Run-time reconfiguration for automatic hardware/software partitioning

Tom Davidson
ELIS department, Ghent University
Sint-pietersnieuwstraat, 41
9000, Ghent, Belgium
Email: tom.davidson@ugent.be

Karel Bruneel
ELIS department, Ghent University
Sint-pietersnieuwstraat, 41
9000, Ghent, Belgium
Email: karel.bruneel@ugent.be

Dirk Stroobandt
ELIS department, Ghent University
Sint-pietersnieuwstraat, 41
9000, Ghent, Belgium
Email: dirk.stroobandt@ugent.be

***Abstract*—Parameterisable configurations allow very fast run-time reconfiguration in FPGAs. The main advantage of this new concept is the automated tool flow that converts a hardware design into a more resource-efficient run-time reconfigurable design without a large design effort. In this paper, we show that the automated tool flow for run-time reconfiguration can be used to easily optimize a full hardware implementation for area by converting it automatically to a hardware/software implementation. This tool flow can partition the design in a very short time and, at the same time, result in significant area gains. The usage of run time reconfiguration allows us to extend the hardware/software boundary so more functionality can be moved to software.**

We will explain the core principles behind the run-time reconfiguration technique using the AES encoder as an example. For the AES encoder the manual hardware/software partitioning is clear. This manual partitioning will serve as a comparison to the automated partitioning that uses parameterisable configurations. Several possible AES encoder implementations are compared. Our automatically partitioned AES design shows a 20.6 % area gain compared to an unoptimized hardware implementation and a 5.3 % gain compared to a manually optimized 3rd party hardware implementation. In addition, we discuss the results of our technique on other applications, where the hardware/software partitioning is less clear. Among these, a TripleDES implementation shows a 29.3 % area gain using our technique. Based on our AES encoder results, we derive some guidelines for optimizing the impact of parameterisable configurations in general designs.

I. Introduction

Parameterisable configuration is a new concept for FPGA reconfiguration that was developed to allow for easier run-time reconfiguration (RTR) design [1]. This concept is implemented in the *TMAP* toolflow, an alternative to the normal FPGA tool flow. The *TMAP* tool flow allows us to automatically make a design run-time reconfigurable. It's principles and advantages are discussed in Section II-A. 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 aims to fill part of this void, by showing how parameterisable configuration can be used for fast hardware/software partitioning. The *TMAP* tool flow allows us to very quickly transform a hardware implementation to a hardware/software implementation that is optimized for area.

To explain how parameterisable configuration allows for an extended hardware/software partitioning, we use an application that is easy and clear to partition manually. One application that fits these needs is AES, Advanced Encryption Standard, an encryption algorithm detailed by the NIST in [2]. AES is explained in more detail in Section II-B. Next, in Section III, we will first detail the design decisions for our own AES implementation, k_AES, and then explain how exactly the *TMAP* tool flow is used on this application for hardware/software partitioning. Here we will also discuss the advantages of this tool flow compared to a more traditional hardware/software partitioning that does not use run time reconfiguration.

In Section IV 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 two other, manually optimized, designs, Cryptopan [3] and Avalon_AES [4]. And lastly, we will also use *TMAP* on both manually optimized designs, and discuss these results. Section V shows that *TMAP* can attain similar gains in designs that are more difficult to partition manually. In Section V, we will also provide guidelines for designing your implementation to take advantage of the concept of parameterisable configuration, and how to get a maximum gain out of the *TMAP* toolflow.

II. Background

A. Parameterisable 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, which means that 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.

Parameterisable configurations are a new concept for run-time reconfiguration on FPGAs. A parameterisable configuration is an FPGA configuration where some of the bits are expressed as boolean functions instead of boolean values.

Based on a parameterisable configuration, a normal 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.

Parameterisable configurations were developed to solve several problems with the normal way of using run-time reconfiguration in applications. Most of the current run-time reconfiguration techniques, such as the Modular design flow from Xilinx [5], require a lot of below RT-level design work and require the user to store all the possible FPGA configurations externally. The *TMAP* tool flow uses parameterisable configurations as a way to generate all possible configurations at run-time. To use run-time reconfiguration, only the parameterisable configuration needs to be stored, not all separate configurations. [6] and [1] give a detailed overview of this concept and the corresponding tool flow. It's main advantages are the high degree of automation, and the possibility to rapidly generate new configurations at run-time.

