Implementing CNNs Using a Linear Array of Full Mesh CGRAs

Valter Mário[1], João D. Lopes[2](✉), Mário Véstias[3], and José T. de Sousa[1,2]

[1] IObundle Lda/IST, Lisboa, Portugal
[2] INESC-ID/IST, Lisboa, Portugal
joao.d.lopes@tecnico.ulisboa.pt, jts@inesc-id.pt
[3] INESC-ID/ISEL, Lisboa, Portugal

Abstract. This paper presents an implementation of a Convolutional Neural Network (CNN) algorithm using a linear array of full mesh dynamically and partially reconfigurable Coarse Grained Reconfigurable Arrays (CGRAs). Accelerating CNNs using GPUs and FPGAs is more common and there are few works that address the topic of CNN acceleration using CGRAs. Using CGRAs can bring size and power advantages compared to GPUs and FPGAs. The contribution of this paper is to study the performance of full mesh dynamically and partially reconfigurable CGRAs for CNN acceleration. The CGRA used is an improved version of the previously published Versat CGRA, adding multi CGRA core support and pre-silicon configurability. The results show that the proposed CGRA is as easy to program as the original full mesh Versat CGRA, and that its performance and power consumption scale linearly with the number of instances.

Keywords: Convolutional Neural Networks · Coarse Grained Reconfigurable Arrays · Reconfigurable computing · Embedded systems

1 Introduction

During the last few years we have seen extensive developments in Machine Learning (ML), Artificial Intelligence (AI) and the Internet of Things (IoT). These advances increased the complexity of algorithms and the need to lower the size and power consumption of the hardware platforms used to run them. To tackle computational complexity, it is common practice to use dedicated hardware to speed up computations. However, using non-programmable hardware offers poor scalability, prevents updates and upgrades, and increases the cost of design errors. For these reasons, programmable hardware such as GPUs or FPGAs are preferred for these functions but their large size and high power consumption prevents their use in embedded devices powered by batteries.

For embedded applications a more suitable accelerator is the Coarse Grained Reconfigurable Array (CGRA), which is also programmable and can be made small and energy efficient. A CGRA is a collection of programmable Functional Units (FUs) and embedded memories connected by programmable interconnects.

© Springer Nature Switzerland AG 2020

When programmed, specialized hardware datapaths are formed in the CGRA, able to execute the target tasks orders of magnitude faster than a regular CPU.

In the last 25 years, CGRAs have become the subject of several research papers [5]. CGRA architectures can be homogeneous [6], using only one type of programmable FUs, or heterogeneous [8], using FUs of different types. As for the programmable interconnections between FUs, direct neighbour-to-neighbour connections or 2D-Mesh networks are the most popular choices [13]. CGRAs can be statically reconfigurable, i.e., they are configured once for an entire application [7], or dynamically reconfigurable [10], that is, they are reconfigured at runtime. Some CGRAs can only be fully reconfigurable [10], whereas others use partial reconfiguration [4,6,12]. The success of any architecture depends crucially on the available tool support [13]. Different types of compilers have been proposed [11] for CGRAs but this is still a critical weakness preventing these architectures from becoming mainstream.

To address the lack of compiler tools, an extreme approach of using a full mesh CGRA called Versat has been proposed in [9]. Being a full mesh, the compilation complexity, namely the need to place and route the designs is removed, and the configurations can even be produced on-the-fly by the application itself. Versat featured self-generated dynamic and partial reconfiguration driven by an external controller unit. The Versat core was good for applications that require a small number of FUs. However, ML applications require a massive amount of parallelism, which was unattainable with the original Versat core. In fact, its full mesh structure cannot scale spatially, creating routing congestion and forcing lower operation clock frequencies.

To target ML applications, this work proposes the use of a multi-core Versat architecture controlled by a simple RISC-V [2] processor. The RISC-V architecture is supported by the GNU toolchain, enabling the development of applications using the C and C++ languages.

2 The Deep Versat Architecture

The multi-core architecture has been called Deep Versat and is organised as a ring of Versat cores. This topology is one of the simplest that can utilise multiple instances, and its ring structure facilitates the reuse of the data left in the accelerator between different configurations.

Each individual Versat core keeps the full mesh topology of the original proposal, for retaining its programmability, but the size of each core is limited to 10 FU output ports. A large number of cores can be added to the ring, depending on the needs of the target application, and the limit is only the device size. A block diagram of this architecture is depicted in Fig. 1.

