterman algorithm [20].

If there are not enough PEs to compare the sequences completely, memory is used to store temporary data. Then, the horizontal sequence is streamed through the array multiple times, each time with different column characters. In Figure 6, you can see the progress of the PEs of this systolic array through the score matrix. Each clock tick a PE calculates the score for its row and column characters.

In general the DNA sequences that are compared are longer (2^{13}) than the number of PEs that fit on an FPGA, at most hundreds on very large FPGAs. This means that more PEs will always speed up the execution.

8.2. RTR Aligner. The parameter selection in the case of the DNA aligner is less straightforward than in the case of AES. However, looking at the inputs of the systolic array we can see that, for each PE, the column character stays constant while the whole vertical sequence is streamed through the systolic array. On the design level the column sequence was chosen as a parameter, for an individual PE this translates to the column character being chosen as a parameter.

Before applying *TMAP* to the Flexware aligner, some changes to the control flow were made so the aligner could be started and stopped without any problems. This is necessary because between configurations the aligner will have to stop working and we need to be able to restart it when the reconfiguration is done. The result of these changes and applying the *TMAP* tool flow is the RTR aligner. We will compare this RTR aligner to the original Flexware aligner in detail below.

In most practical cases, the horizontal sequence will always be longer than the amount of PEs. This means that

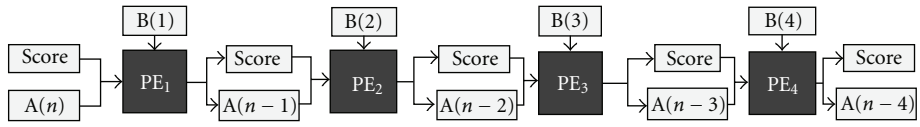

Figure 5: A systolic array.

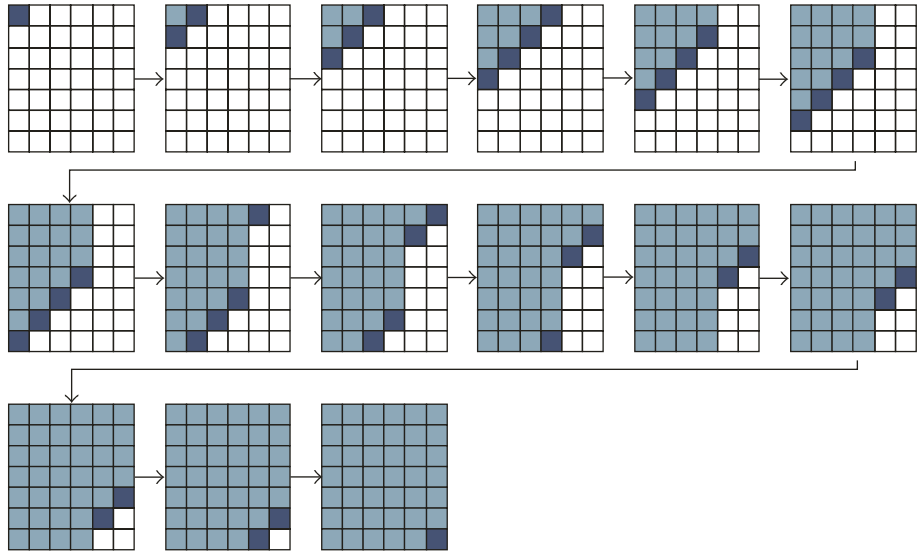

Figure 6: Progress of the PEs in the score matrix during the Smith-Waterman algorithm [20].

the column character of the PEs will change several times during operation. The PEs will need to be reconfigured every time they are started on a new column. This introduces an overhead that will be discussed in Section 9.

From the previous details on the Flexware aligner, and from actual measurements in Figure 8, it is clear that extra PEs are an important factor in the system's overall efficiency. In the original Flexware aligner each PE required its own BlockRAM. This is not the case for the RTR aligner where the number of PEs has no influence on the BRAM usage. In FPGAs with a limited number of BRAMs this can make a huge difference in resource usage. For example, Table 2 shows that the resource usage of the RTR aligner on a Spartan device is a lot better than for the Flexware aligner. It should be noted that only comparing the increase in the number of PEs is not a good comparison. The RTR aligner will be less efficient because it will have to be reconfigured several times. This will be discussed in a lot more detail in the next section, where we will show experimental results based on an actual FPGA implementation.

