Versat, a Minimal Coarse-Grain Reconfigurable Array

João D. Lopes and José T. de Sousa(✉)

INESC-ID/IST, University of Lisbon, Lisbon, Portugal
jose.desousa@inesc-id.pt

Abstract. This paper introduces Versat, a minimal Coarse-Grain Reconfigurable Array (CGRA) used as a hardware accelerator to optimize performance and power in a heterogeneous system. Compared to other works, Versat features a smaller number of functional units and a simpler controller, mainly used for reconfiguration and data transfer control. This stems from the observation that competitive acceleration can be achieved with a smaller array and more flexible reconfigurations. Partial reconfiguration plays a central role in Versat's runtime reconfiguration scheme. Results on core area, frequency, power and performance are presented and compared to other implementations.

Keywords: Reconfigurable computing · Coarse-grain reconfigurable arrays · Heterogeneous systems

1 Introduction

A suitable type of reconfigurable hardware for embedded devices is the Coarse-Grain Reconfigurable Array (CGRA) [1]. Fine grain reconfigurable fabrics, such as FPGAs, are often too large and power hungry to be used as embedded cores. It has been demonstrated that certain algorithms can run orders of magnitude faster and consume lower power in CGRAs when compared to CPUs (see for example [2]).

A CGRA is a collection of programmable functional units and embedded memories, interconnected by programmable switches for forming hardware datapaths that accelerate computations. The reconfigurable array is good for accelerating program loops with data array expressions in their bodies. However, the parts of the program which do not contain these loops must be run on a more conventional processor. For this reason, CGRA architectures normally feature a processor core. For example, the Morphosys architecture [3] uses a RISC processor and the ADRES architecture [4] uses a VLIW processor.

This work started with 3 observations: (1) because of Amdahl's law, accelerating kernels beyond a certain level does not result in significant overall acceleration and energy reduction of the application; (2) the kernels that are best accelerated in CGRAs do not require much control code by themselves; (3) the compute intensive inner loops that are normally accelerated in CGRAs tend to

I. Dutra et al. (Eds.): VECPAR 2016, LNCS 10150, pp. 174–187, 2017.
DOI: 10.1007/978-3-319-61982-8_17

cluster in the code. About observation (2), note that typical target kernels are transforms (IDCT, FFT, etc.), filter banks (FIR, IIR, etc.), and others. Observation (3) refers to loop nests and loop sequences where the data produced in one loop is consumed in the next loop. If the hardware does not support these constructs, the associated processor needs to reconfigure the array after each inner loop, and the resulting time overhead may cancel the acceleration gains. Previous work has targeted this problem by proposing CGRAs that can support nested loops. For example, the approach in [5] supports nested loops using special address generation units and has been successfully used in commercial audio codec applications.

This paper extends the work in [5] to sequences of loop nests, achieving further acceleration and increasing the granularity of the tasks handled by the reconfigurable array. We propose a new architecture, Versat, which uses a relatively small functional unit array coupled with a very simple controller. A smaller array limits the size of the data expressions that can be mapped to the CGRA, forcing large expressions to be broken down into smaller expressions executed sequentially in the CGRA. Therefore, Versat requires mechanisms for handling large numbers of configurations efficiently and flexibly.

Versat cores are to be used as co-processors in an embedded system containing one or more commercial application processors. The advantage of having one or more Versat cores in the system is to optimize performance and energy consumption during the execution of compute intensive tasks. Application programmers can use the Versat cores by calling procedures that get executed on the Versat cores. For that purpose, a Versat API library must be linked with the host application. The API library is created by Versat programmers. In this way, the software and programming tools of the CGRA are clearly separated from those of the application processor.

A compiler for Versat has been developed. The compiler is simple as we have restricted its functionality to the tasks that CGRAs can do well. The syntax of the programming language is a small subset of the C/C++ language, with a semantics that enables the description of hardware datapaths. The compiler is not described in this paper whose main thrust is the description of the architecture and VLSI implementation.