The *TMAP* tool flow is a modification of conventional FPGA mappers, these conventional mappers are unable to work with parameterisable configurations. The core principle of the *TMAP* tool flow is that the inputs of the design are split up in two groups, a group that is slowly changing (parameters), and a group that is swiftly changing (regular signals). Based on this information, the *TMAP* tool flow then produces a parameterisable configuration from the design. Certain bits in this parameterisable configuration will be boolean expression of the parameters. To generate the FPGA configuration for a specific parameter value, these boolean expressions are evaluated. The resulting FPGA configuration will be smaller and faster than the result of a regular FPGA toolflow. When a parameter changes, however, a new configuration needs to be generated and the FPGA will need to be reconfigured. Both the generation and the reconfiguration take time, so using run-time reconfiguration introduces an overhead. This overhead, and it's impact, is very application dependent, as a general rule one run-time reconfiguration will take in the order of milliseconds. Once more particulars of the application are known, there are several ways to reduce this overhead.

A much more detailed overview of parameterisable configurations is given in both [6] and [1]. Important for us is that the input of the *TMAP* toolflow is annotated VHDL and it's output is a working parameterisable 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 add these annotations, 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. The configuration manager is generally a CPU. (we will use a PowerPC on the FPGA but that is not the only option)

B. 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 AES-standards. We will only discuss the hardware implementation of the algorithm, and therefore only focus on the practical implications of AES.

In it's 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, who's 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 byte-transformations that are applied to the state. A specific combination of these operations is grouped in a round, and each round has a round key input. This round key value is different for each round, and these values are the result of the key expansion.

The input data is transformed by the addRoundkey transformation, with the first round key as input. The resulting data is then fed to the first round, in which the data is first substituted (`SubBytes()`), then the rows are shifted (`ShiftRows()`), the columns are mixed (`MixColumns()`) and lastly the second round key is added using the bit-wise XOR operation (`addRoundkey()`). The output data is then fed to the next round, and so on, for ten rounds in total. The only unknown in this process at this point are the round keys, whose generation will be discussed in the next Section.

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. We will first discuss these different transformations and then the complete key expansion.

The key expansion is done on the word -32-bit- level, in contrast to the State level data encoding transformations. The three word level transformations are `SubWord()`, `RotWord()` and `Rcon[i]`. `SubWord()` applies the S-box substitution to every byte of the word. `RotWord()` performs a cyclic permutation on the byte level, and `Rcon[i]` is a constant word that is round dependant. Their values can be found in [2].

In contrast to the encoding, not every word is generated in the same manner but the details are beyond the scope of this paper.

III. Automatic Hardware/Software partitioning

A lot of the hardware/software partitioning research is focused on the system level [7] [8] and the algorithmic aspects of hardware/software partitioning [9] [10]. By using our approach as a backend to the conventional hardware/software partitioning tool flows or methods, such as [11] [12], even more functionality can be moved from expensive FPGA resources to cheap CPU cycles. The *TMAP* tool flow is capable of doing this because it makes use of dynamic reconfigurability of the FPGA.

The *TMAP* tool flow is currently used for run-time reconfiguration on FPGAs. We want to use this tool flow to optimize a hardware design, to partition the design into a hardware and a software part in a highly automated fashion. The *TMAP* toolflow is based on the distinction between the slowly changing inputs, called the *parameters*, and swiftly changing inputs (regular inputs). Once we have selected the parameters, the tool (*TMAP*) generates a parameterisable configuration of the design. This parameterisable configuration can then be split up in a hardware and a software part. The hardware part are the bits in the parameterisable configuration that have boolean values (0 or 1). This will give us an incomplete FPGA configuration. The software part then consists of boolean expressions in the parameterisable configuration that are dependent on the parameters. Solving these boolean expressions will generate the values to complete the FPGA configuration, and is done in software by the configuration manager. As discussed in II-A, once the designer has selected the *parameters*, the *TMAP* toolflow is able to generate the parameterisable configuration automatically.

From a hardware/software point of view, the hardware part contains all the parts of the design that are dependent on the swiftly changing inputs. The parts of the design that are only dependent on the slowly changing inputs are moved to the software. The software takes care of the slowly changing inputs by reconfiguring the hardware every time these slowly changing inputs change. The hardware runs on the FPGA, the software on the PowerPC.

A. 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 usage 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, an approach without run-time reconfiguration, we would select the slowly changing inputs of the design, these are similar to the *parameters* selected in the *TMAP* toolflow. 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 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.