8.3. Experimental Results. In this section we will compare the operation of the Flexware aligner and the RTR aligner. The next section will discuss the specialization overhead and optimizations based on the data in this section.

As discussed in Section 2, the reconfiguration platform consists of two elements, the FPGA itself and a configuration manager. The configuration manager is generally a CPU.

Table 2: The resource usage in the Flexware and RTR aligner on a Spartan XC3S1600E-4FG484.

	Flexware	RTR	Max. avail.
Slices	3766 (25%)	14708 (99%)	14752
Slice flip flops	2635 (8%)	9515 (32%)	29504
4-input LUTs	6967 (23%)	27382 (92%)	29504
BRAM	36 (100%)	19 (52%)	36
PEs	14	50	—

Our test platform is shown in Figure 7. We used the Virtex II Pro, this FPGA has a PowerPC. The PowerPC is used as the configuration manager. Additionally, the Virtex II Pro has a readily available run-time reconfiguration interface, the HWICAP. We will use this platform to test our design, all the data in this section was collected on the test platform. In all our tests, the OPB/PLB-bus was clocked at 100 Mhz.

8.3.1. Flexware Aligner versus RTR Aligner. In Figure 8 we display the influence of the amount of PEs on the execution time of both the Flexware aligner and the RTR aligner. For the RTR aligner this is the execution time *without the specialization overhead*. As you can see there is a difference between both aligners, but at most it is 18 μs, this is caused by the control flow changes that were made to the RTR aligner to allow for dynamic circuit specialization. It is also clear

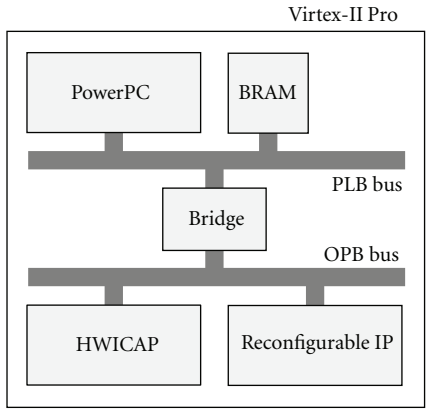

Figure 7: Virtex II Pro run-time reconfigurable platform.

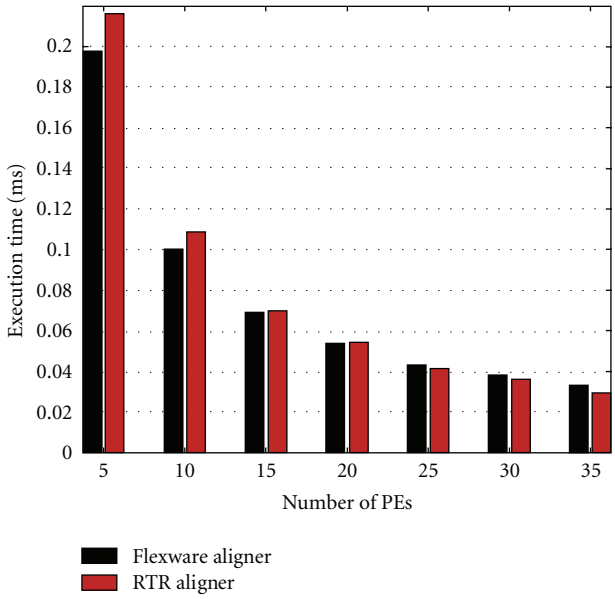

Figure 8: Execution time (ms) for both aligners, without reconfiguration overhead, based on measurements on the test platform when comparing a 188 to a 258 DNA sequence.

that the relative gain of additional PEs decreases as the total number of PEs in the system increases, for both designs.

For the designs described above the maximum clock speed was measured too. For 4 PEs the RTR aligner clocks at 62 Mhz and the Flexware aligner at 59 Mhz. However, this difference reduces steadily as the area of both designs increases with additional PEs. For designs with 20 or more PEs this difference is less than 1 Mhz. For both designs the clock speed gradually settles down to around 50 Mhz when designs span the complete Virtex II Pro FPGA. There is no clear difference between the two, certainly not for the largest designs.