In order to make the reconfiguration process efficient, full reconfiguration of the array should be avoided. In this work we exploit the similarity of different CGRA configurations by using *partial reconfiguration*. If only a few configuration bits differ between two configurations, then only those bits are changed. Most CGRAs are only fully reconfigurable [3,4,6] and do not support partial reconfiguration. The disadvantage of performing full reconfiguration is the amount of configuration data that must be kept and/or fetched from external memory. Previous CGRA architectures with support for partial reconfiguration include RaPiD [7], PACT [8] and RPU [2]. RaPiD supports dynamic (cycle by cycle) partial reconfiguration for a subset of the configuration bitstream, using smaller stored contexts. In PACT, one of its processors has access to the configuration memory of the array, but using this feature for partial reconfiguration is

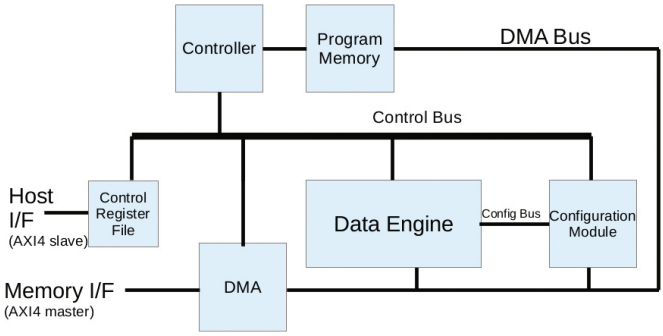

Fig. 1. Versat top-level entity.

reportedly slow and users are recommended to avoid it and resort to full reconfiguration whenever possible. In RPU, a kind of partial reconfiguration called Hierarchical Configuration Context is proposed to mitigate these problems. In this work we propose a configuration register file using registers of variable length, organized in configuration spaces and low-level configuration fields, where each register corresponds to a configuration field, and we allow random access to the configuration fields. This scheme is more flexible than the hierarchical organization of the configuration contexts in [2].

2 Architecture

The top level entity of the Versat module is shown in Fig. 1. Versat is designed to carry out computations on data arrays using its Data Engine (DE). To perform these computations the DE needs to be configured using the Configuration Module (CM). A DMA engine is used to transfer the data arrays from/to the external memory. It is also used to initially load the Versat program and to move CGRA configurations to/from external memory.

The Controller executes programs stored in the Program Memory (8 kB). A program executes an algorithm, coordinating the reconfiguration and execution of the DE and the DMA. The controller accesses the modules in the system by means of the Control Bus.

Versat has a host interface and a memory interface. The host interface is used by a host system to command the loading and execution of programs. The host and the Controller communicate using the Control Register File (CRF). The memory interface is used to access data from an external memory using the DMA.

2.1 Data Engine

The Data Engine (DE) has a fixed topology using 15 functional units (FUs) as shown in Fig. 2. It is a 32-bit architecture and contains the following FUs: 4 dual-port 8 kB embedded memories, 4 multipliers, 6 Arithmetic and Logic Units

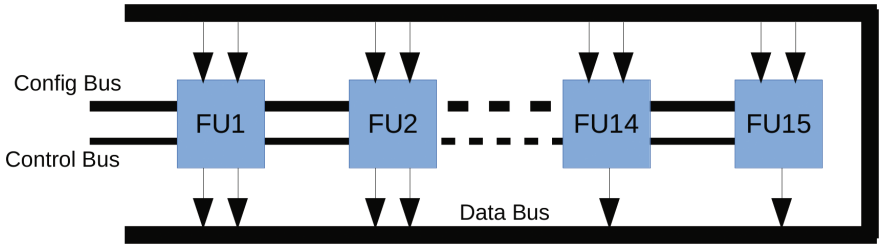

Fig. 2. Data engine.

(ALUs) and 1 barrel shifter. The Controller can read and write the output register of the FUs and can read and write to the embedded memories. In this work, embedded memory blocks are treated like any other FU by our mapping tool.

Each FU contributes its 32-bit output(s) to a wide Data Bus of 19×32 bits, and is able to select one 32-bit data bus entry for each of its inputs. The FUs read their configurations from the Config Bus. Each FU is configured with an operation and input selections. The *coarse-grain reconfiguration* means that there is a fixed set of operations available in the accelerator. For example, an ALU can be configured to perform addition, subtraction, logical AND, maximum and minimum, etc.

In Fig. 3, it is shown in detail how a particular FU is connected to the control, data and configuration busses. The FU is labeled FU5 and it is of type ALU. It is a pipelined ALU with 2 pipeline stages. The last pipeline stage stores the output of the ALU (output register). FU5 is selecting one of the 19 sub-busses of the Data Bus for each of its two inputs. Although Fig. 2 shows the Config Bus going to all FUs, in fact only the configuration bits of each FU are routed to that FU. These bits are called the *configuration space* of the FU. The configuration space is further divided in *configuration fields* with specific purposes. In Fig. 3, the example ALU has a configuration space with 3 configuration fields: the selection of the ALU's input A (5 bits), the selection of the ALU's input B (5 bits) and the selection of the ALU's function (4 bits). Our partial reconfiguration scheme works at the field level. Fields can be reconfigured one by one by the Controller. The ALU output (pipeline register 1) can be read or written by the Controller as shown in the figure. This feature enables a functional unit to be used as a shared register between the Controller and the DE.