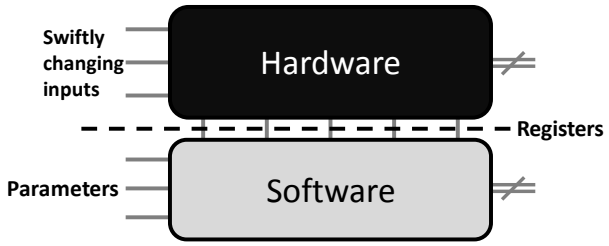

Fig. 1. Conventional hardware/software partitioning

When using an approach that includes run-time reconfiguration, several things change. Using the parameterisable configuration concept, and the *TMAP* toolflow, we can extend the hardware/software boundary by moving it to the configuration memory instead of adding registers to the design. In addition, because parameterisable configurations are used, the hardware will be a specialized circuit that is optimized for specific parameter values, 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.

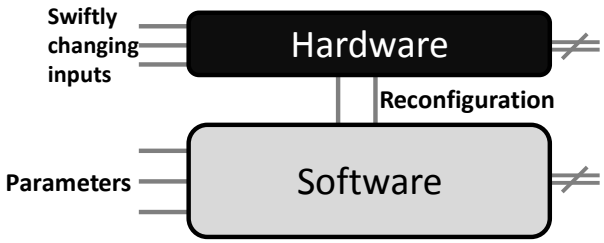

Fig. 2. Hardware/software partitioning using TMAP

Since the *TMAP* toolflow is automatic once the parameters in the design are selected, only one extra design decision is made, compared to a normal design, eg. the selection of the parameters. The time needed for such a decision is influenced by a lot of different factors and circumstances. If the designer is familiar with the design, as was the case in the AES example, partitioning the design using *TMAP* can be done in a few days. One of the big advantages of using *TMAP* is that the feasibility of any parameter selection can be checked in a few minutes. The other advantage is that, for some applications, we can achieve an extra area gain by extending the hardware/software boundary using run time

reconfiguration. As the *Quadratic* application in Section V-A shows, this gain can be significant.

B. k_AES

To explain how the *TMAP* toolflow works for hardware/software partitioning, we chose an application where the partition boundary is easy to identify. As described in Section II-B, 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 timescale. In most practical applications the key changes much slower than the input data. So, if we were to partition the AES application manually we would implement the key expansion in software and the data encoding in hardware.

To show that parameterisable configuration can significantly optimize a design, we will write our own AES implementation, *k_AES*. In the next section IV this design will be compared with several other designs. *k_AES* is almost a direct reflection of the hardware described in the NIST document [2]. The design was deliberately not optimized, and was therefore very simple to write and test.

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 complete implementation is pipelined and 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.

C. Automatic partitioning of k_AES

The next step is then to use the *TMAP* toolflow 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 an approach like this. For example, in our AES encoder, *k_AES*, a key is used to encode the data, this key changes very slowly compared to the data in most cases. Our RTR-based hardware/software partitioning can then be applied very quickly to this design. As we will show in the results, these kind of optimizations outperform manually optimized full hardware implementations.

IV. Comparison of AES implementations

Since it is part of *TMAP* tool flow, Quartus was used as the technology mapper for all designs, for both when TMAP was and was not used. Also, to provide a good basis to compare the different designs, we will only compare full LUT-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.

A. Resource optimization through HW/SW partitioning

In Sections II-A and III, we discussed the concept of parameterisable configuration, the *TMAP* tool flow and how they can be used to automatically partition a hardware design. The result of the *TMAP* tool flow can be seen in table I.

TABLE I
The resource usage, in LUT's, of several AES designs

	FPGA	TMAP	Gain
k_AES	45178	35843	20.3 %
Cryptopan	37874	37750	0.3 %
Avalon_AES	8448	8448	0%

The first important result in table I is that, as suggested in Section III, *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 software. For the *k_AES* design this means that almost the full key expansion has been moved to software.

One comment is needed here: the usage of any RTR platform will introduce a reconfiguration overhead. Each time the key changes, the new FPGA configuration has to be generated and the FPGA itself has to be reconfigured. The impact of this overhead is completely dependent on factors that vary from design to design and the context in which the design is used. With this qualifier in mind, the order of the time delay will be milliseconds.

We can compare these results to a manual hardware/software partitioning, by removing the key expansion from the design and adding 11 roundkey's as external inputs. The key expansion then needs to be implemented in software. If we manually partition the design this way, we get an AES design that is 35,568 LUTs, about 0.7 % smaller than the *k_AES* design. Both designs are very similar but the parameterisable tool flow introduces a small area overhead, resulting in the 300 LUT difference in both designs. In the case of AES however, this manual partitioning is very easy, the key expansion and the data encoding are clearly separated. In section V we will discuss other applications where this partitioning is less clear.