Now we look at the RTR aligner with the specialisation overhead included. In this case the total execution time is around 3.7 seconds, for any number of PEs, when comparing a 188 to a 258 DNA sequence. It is very clear that this specialization overhead imposes too large a cost on the

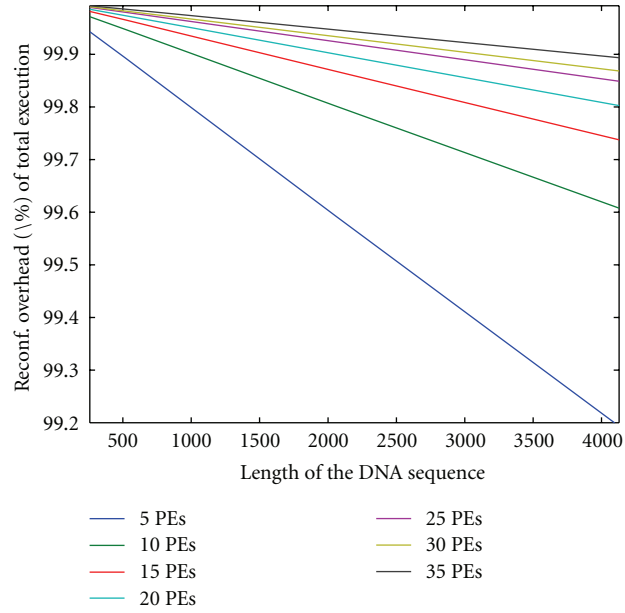

Figure 9: Fraction of the total time spent on reconfiguration, with increasing DNA sequence length, based on measurements on the test platform.

design, in this case. Using HWICAP reconfiguration makes the system very inefficient compared to the Flexware aligner. In the following optimizations we will reduce this overhead drastically and eventually make the design more efficient than the Flexware aligner. The reason there is no reduction of the total execution time with additional PEs is because the added reconfiguration time of the extra PEs offsets the execution time increase. This will also change with our optimizations.

Figure 9 shows that the fraction of the time spent on actual execution increases with longer DNA sequences, regardless of the number of PEs.

8.3.2. Increasing the DNA Sequence Length. The measurements for Figure 8 were taken comparing sequences of 188 and 258 characters. Those are small sequences in DNA alignment, where sequences in the order of 2^{13} are a lot more common. Increasing the sequence length will impact both designs in a different way.

The Flexware aligner's overall execution time is proportional with the product of the sequence lengths. If one sequence is doubled, the execution time also doubles. If both sequences double the execution time is quadrupled.

Before we can discuss the impact of increasing the sequence length on the RTR aligner, we need to make clear which sequence is chosen as a parameter. Because of the specialization overhead, minimizing the number of reconfigurations is essential. This means choosing the shortlinebreak est sequence as a parameter is clearly the best choice, this can greatly reduce the number of times the PEs need to be reconfigured.

With the shortest sequence as the parameter, increasing the other sequence length will work to the advantage of

the RTR aligner. The number of reconfigurations will stay the same but the time spent on execution will increase, as with the Flexware aligner. This means the relative cost of reconfiguring will decrease.

Another way to see this is that if the nonparameter sequence length increases, a configuration will be used longer before it is changed. From this we can conclude that for the RTR aligner to be working efficiently, the nonparameter sequence needs to be as long as possible. We will go into further detail on this in the next section.

9. Specialisation Overhead

A fundamental property of dynamic circuit specialization is the overhead incurred by this dynamic specialization. There are several possible scenarios, one was discussed with the AES implementation in Section 5.3. In AES the parameter change is decided externally, by a user or a governing program, because its parameter, the key, is an input of the global design.

In the RTR aligner there is a subtle difference. There is also an external parameter decision, a program or user decides which two sequences need to be compared. Internally, however, the application itself will govern the actual parameter change at the input of the PEs. Each time a PE is started on a new column, it will need to be reconfigured.

In this section we will discuss the RTR aligner specialisation overhead in more detail. In the last parts of this section we will also discuss optimizations we have implemented to reduce the specialization overhead and make the RTR aligner more efficient than the Flexware aligner.