From the explanation in the previous paragraph, one concludes there are direct connections from any FU to any other FU. This complete mesh topology may be unnecessary but it greatly simplifies the compiler design as it avoids expensive place and route algorithms commonly used in CGRAs. More compact interconnect may be developed in the future simultaneously with compiler improvements. In any case, the interconnect consumes very little power since Versat is reconfigured only after a complete program loop is executed in the DE. Moreover our IC implementation results indicate that only 4.04% of the core area is occupied by the full mesh interconnect, which means there is little

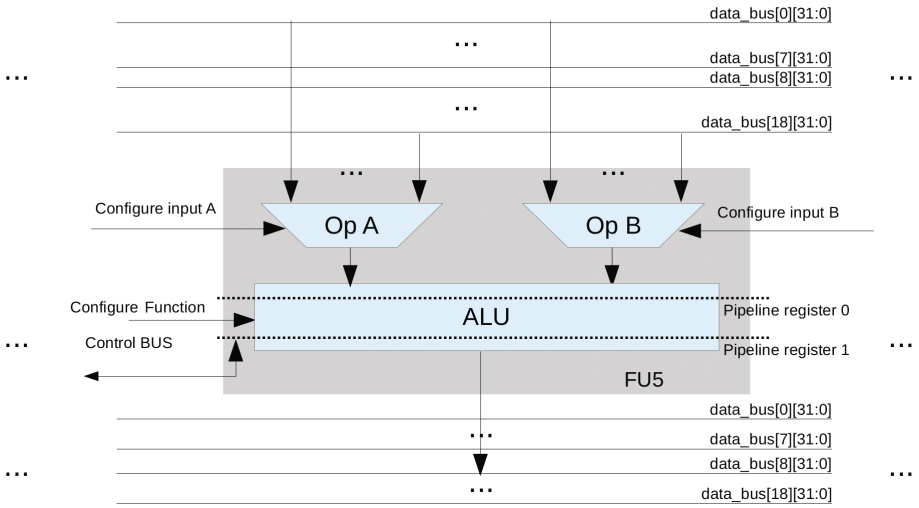

Fig. 3. Functional unit detail.

motivation to optimize the interconnect. One could argue that a full mesh topology also limits the frequency of operation. However, our IC implementation is able to work at a maximum frequency of 170 MHz in a 130 nm process, while many target applications that we have investigated, for example, in the multimedia space, are required to work at even lower frequencies because of power and energy constraints.

Each configuration of the DE corresponds to one or more hardware datapaths. Datapaths can have parallel execution lanes to exploit Data Level Parallelism (DLP) or pipelined paths to exploit Instruction Level Parallelism (ILP). Given enough resources, multiple datapaths can operate in parallel in Versat. This corresponds to having Thread Level Parallelism (TLP). In Fig. 4, three example hardware datapaths that can be mapped onto the DE are illustrated. Datapath (a) implements a pipelined vector addition. Despite the fact that a single ALU, configured as an adder, is used, ILP is being exploited: the memory reads, addition operation and memory write are being executed in parallel for consecutive elements of the vector. Datapath (b) implements a vectorized version of datapath (a) to illustrate DLP. The vectors to be added spread over memories M0 and M2, so that 2 additions can be performed in parallel. ILP and DLP can be combined to yield very parallel datapaths such as datapath (c), whose function is to compute the inner product of two vectors. Four elements are multiplied in parallel and the results enter an adder tree followed by an accumulator.

Each memory port is equipped with an Address Generator Unit (AGU) to access data from the embedded memories during the execution of a program loop. The discussion of the details of the AGU falls out of the scope of this paper. Our scheme is similar to the one described in [9] in the sense that both schemes use parallel and distributed AGUs. We will simply state the following