B. Comparison to opencores

To compare our automatically partitioned design to other designs, we looked at the publicly available AES implementations found on the opencores.org website. We found two AES-implementations that are part of bigger designs, the *Avalon_AES* implementation [4] and the *Cryptopan* [3] application. The original aims of these applications are less important and we will only discuss the AES-implementations within these applications.

In the first application, **Cryptopan**, 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 Cryptopan 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 cyle.

As the table I indicates, we see that, the un-optimized *k_AES* implementation is quite a bit larger than the *Cryptopan* design, 45178 LUT's compared to 37874 LUT's. The optimized version of k_AES however, *k_AES* shows a 5 % area gain, compared to the hardware optimized *Cryptopan* design. When comparing *k_AES* and *Cryptopan*, 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. It's very hard to compare both designs, because they are both Pareto efficient. Which one is better is decided by external factors.

If we look at the last column of table I, we see that the gains to the other designs for using *TMAP* are a lot less or even non-existent. The reason is closely linked to how parameterisable configurations work. Since we chose the key input as a parameter, all the signals that are only dependent on the key input are removed from the design. In the case of a fully parallelised design, like *k_AES*, this constitutes to almost the complete key expansion being moved to software. 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 that a (much) smaller part of the key-expansion is moved to software. This is clearly the case in both opencore designs.

These results were attained by comparing 128-bit key AES implementations. However, these results are transferable to both the 192-bit and the 256-bit key implementations. Since the key expansion is completely dependent on the key length, and the data stays 128-bit wide regardless of key length, it is likely that in those cases the key expansion represents an even greater fraction of the total design, and so could lead to even bigger percentage gains.

V. Automatic Hardware/software partitioning of other applications

In Sections III and IV, we used the AES application as an example of how *TMAP* can be used to automatically partition a hardware design. This application was chosen as a simple and clear way to demonstrate how gains are attained by this technique compared to manual partitioning. A great advantage is that the *TMAP* tool flow is fast and automatic. Once the parameter input is selected, the time needed to partition the design is in the same order as the time a conventional FPGA mapper needs to process the design. The time needed for a manual partitioning is several orders of magnitude larger, especially in the case of a complex design.

Not every application has such a clear hardware/software boundary as the AES application. There are many applications where the manual partitioning is both hard to identify and hard to implement. Additional examples show that, even in the case where the hardware/software boundary is less clear, similar gains are found using *TMAP*. We will discuss several, more complex, applications in Section V-A.

How large the gain is of using *TMAP* depends on the exact implementation. For example, both the *Cryptopan* and the *Avalon_AES* implementations show almost no gain for using *TMAP*. In Section V-B we will discuss guidelines on how to recognize and write applications that are suitable for *TMAP*.

A. More complex designs

To find applications that are harder to partition manually, we turned to the opencores.org website again. We found several applications that are all, in different degrees, more complex and harder to partition than the AES application.

The first application is **TripleDES** [13]. TripleDES is also an encoding algorithm. This 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. The result is then a 28.7 % area gain, from 3584 to 2552 LUTs. Here the partitioning is more complex because we have three different key expansions and three different keys that need to be moved to software. *TMAP* can partition this design without any extra interventions, once the parameters are selected, the tool flow is fully automatic. It takes in the order of minutes to generate the hardware/software partitioned design. The bulk of the design time in this case is finding the parameters.

The second application, **Quadratic** [14], is a hardware module to compute quadratic polynomial expressions. For this application the coefficients are a good parameter choice. In this case manually partitioning the design would be very difficult and time consuming. From the VHDL code, on the RT-Level, it is very hard to determine which parts of the design are only dependent on the coefficients. Additionally, it is not very clear what exact form this dependency takes, so manually moving these parts to software is very difficult. The toolflow solves those two problems automatically. The resulting design is 21.9 % smaller than the original design, from 569 to 444 LUT's.

As was indicated in Section III, this gain is only possible because *TMAP* uses run-time reconfiguration. The 21.9 % area gain is completely due to the fact that *Quadratic* will be

implemented on an FPGA and will use run-time reconfiguration. In a conventional hardware/software partitioned design the hardware has to be able process all possible coefficients. Using run-time reconfiguration and parameterisable configurations we can generate specialized designs for each possible coefficient, which results in the 21.9 % area gain.

Both of the above cases show that, even for more complex designs, the gains reached by using *TMAP* are significant. The *Quadratic* application shows that, even if manual partitioning is unfeasible, *TMAP* still attains large area gains.

B. General Guidelines for efficient parameterisable config.

As the previous Section shows, using *TMAP* can result in significant area gains, regardless of the design complexity. However, there must be other factors that influence how much gain is attainable for certain applications. In this Section we will offer guidelines, both for recognizing applications and for writing implementations that are easily parameterisable.