In the case of the RTR aligner the specialization overhead has a huge influence on the overall efficiency of the design. This is because the timing of the specialization is a part of the systems normal operating environment. Each comparison of two DNA sequences will entail a certain number of reconfigurations. This number is dependent on the length of the sequences and the number of processing elements. To be clear, each time the last PE finishes its column, all the PEs are reconfigured and a start signal is given to restart the calculations. The next time the last PE has finished its column, a new reconfiguration is done. This goes on until every element of the score matrix has been calculated. The average specialization overhead of one such reconfiguration, which is reconfiguring each PE one time, is displayed in Figure 10. It is clear that if more PEs need to be reconfigured, the specialization takes longer. This increase is invariable of sequence length. Figure 10 also shows that sometimes a larger design will have a smaller specialization overhead. We will discuss this later in the subsection about the reconfiguration overhead.

There are a lot of ways to minimize this specialization overhead, several of the more interesting ones will be applied to the RTR aligner in order to make it more efficient than the Flexware aligner for the case shown in Table 2.

Before we look at optimizing the specialization overhead, we must first determine in which cases it will be used. The RTR aligner uses no BRAMs for its PE implementation, the Flexware aligner uses one BRAM for every PE. Most high-end

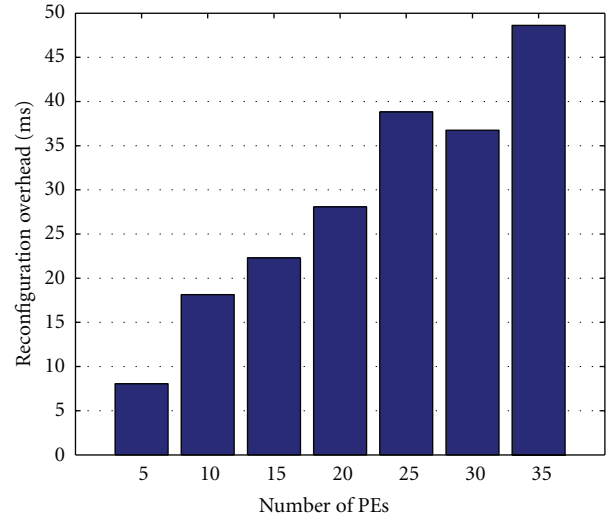

Figure 10: Specialization overhead (ms) needed to reconfigure the PEs of the RTR aligner one time, based on measurements in the test platform.

FPGAs, such as the Xilinx Virtex-line, have enough BRAMs. They generally run out of LUTs before all BRAMs are used by the Flexware aligner. Cheaper FPGAs, however, such as the Xilinx Spartan discussed below quickly run out of BRAMs before all the available LUTs are used, making these FPGAs much less optimal for the Flexware aligner. An example of this is shown in Table 2.

The reason why Spartan implementations are interesting is the price difference between Spartan and Virtex FPGAs. This difference is easily a factor of 10 for equally sized FPGAs.

In [21] we calculated the maximum allowed specialization overhead for the RTR aligner to be as efficient as the Flexware aligner. In the case of the Spartan displayed in Table 2, this resulted in the following upper bound for the specialization overhead of 50 PEs, 455 μ seconds. This is an upper bound for the time needed to reconfigure all the PEs one time. From the measurements of the specialization overhead displayed in Figure 10 it is clear that we already surpass this upper bound for 5 PEs, let alone 50. To improve the efficiency and meet this upper bound, we will discuss several optimizations to the RTR aligner. The SRL optimization discussed below is interesting for any run-rime reconfigurable application that needs to be reconfigured as fast as possible.

Sadly, we have no Spartan XC3S1600E-4FG484 board to test the final design. We will apply the optimisations to the design implemented on the test platform. Based on these results we will discuss the effect of the optimisations on the application. The biggest disadvantage is that the Virtex II Pro does not have enough area to implement 50 PEs. So, in the following we will always discuss the results on the test platform first. Next, we will discuss what the effect of these optimisations is on the use case with the Spartan FPGA.

The first thing we have to realize is that the specialization overhead can be split up in two parts, the evaluation time and the reconfiguration time. The evaluation time is the time

needed for evaluating Boolean functions of the parameters, as described in Section 2. The reconfiguration time is the time needed to write these values to the configuration memory of the FPGA.

From Table 3, we can see that the evaluation time takes about 0.24% of the total specialization overhead and the actual reconfiguration time takes up 99.76% of the total specialization overhead. Also, these numbers are fairly constant when the number of PEs increases.