Figure 1 shows several Versat cores linearly interconnected forming a ring. Since CGRAs are used to accelerate program loops a linear topology can easily exploit loop optimisation techniques such as loop unrolling. The ring topology facilitates the reuse of the data left in any of the Versat cores by the next configuration applied to the array.

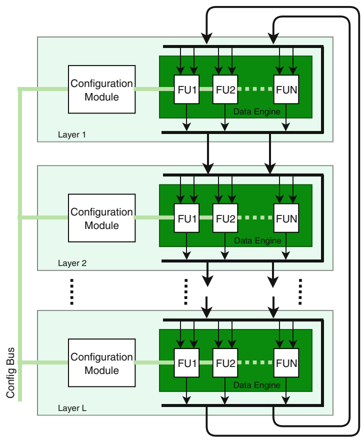

Fig. 1. Deep Versat architecture.

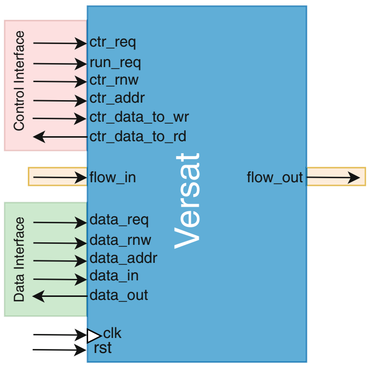

Fig. 2. Versat symbol and interface.

In each Versat core, the FUs can select as inputs any FU outputs from the previous core and from itself. Hence, each core can only produce at most 10 output ports so that the number of selection inputs for each FU port does not exceed 20: 10 from the previous core and 10 from the current core. This way, the routing complexity of each core is similar to that of the previous Versat core proposal and independent of the number of cores. The number of Versat cores is only limited by the device size.

The individual Versat cores have been streamlined and simply comprise the Data Engine (DE), which is formed by its FUs, and the Configuration Module (CM), which holds the FU configurations. The interface of each CGRA is represented in Fig. 2 and consists of Control, Data and Flow interfaces. The Control Interface is used to read and write to the Versat registers. The Data Interface is used by the processor or DMA core to read and write data from/to the Versat memories. The Flow Interface consists of the `flow_in` and `flow_out` buses, and it is used to connect two consecutive Versat cores.

3 The RV32 Deep Versat System

To control the Deep Versat core, the picoRV32 open source processor [3] has been adopted. The picoRV32 processor is a RISC-V architecture which can be programmed using the well known `gcc` and `g++`, C and C++ compilers, respectively.

The picoRV32 processor runs the application code and uses the Deep Versat core as an accelerator. For many applications the use of a single picoRV32 core and a Deep Versat core will suffice. Other applications may require a more powerful processor for running the software, for example, a superscalar RISC-V or ARM core. The system shown in Fig. 3 is composed of the picoRV32 processor having as peripherals the Deep Versat core and a UART core.

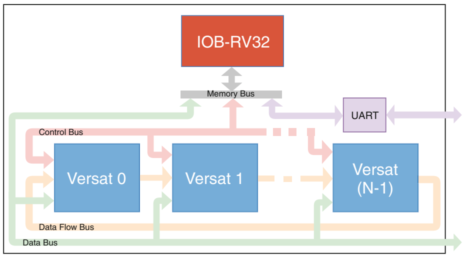

Fig. 3. The RV32 Deep Versat system.

Table 1. Memory map.

Peripheral	Base address
UART module	0×10000000
Deep Versat control bus	0×11000000
Deep Versat data bus	0×12000000

The peripherals are memory mapped as described in Table 1. The picoRV32 system uses a 32-bit address bus where 8 bits are used to choose the peripheral. Hence, there are 24 bits to address Deep Versat. 15 out of these 24 bits are used for internally addressing each Versat core. The remaining 9 bits are used to select the Versat core, so in theory the Deep Versat core may contain as many as 512 Versat cores for achieving maximum parallelism and acceleration.