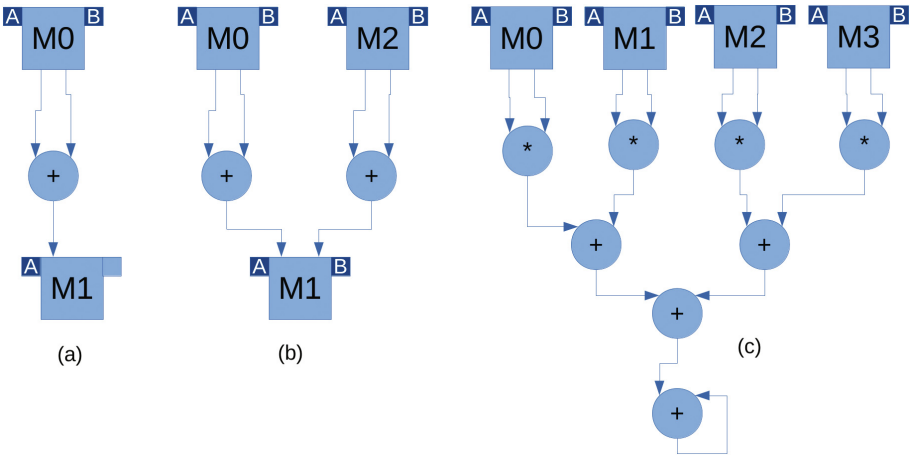

Fig. 4. Data engine datapaths.

properties of our AGUs: (1) two levels of nested loops are supported (reconfiguration after each inner loop would cause excessive reconfiguration overhead); (2) the AGUs can start execution with a programmable delay, so that paths with different accumulated latencies can be synchronized; (3) one AGU can be started independently of the other AGUs, which may be at rest or running.

The third property is instrumental for exploiting TLP, which can be illustrated using datapath (b) in Fig. 4. Suppose one block of vector elements to be added are placed in memory M0, and that address generators M0-A, M0-B and M1-A are started right away (Thread 1). In parallel, one can move the next block to memory M2 and start AGUs M2-A, M2-B and M1-B (Thread 2). Then the Controller can monitor the completion of Thread 1 in order to restart it with a new vector block, and then monitor the completion of Thread 2 to also restart it with a new block. By alternately managing Thread 1 and Thread 2, vectors that largely exceed the capacity of the Versat memories can be processed in a continuous fashion.

2.2 Configuration Module

The set of configuration bits is organized in configuration spaces, one for each FU. Each configuration space may contain several configuration fields. All configuration fields are memory mapped from the Controller point of view. Thus, the Controller is able to change a single configuration field of a functional unit by writing to the respective address. This implements partial reconfiguration. Configuring a set of FUs results in a custom datapath for a particular computation.

The Configuration Module (CM) is illustrated in Fig. 5 with a reduced number of configuration spaces and fields for simplicity. It contains a variable length configuration register file, a configuration shadow register and a configuration memory. The configuration shadow register holds the current configuration of the DE,

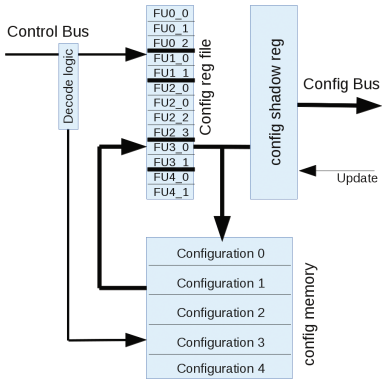

Fig. 5. Configuration module.

which is copied from the main configuration register whenever the Update signal is asserted. In this way, the configuration register can be changed while the DE is running. Figure 5 shows 5 configuration spaces, FU0 to FU4, where each FUj has configuration fields FUj_i. Note that, unlike what is suggested by the figure, the FUj_i fields do not have necessarily the same length (number of bits). A configuration memory that can hold 5 complete configurations is also shown. In the actual implementation the configuration word is 660 bits wide, there are 15 configuration spaces, 110 configuration fields in total and 64 configuration memory positions.

Still referring to Fig. 5, if the CM is being addressed by the Controller, the decode logic checks whether the configuration register file or the configuration memory is being addressed. The configuration register file accepts write requests and ignores read requests. The configuration memory interprets read and write requests as follows: a read request causes the addressed contents of the configuration memory to be read into the configuration register file; a write request causes the contents of the configuration register file to be stored into the addressed position of the configuration memory. This is a mechanism for saving and loading entire configurations in a single clock cycle with all configuration fields concatenated in a 660-bit word.

The CM has a special address that, when the Controller writes anything to it, all bits of the configuration register are cleared in one clock cycle. The default values of the configuration fields are coded with the value zero, so that this action restores the default configuration. Building a configuration of the DE from the default configuration is about 40% faster than writing all fields because many fields are left with their default values. The default values have been chosen so that they have a high likelihood of being used.