We can decrease both of these overheads in different ways. Because the reconfiguration time is by far the largest factor in the overhead we will optimize this reconfiguration time first.

9.1. Reconfiguration Time. A large part of the reconfiguration time is introduced by the HWICAP. The way it handles the reconfiguration is not always efficient [22]. It changes the truth tables in three stage. First, it needs to read a full frame from the configuration memory. Then it changes the truth table values. And last, it writes back the full frame. In addition, it writes a padding frame to flush the reconfiguration pipeline [23]. In the Virtex II Pro, each frame contains 320 LUTs with a small header. One can see that changing a single LUT truth table introduces a large overhead. Furthermore, the overhead is dependent on the placement of our design. In a worst case scenario, every LUT of a PE is in a different column. Each PE has 18 LUTs that need to be reconfigured. So, for each PE 18 frames have to be read and 36 have to be written. Figure 10 shows this dependency. If a larger design is placed more efficiently then this can lead to a significant reduction in the total reconfiguration time.

The HWICAP is not the only way to access the configuration memory of the FPGA at run time. In [24] the reconfiguration time is reduced using shift register LUTs (SRLs). An SRL is an LUT whose truth table bits are also arranged as a shift register. This shift register functions like any shift register, with a shift_in, a shift_out, and a clock. Thus, the truth table of an SRL can be changed by shifting in new values through the shift_in input. By connecting the shift_out output of one SRL to the shift_in input of another SRL, a long SRL chain can be formed. This long SRL chain can then be used to change the truth tables of all the LUTs that are part of it.

Such a long SRL chain can then be used to reconfigure the FPGA very quickly, by shifting the new configuration into the truth tables of the connected LUTs. This way the reconfiguration time is reduced to only hundreds or thou-\linebreak sands of clock cycles, instead of hundreds of thousands as with the HWICAP.

The SRL chain(s) can be added on top of the already existing design. The parameterised configurations tool flow determines which truth tables need to be changed. These LUTs are then marked and arranged in an SRL chain that is completely independent of the design. The chain is connected at the input with the configuration manager, it has no output and is only used to change the truth tables of the LUTs. Reference [24] also shows that the impact on the clock frequency of the design of adding these shift register LUT chain(s) is minimal. Important to note is that not every FPGA has this SRL functionality. The Spartan 3 and Virtex II Pro LUTs both have this capability.

Table 3: The specialization overhead, split up in reconfiguration time and evaluation time.

PEs	Reconfiguration time (ms)	Evaluation time (ms)
5	7.9 (99.76%)	0.019 (0.24%)
10	18.0 (99.76%)	0.044 (0.24%)
15	22.1 (99.74%)	0.057 (0.26%)
20	27.8 (99.76%)	0.069 (0.24%)
25	38.5 (99.76%)	0.095 (0.24%)
30	36.4 (99.74%)	0.095 (0.26%)
35	48.2 (99.76%)	0.120 (0.24%)

Table 4: The reconfiguration time, using shift register LUT reconfiguration in the RTR aligner.

PEs	Clock cycles	Theoretical (μs)	Measured (μs)
5	1440	27.91	29.15
10	2880	56.78	59.93
15	4320	84.18	88.93
20	5760	112.31	118.94
25	7200	142.68	151.13
30	8640	172.04	182.23
35	10080	199.83	211.93

The reconfiguration time using SRL reconfiguration can be seen in Table 4. Compared to the reconfiguration overhead using the HWICAP, as seen in both Figure 10 and Table 3, this is a huge decrease. The SRL chain is clocked at the same speed as the design.

The third column shows the theoretical reconfiguration time, assuming one long SRL. We know that we need to reconfigure 18 LUTs in each PE and that reconfiguring 1 bit takes one clock cycle. Based on these numbers we can calculate for each RTR implementation how long the reconfiguration using SRLs should take. In the next column we show our measured results on the test platform. In each instance we see that the measured reconfiguration time is 5% to 6% larger than the expected value. This is because of inaccuracies in our measurements. The reconfiguration time is completely dependent on the SRLs, which are completely deterministic. However, this extra overhead has no impact on our conclusions. The reconfiguration time using SRLs is two orders of magnitude smaller than the run-time reconfiguration overhead using the HWICAP.