4 Pre-silicon Configurability

Pre-silicon configurability consists in the ability to choose the set of FUs for the Versat core before the circuit is implemented. This powerful feature enables tailoring and optimizing Versat for different applications. The automatic generation of the FU array has been implemented using Verilog macros and *generate for* statements (e.g. Fig. 4).

```
generate
 for (i=0; i < ‘nALU; i=i+1) begin : add_array
  xalu alu (
    .clk(clk),
    .rst(run_reg),
    // Data IO
    .data_bus(data_bus),
    .result(data_bus[‘DATA_ALU0_B - i*‘DATA_W -: ‘DATA_W]),
    // Configuration data
    .configdata(config_reg_shadow[‘CONF_ALU0_B -
                       i*‘ALU_CONF_BITS -: ‘ALU_CONF_BITS])
  );
 end
endgenerate
```

Fig. 4. Pre-silicon configuration of the ALU array.

To configure Versat at pre-silicon time, the user sets the types and numbers of FUs to be instantiated using the macros in the main header file. This file can be edited for each specific application. The size of each memory can also be specified, allowing Versat to have memories of different sizes. An obvious future improvement is to replace the macros by Verilog generic parameters, which will allow instantiating multiple and heterogeneous Versat cores.

5 The Deep Versat API

As can be seen in Fig. 1, the Deep Versat hierarchy is the following: Deep Versat is an array of Versat cores, and each Versat is an array of FUs. Modeling hardware with an object oriented language is convenient as the hardware modules can be represented by classes whose methods are used to configure and operate the modules. The Deep Versat API has been written in the C++ object oriented programming language.

To represent the Deep Versat hardware an array of *CVersat* objects is declared. One may ask why the size of this array must be declared if the number of layers is already known from the Verilog code. The reason is that the API makes it possible to work with a virtual Deep Versat core whose size is different from its physical size. This is useful if the application does not need to use all cores or would need to use more cores than the ones actually present. This is called virtual hardware, a feature that is planned but not yet implemented.

After declaring the CVersat object, one needs to populate it with a set of FUs. At the moment, the available FU types are the following: ALU (of 2 different types), Barrel Shifter, 1 multiplier type and 1 new multiply-accumulate (MAC) type. The FU population in each CVersat class has to of course match the FUs in the actual Versat core, and in the future this can be automated so that both the hardware and software are created consistently. The FUs in each Versat core can be operated by the control processor by writing to their configuration registers in the configuration module. There are registers to configure the FU function and connections.

6 The CNN Application: Handwritten Digit Recognition

The chosen application is a handwritten digit recognition program that uses the well known *mnist* dataset [1] and performs Convolutional Neural Network (CNN) inference on a previously trained network (Fig. 5). Each 28×28 image passes through a series of layers in order to be classified. The layers are of the following types: convolutional, pooling, fully connected and softmax. The output represents the most likely classification, from digit 0 to digit 9, and its respective probabilistic value between 0 and 1.

The convolutional layer performs the multiply-accumulate function of each element of the filter by the corresponding pixel of the image. 22 matrix filters of dimension 5×5 are used, which produces a $22 \times 5 \times 5$ tensor. The maxpool layer is responsible for down sampling the largest images, from size 24×24 down to size 12×12, while keeping the relevant information. The process used in this layer is simple: it goes through the 24×24 image and takes the greatest value in each 2×2 region. Hence, the output of the pooling layer is $22 \times 12 \times 12$ tensor. The (fully) connected layer takes the $22 \times 12 \times 12$ tensor produced and again uses a convolutional process to turn it into a 10-element vector, where each position contains the votes for the respective digit. The last layer of the CNN is the softmax layer. It finds the digit with most votes and classifies it as the most likely handwritten digit represented in the image.

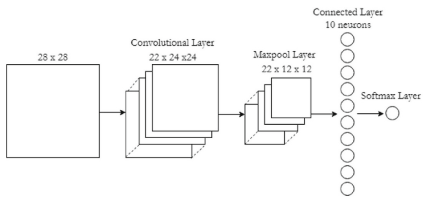

Fig. 5. CNN architecture with a single convolutional layer.

Table 2. Execution time per layer when running on the ARM processor.

Layer	Execution time (μs)	%
Convolutional	32839	88.41%
Maxpool	1300	3.50%
Connected	2998	8.07%
Softmax	5	0.01%
Total	37142	100%

The application is divided in four parts corresponding to the four CNN layers. The time profile of the application is presented in Table 2 for a 667 Mhz ARM Cortex-A9 processor. The table shows that the layer that takes most of the execution time is the convolutional layer, which has been chosen for acceleration. The software code that implements this layer is divided in two parts: (1) the *preparation* of the images for convolution with the filter, which is basically a replication of the data in the memory, and (2) the convolution itself, which is done by the General Matrix Multiply (GeMM) algorithm.