To use parameterisable configurations, the designer has to make one, very significant, design choice: The parameter selection. Once the parameter is selected, the tool flow is fully automatic. The main property for applications is whether or not it is easy to identify input signals that change on a much slower timescale than the other inputs. The slowly changing input signals are then prime candidates for selection as parameters. In all of the examples in this paper, the parameter selection was clear. In the case of encoding algorithms the key is slowly changing compared to the data input. In most cases the rate at which the inputs change is heavily dependent on the context of the application.

As the different AES implementations show, even if the application and the parameter are selected, it is still possible to have very different results for using *TMAP*, based on the choices made in the implementation. The wrong choices can lead to almost no gains at all, or the right ones to maximal gains. To find which choices we should make, we look closely at how the *TMAP* toolflow works.

Once the parameter is chosen, the toolflow will propagate this choice through the design to find all the hardware that is only dependent on the parameters. That part of the hardware is moved to the software. Only the hardware that has only parameters as inputs is moved, so from the moment a swiftly changing signal is an input, that part of the design stays in the hardware. In order to reach maximal gains, we need to maximize the part of the design that is only dependent on the parameters. In most cases this means making the parameter dependent parts of the design, or even the whole design, as parallelised as possible. The different AES implementations are perfect example: The more parallelised the design, (such as with *k_AES*) the greater the gain achieved by *TMAP*.

Additionally, *TMAP* leverages the run-time reconfiguration possibilities of FPGA's, which are all based in the LUT's. Using *TMAP* will not lead to a reduction in the usage of other FPGA resources, such as BRAM, but could change the tradeoff between different FPGA resources, as in [15].

VI. Conclusions

In this paper we have shown that the concept of parameterisable configurations and the corresponding *TMAP* tool flow can be used as an automatic hardware software partitioning tool. In contrast to other partitioning tools, the *TMAP* toolflow makes use of the inherent reconfigurability of FPGAs. This results in an extension of the hardware/software boundary of the design, more expensive FPGA resources can be traded for cheap CPU cycles. As our experiments have shown, the *TMAP* tool flow succeeds in reducing the FPGA resource usage by up to 20 %. In addition, once the the parameters are selected, the *TMAP* toolflow is automatic. The design time for a hardware/software partitioning using the *TMAP* toolflow is small compared to other methods of hardware optimization.

Acknowledgment

This work 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*, 2008, pp. 361–366.

[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] X. Inc., *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*, Xilinx Inc., 2002.

[6] K. Bruneel and D. Stroobandt, "Reconfigurability-aware structural mapping for LUT-based FPGAs," *Reconfigurable Computing and FPGAs, International Conference on*, pp. 223–228, 2008.

[7] K. A. and S. P.A., "Hardware/software partitioning for multifunction systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 17, no. 9, pp. 819–837, 1998.

[8] L. Silva, A. Sampaio, and E. Barros, "A constructive approach to hardware/software partitioning," *Formal Methods in System Design*, vol. 24, no. 1, pp. 45–90, 2004.

[9] J. Wu, T. Srikanthan, and C. Yan, "Algorithmic aspects for power-efficient hardware/software partitioning," *Mathematics and Computers in Simulation*, vol. 79, no. 4, pp. 1204 – 1215, 2008, 5th Vienna International Conference on Mathematical Modelling/Workshop on Scientific Computing in Electronic Engineering of the 2006 International Conference on Computational Science/Structural Dynamical Systems: Computational Aspects. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0T-4PRRBMB-1/2/d05ae632a20940ef03974ca0a5df106d

[10] W. Jigang, T. Srikanthan, and G. Chen, "Algorithmic aspects of hardware/software partitioning: 1d search algorithms," *IEEE Transactions on Computers*, vol. 59, no. 4, pp. 532–544, 2010.

[11] A. C. S. Beck and L. Carro, "Dynamic reconfiguration with binary translation: breaking the ilp barrier with software compatibility," in *DAC '05: Proceedings of the 42nd annual Design Automation Conference*. New York, NY, USA: ACM, 2005, pp. 732–737.

[12] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 3, pp. 659–681, 2006.

[13] "Opencores.org, the triple des project," http://www.opencores.org/project,3des_vhdl.

[14] "Opencores.org, the quadratic polynomial project," http://www.opencores.org/project,quadratic_func.

[15] T. Davidson, K. Bruneel, H. Devos, and D. Stroobandt, "Applying parameterizable dynamic configurations to sequence alignment," in *ParCo 2009, Proceedings*, Lyon, France, 2010, p. 8.