In the use case shown in Table 2 we have 50 PEs at a clock speed of 46.25 Mhz. The theoretical reconfiguration time is then 313.04 μs. The upper bound from [21] is 455 μs. We are well below this upper bound, even if we assume that the 6% overhead is not related to inaccuracy in the way we measured. In that case the reconfiguration time is 331.82 μs. Of course, this does not necessarily mean that the total specialisation overhead is below the upper bound. For this to be the case we need to look at the evaluation time too.

9.2. Evaluation Time. The evaluation of the Boolean functions in our run-time reconfigurable platform is done by the PowerPC, shown in Figure 7. However, there are two good reasons to change this in the design on the Spartan. The first reason is that the Spartan has no PowerPC, so we would need to evaluate the Boolean functions on a Microblaze or a custom processor. A second reason to change this is that, in the case of the RTR aligner, the evaluation does not involve any Boolean evaluation. The new truth tables are directly dependent on the parameters, no Boolean expressions are necessary. Because of both these reasons, we propose to use a dedicated hardware block that will function as the configuration manager.

This dedicated hardware block takes a character of the parameter DNA sequence as the input and outputs the bit values that need to be shifted into the SRLs one clock tick at a time. This means, once we have started the dedicated hardware, we can start shifting in the results into the SRL the next clock cycle. The reconfiguration and evaluation can happen at the same time. This will greatly reduce the total specialization overhead. The extra time overhead introduced by the evaluation is then only one clock cycle, compared to the overhead from reconfiguring using SRLs.

Of course the size of this dedicated hardware block will influence how many PEs we can fit on the FPGA. This block uses a very small amount of LUTs (49) but does use one BRAM. In return for adding one BRAM, our total specialization overhead is reduced to the values seen in Table 4, with one additional clock cycle for starting the dedicated hardware block. This is a huge decrease from the specialization overhead shown in Figure 10. We can clearly see the advantages of these optimizations.

An advantage in addition of using the SRLs combined with the dedicated hardware is that we can further reduce our specialization overhead by using two or more SRL chains and an equal number of dedicated hardware blocks. In this case one SRL chain will be split up in two or more SRL-chains that are half as long, this will then reduce the reconfiguration time by half.

Finally, we look back at the upper bound, 455 μs proposed in [21], for the RTR aligner presented in Table 2. Since the Spartan FPGA has SRL capabilities, we can make use of the SRL reconfiguration. Additionally, because we still have unused BRAMs in our RTR aligner, we can use the dedicated hardware block. We do, however, have to remove one PE to make room for the dedicated hardware block. If we lower the number of PEs of the RTR aligner by one, the new upper bound becomes 442.43 μs. For 49 PEs and using both optimizations, our specialization overhead is reduced to 305.14 μs, well below our new upper bound. If we again take an extra 6% of reconfiguration time into account this number becomes 323.45 μs, still below the upper bound.

We can calculate, based on [21] that the RTR aligner will finish its execution earlier than the Flexware aligner on the Spartan XC3S1600E-4FG484, provided the nonparameter sequence is long enough. If we assume the nonparameter sequence is 2^{13} characters, then the RTR aligner will be between 1.15 and 1.29 times more efficient than the Flexware aligner, depending on the length of the parameter sequence.

The RTR aligner is more efficient than the Flexware aligner as long as the nonparameter sequence is longer than 6046 characters. We can decrease this number and increase the overall efficiency by using two or more SRL chains and dedicated hardware blocks. For example, using two SRL chains, and two dedicated hardware blocks, increases the maximum efficiency of the RTR over the Flexware aligner to 1.90. The nonparameter sequence lower bound for efficiency is then lowered to 2824. This trend holds for larger numbers of SRL chains.

Since one PE in the RTR aligner has a size of 369 LUTs, we can fit 7 dedicated hardware blocks in the area used for one PE. This means we can use 7 SRL chains. This improves the maximum efficiency of the RTR aligner to 2.80-times the Flexware aligner for nonparameter sequences longer than 758 characters.