As there are 22 filters of 5×5 coefficients, a matrix B of size 22×25 is created. The input image is prepared, that is, it is transformed into a 576×25 matrix A, where the number of rows is 24×24 and the number of columns is 5×5. It can be shown that the convolution is equivalent to computing matrix $C = AB^T$. The preparation of matrix A, being just a replication of the image data, is not very interesting from the point of view of acceleration. The part that takes most of the execution time and is candidate for acceleration is the GeMM algorithm.

Each Versat core can execute 2 nested loops. Thus, a single Versat core with the new multiply-accumulate FU (MULADD) could be used to run the GeMM algorithm. However, to scale the performance, multiple Versat cores are used by distributing the workload among them using the loop unrolling technique. The inner-most loop of the GeMM, which in this case goes from 0 to 24 is distributed over 5 cores, resulting in a 5-core Deep Versat architecture. Note that this is possible because there are no data dependencies between iterations. The first core computes elements 0 to 4, the second core computes elements 5 to 9 adding its result to the result coming from the first core and so on up to the fifth core. This creates a pipeline structure with 5-cycle latency and a throughput of one result per cycle. Therefore, the execution time of the GeMM is reduced roughly 5 times, which is the expected acceleration for 5 cores running in parallel compared to a single core. For simplicity, only 2 out of the 5 cores are shown in Fig. 6.

As can be seen in Fig. 6, each Deep Versat core uses 1 MULLADD FU and 4 AGU blocks from 4 memory units (shown in blue). Two of the AGUs are used for addressing the MULADD operands, another is used for controlling the MULADD and the last is used for addressing the result from the previous core, which had been stored in a memory of the current core. For a single MULADD,

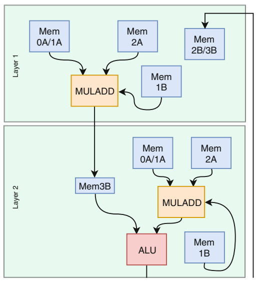

Fig. 6. Two convolutional layers.

2 data memories would be enough but in fact 5 memories per core have been used, because this is the number required by the fully connected layer, which was accelerated in a similar way. Each of the 5 memories can hold up to 8192 data words. If needed 2 MULADDs per core could be used to double the parallelism.

Given the chosen memory sizes, the 14400-word matrix A needs to be stored in 2 different memories. Half the matrix is stored in memory 0 and the other half is stored in memory 1, as shown in Fig. 6. The same happens to the output matrix C, which has $576 * 22$ words divided between memories 2 and 3. The Deep Versat API code used to configure layers 1 and 2 of the designed datapath is presented in Fig. 7 to illustrate the process. The `setConf` method, whose details are not explained here due to lack of space, is used to create full configuration of an FU, and the `writeConf` method is used to write the configuration to the configuration registers. The `setStart` method is used to set only one configuration register, the start address of a memory port. Partial reconfiguration is clearly illustrated here: `setConf` configures just one FU and `setStart` configures just one configuration register of an FU. The `versatRun` function runs Deep Versat after waiting for the previous run to finish.

7 Experimental Results

The described system has been run on a Xilinx XCKU040 FPGA of the Kintex UltraScale product family, and compared with 2 other systems running the same application: a RISC-V + single Versat system, and an ARM Cortex-A9 processor + 4 General Matrix Multiply (GeMM) IPs.

Table 3 compares the FPGA resources used and the execution performance in each system. RAM stands for 36kbit RAM blocks, Frequency is the clock frequency, WNS stands for Worst Negative Slack, Time is the execution time and Speedup is the ratio of the execution time on the ARM system over the execution time on each system. Note that the ARM core runs independently at 667 MHz.

The RISC-V + Deep Versat system is around five times larger than the RISC-V + Versat system, which is expected since Deep Versat integrates 5 single Versat cores. It is not possible to make direct size comparisons to the ARM + 4 GeMM

```
void gemmBT(CVersat v1, CVersat v2) {
 int rowsA = 24*24, colsA = 25, rowsB = 22;
 int i;

 //Config MEM2A to read filter weigths (0-110)
 v1.memPort[m2A].setConf(0, 22, 1, 0, 5, 5, 0, 0, 0);
 v1.memPort[m2A].writeConf();
 v2.memPort[m2A].setConf(0, 22, 1, 1, 5, 5, 0, 0, 0);
 v2.memPort[m2A].writeConf();

 //Config MEM1B to control MULADDs
 v1.memPort[m1B].setConf(0, 22, 1, 1, 5, 5, sADDR, -5, 0);
 v1.memPort[m1B].writeConf();
 v2.memPort[m1B].setConf(0, 22, 1, 2, 5, 5, sADDR, -5, 0);
 v2.memPort[m1B].writeConf();

 //Config MEM0A to read 1st half of matrix A (0-7199)
 v1.memPort[m0A].setConf(0, 22, 1, 0, 5, 5, 0, -5, 0);
 v1.memPort[m0A].writeConf();
 v2.memPort[m0A].setConf(5, 22, 1, 1, 5, 5, 0, -5, 0);
 v2.memPort[m0A].writeConf();

 //Config MULADD
 v1.muladd[0].setConf(sMEMA[2], sMEMB[1], sMEMA[0], MULADD);
 v1.muladd[0].writeConf();
 v2.muladd[0].setConf(sMEMA[2], sMEMB[1], sMEMA[0], MULADD);
 v2.muladd[0].writeConf();

 //Pipeline layer 1 - 2
 v2.memPort[m3B].setConf(0, 22, 0, 8, 5, 1, sMULADD_p[0]);
 v2.memPort[m3B].writeConf();

 //AluLite layer2
 v2.alulite[0].setConf(sMEMB[3], sMULADD[0], ALULITE_ADD);
 v2.alulite[0].writeConf();

 //Save 1st part of the result(6336) in v1.MEM2 (1856-8191)
 v1.memPort[m2B].setConf(1856, 22, 1, 20, 5, 1, sALULITE_p[0]);
 v1.memPort[m2B].writeConf();

 //Running layers 1 and 2
 for (i=0; i<rowsA/2; i++) {
   v1.memPort[m0A].setStart(i*colsA+0);
   v2.memPort[m0A].setStart(i*colsA+5);

   //We get 22 results in each run
   v1.memPort[m2B].setStart(1856+rowsB*i);

   versatRun();
 }
}
```

Fig. 7. Code to configure the datapath presented in Fig. 6.

Table 3. FPGA implementation and execution results.

	LUTs	FFs	RAM	DSPs	Frequency (MHz)	WNS (ns)	Time (μs)	Speedup
RISC-V + Versat	7081	3460	62	8	100	0.521	9780	3.36
RISC-V + DeepVersat	40478	14631	196	20	100	0.292	1689	19.44
ARM + 4 GeMM	16706	17715	16	16	100	NA	3961	8.25
ARM	NA	NA	NA	NA	667	NA	32839	1

system, since the ARM processor is a hard macro. However, it should be clear that the ARM + 4 GeMM system is much larger if implemented in a ASIC compared to both the RISC-V + Versat or the RISC-V + DeepVersat systems. This means that combining RISC-V and Versat cores can be very competitive compared to combining standard processors and custom hardware.

As for the execution results, Deep Versat can effectively accelerate this application, and this is true for many other ML algorithms. Even the RISC-V + single

Versat system has a speedup of 3.36x compared to the standalone ARM system. The RISC-V + Deep Versat system is almost 20x faster than the ARM system, and it is faster than the ARM + 4 GeMM IP system by 2.3x. As expected, the RISC-V + Deep Versat is more than 5x faster than the RISC-V + Versat system due to the almost perfect parallelism of the inner loop and some code optimizations that have been done for RISC-V + Deep Versat after the results for the RISC-V + Versat system had been obtained.

8 Conclusions

This paper presents an implementation of a Convolution Neural Network (CNN) using a linear array of full mesh dynamically and partially reconfigurable Coarse Grained Reconfigurable Arrays (CGRAs) called Deep Versat. This design extends the previous single core Versat design by adding spatial scalability: performance scales linearly with the number of Versat cores without impacting the frequency of operation.

The Versat core has been enhanced with the capability of being configured at pre-silicon time. It can be configured with the types and quantities of FUs required. A new Multiply-Accumulate unit (MAC) has been developed, which is useful for the CNN application and others. Additionally, picoVersat, the previous Versat controller, which was only Assembly programmable, has been removed from the architecture which now relies on an external processor for control. A RISC-V open source core called picoRV32 has been adopted for controlling Deep Versat.