10. Conclusions

In this paper we have shown that parameterised configurations as a method for dynamic circuit specialization and the corresponding *TMAP* tool flow can be used to improve key-based encryption and DNA alignment. In the case of encryption we show an area gain of 20.6% in the case of AES encryption, a gain of 29.3% for a TripleDES implementation, and a 72.7% gain for a RC6 implementation. As an example of a string matching algorithm we chose an existing DNA aligner, which we adapted to the RTR aligner. This RTR aligner runs more efficiently on cheaper FPGAs than the original design. We also showed methods to greatly reduce the specialization overhead introduced by dynamic circuit specialization, and discussed their effects in detail for the RTR aligner. They allow the RTR aligner to be up to 2.80-times more efficient that the Flexware aligner for nonparameter sequences longer than 758 characters.

Acknowledgment

This paper was funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders.

References

[1] K. Bruneel and D. Stroobandt, "Automatic generation of run-time parameterizable configurations," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 361–366, September 2008.

[2] N. I. of Standards and Technology, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," November 2001.

[3] "Opencores.org, the Cryptopan project," http://www.opencores.org/project,cryptopan_core.

[4] "Opencores.org, the Avalon AES project," http://www.opencores.org/project,avs_aes.

[5] "Opencores.org, the AES128 project," http://opencores.org/project,aes_crypto_core.

[6] "Opencores.org, the AES core project," http://opencores.org/project,aes_128_192_256.
[7] D. Coppersmith, D. B. Johnson, and S. M. Matyas, "A proposed mode for triple-DES encryption," *IBM Journal of Research and Development*, vol. 40, no. 2, pp. 253–261, 1996.
[8] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6 block cipher," in *Proceedings of the 1st Advanced Encryption Standard (AES) Conference*, p. 16, 1998.
[9] Xilinx Inc., *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*, Xilinx Inc., 2002.
[10] N. McKay and S. Singh, "Dynamic specialisation of XC6200 FPGAs by parial evaluation," in *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, from FPGAs to Computing Paradigm*, pp. 298–307, Springer, London, UK, 1998.
[11] E. Bergeron, M. Feeley, and J. P. David, "Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC '08/ETAPS '08)*, pp. 178–192, Springer, Berlin, Germany, 2008.
[12] J. Surís, C. Patterson, and P. Athanas, "An efficient run-time router for connecting modules in FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 125–130, September 2008.
[13] K. Bruneel and D. Stroobandt, "Reconfigurability-aware structural mapping for LUT-based FPGAs," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 223–228, December 2008.
[14] T. Davidson, K. Bruneel, and D. Stroobandt, "Run-time reconfiguration for automatic hardware/software partitioning," in *Proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 424–429, IEEE Computer Society, Los Alamitos, Calif, USA, 2010.
[15] A. Kalavade and P. A. Subrahmanyam, "Hardware/software partitioning for multifunction systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 9, pp. 819–837, 1998.
[16] "ABC: A System for Sequential Synthesis and Verification," Berkeley Logic Synthesis and Verification Group, http://www.eecs.berkeley.edu/~alanmi/abc/.
[17] N. Seewald and K. Sissell, "Braskem triples income, postpones venezuela projects," *Chemical Week*, vol. 171, no. 22, 2009.
[18] "Opencores.org, the RC6 cryptocraphy project," http://opencores.org/project,cryptography.
[19] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.
[20] F. Project, "Deliverable 2.6: Report on the process of the implementation of the Smith-Waterman algorithm on the FPGA architecture," November 2007.
[21] T. Davidson, K. Bruneel, H. Devos, and D. Stroobandt, "Applying parameterizable dynamic configurations to sequence alignment," in *Proceedings of International Conference on Parallel Computing (ParCo '10)*, p. 8, Lyon, France, 2010.
[22] K. Bruneel, F. M. M. A. Abouelella, and D. Stroobandt, "Automatically mapping applications to a self-reconfiguring platform," in *Proceedings of Design, Automation and Test in Europe*, K. Preas, Ed., vol. 4, pp. 964–969, Nice, France, 2009.
[23] O. Blodget, P. James-Roxby, E. Keller, S. Mcmillan, and P. Sundararajan, "A self-reconfiguring platform," in *Proceedings of Field Programmable Logic and Applications*, pp. 565–574, 2003.
[24] B. Al Farisi, K. Bruneel, H. Devos, and D. Stroobandt, "Automatic tool flow for shift-register-LUT reconfiguration: making run-time reconfiguration fast and easy (abstract only)," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '10)*, pp. 287–287, ACM, New York, NY, USA, 2010.