The new Deep Versat core is a ring of several new Versat cores created using Verilog generate statements. With the RISC-V processor used for control, which is programmable using the GNU toolchain, a C++ software API for reconfiguring and running Deep Versat has been developed. In essence, Deep Versat retains the programmability of the previous Versat core but can be pre-silicon configured to optimise the size and power consumption of the target application. Like the previous Versat architecture, the new Deep Versat architecture is also dynamically and partially reconfigurable to take advantage of the space and time locality of hardware configurations.

In the CNN algorithm, the neurons are organized in layers and it is important to have as many of them as possible working in parallel. The layers only differ in the activation functions of the neurons, the way they are interconnected or the way they access data from the memories. The chosen application contains the fundamentals of modern AI algorithms for image recognition, and is a perfect fit for CGRA implementation. In this paper, a 5-core Deep Versat instance has been used to accelerate a CNN handwritten digit recognition algorithm. The implementation runs 19x faster compared to an ARM Cortex-A9 processor hard macro in a Xilinx FPGA. If the ARM system is accelerated using 4 GeMM IP cores, the RISC-V + Deep Versat system is still more than 2 times faster.

It is concluded that by using a multi-core CGRA architecture, the system size grows proportionally with the workload and the clock frequency does not degrade with size. Given the preliminary nature of this work, the considered CNN network is not too complex, but the results clearly show that the same methodology can be applied to larger CNNs, serving as a good alternative to FPGAs and GPUs.

Acknowledgments. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects PTDC/EEI-HAC/30848/2017 and UIDB/50021/2020.

References

1. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/
2. RISC-V: The Free and Open RISC Instruction Set Architecture. https://riscv.org/
3. PicoRV32 - a RISC-V CPU. https://github.com/cliffordwolf/picorv32 (2019)
4. Baumgarte, V., Ehlers, G., May, F., Nückel, A., Vorbach, M., Weinhardt, M.: PACT XPP - a self-reconfigurable data processing architecture. J. Supercomput. **26**(2), 167–184 (2003). https://doi.org/10.1023/A:1024499601571
5. De Sutter, B., Raghavan, P., Lambrechts, A.: Coarse-grained reconfigurable array architectures. In: Bhattacharyya, S.S., Deprettere, E.F., Leupers, R., Takala, J. (eds.) Handbook of Signal Processing Systems, pp. 449–484. Springer, Boston (2010). https://doi.org/10.1007/978-1-4419-6345-1_17
6. Ebeling, C., Cronquist, D.C., Franklin, P.: RaPiD — reconfigurable pipelined datapath. In: Hartenstein, R.W., Glesner, M. (eds.) FPL 1996. LNCS, vol. 1142, pp. 126–135. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61730-2_13
7. Hartenstein, R., Herz, M., Hoffmann, T., Nageldinger, U.: Mapping applications onto reconfigurable kressarrays. In: Lysaght, P., Irvine, J., Hartenstein, R. (eds.) FPL 1999. LNCS, vol. 1673, pp. 385–390. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-540-48302-1_42
8. Heysters, P.M., Smit, G.J.M.: Mapping of DSP algorithms on the MONTIUM architecture. In: Proceedings of the International Parallel and Distributed Processing Symposium, p. 6, April 2003
9. Lopes, J.D., de Sousa, J.T.: Versat, a minimal coarse-grain reconfigurable array. In: Dutra, I., Camacho, R., Barbosa, J., Marques, O. (eds.) VECPAR 2016. LNCS, vol. 10150, pp. 174–187. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61982-8_17
10. Mei, B., Lambrechts, A., Mignolet, J.-Y., Verkest, D., Lauwereins, R.: Architecture exploration for a reconfigurable architecture template. Des. Test Comput. **22**(2), 90–101 (2005)
11. Mei, B., Vernalde, S., Verkest, D., De Man, H., Lauwereins, R.: DRESC: a retargetable compiler for coarse-grained reconfigurable architectures (2002)
12. Hemani, A., Shami, M.A.: Partially reconfigurable interconnection network for dynamically reprogrammable resource array (2009)
13. Wijtvliet, M., Waeijen, L., Corporaal, H.: Coarse grained reconfigurable architectures in the past 25 years: overview and classification (2016)