In most applications there is also a high likelihood that one configuration will be reused again, as is or with little modifications. Thus, it is useful to save certain configurations in the configuration memory to later load them and eventually tweak them.

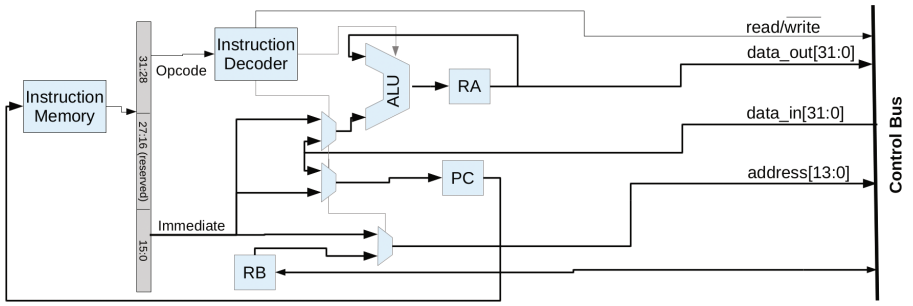

Fig. 6. Controller.

2.3 Controller

Versat uses of a minimal controller for reconfiguration, data transfer and simple algorithmic control. The instruction set contains just 16 instructions for the following actions: (1) loads/stores; (2) basic logic and arithmetic operations; (3) branching. Versat has an accumulator architecture with a 2-stage pipeline shown in Fig. 6. The controller architecture contains 3 main registers: the program counter (PC), the accumulator (RA) and the address register (RB), which is used in indirect loads and stores. There is only one instruction type as illustrated in the figure. The controller is the master of a simple bus called the Control Bus, whose signals are also explained in the figure.

The instruction set is outlined in Table 1. Brackets are used to represent memory pointers. For example, (Imm) represents the contents of the memory position whose address is Imm.

Table 1. Instruction set.

Mnemonic	Opcode	Description
nop	0x0	No operation; PC = PC+1
rdw	0x1	RA = (Imm); PC = PC+1
wrw	0x2	(Imm) = RA; PC = PC+1
rdwb	0x3	RA = (RB); PC = PC+1
wrwb	0x4	(RB) = RA; PC = PC+1
beqi	0x5	RA == 0? PC = Imm: PC = PC+1; RA = RA−1
beq	0x6	RA == 0? PC = (Imm): PC = PC+1; RA = RA−1
bneqi	0x7	RA != 0? PC = Imm: PC = PC+1; RA = RA−1
bneq	0x8	RA != 0? PC = (Imm): PC = PC+1; RA = RA−1
ldi	0x9	RA = Imm; PC = PC+1
ldih	0xA	RA[31:16] = Imm; PC = PC+1
shft	0xB	RA = Imm < 0? RA = RA << 1: RA=RA >> 1
add	0xC	RA = RA+(Imm); PC = PC+1
addi	0xD	RA = RA+Imm; PC = PC+1
sub	0xE	RA = RA−(Imm); PC = PC+1
and	0xF	RA = RA&(Imm); PC = PC+1

2.4 Qualitative Comparison with Other Architectures

Versat has some distinctive features which can not be found in other architectures: (1) it has a small number of processing elements (PEs) organized in a full mesh structure; (2) it has a fully addressable configuration register combined with a configuration memory to support partial configuration; (3) it has a dedicated controller for reconfiguration, DMA management and simple algorithm control – no RISC [3] or VLIW [4] processors are used.

CGRAs started as 1-D structures [7] but more recently square mesh 2-D PE arrays are more common [2–4]. However, the problem with square mesh topologies is that many PEs end up being used as routing resources, reducing the number of PEs available for computation and requiring sophisticated mapping algorithms [10]. Thus, we decided to use a rich interconnect structure and fewer PEs. As explained before, for a small number of PEs, the silicon area occupied by the full mesh interconnect is less than 5% and the limits placed in the frequency of operation are not as stringent as the ones imposed by the energy budgets of certain applications.

It is also important to keep the configuration time to a minimum. As explained in [2], the reconfiguration time in CGRAs can easily dominate the total execution time. To counter this effect we have decided to take partial reconfiguration to the extreme of using a fully addressable configuration register. This keeps the reconfiguration time to a minimum and contrasts with the more moderate hierarchical reconfiguration scheme proposed in [2].

Since it is crucial to have the reconfigurations done quickly, we have decided to include a small dedicated controller with just 16 instructions and low IO latency. It turned out that this controller also proved useful in managing data transfers and running the algorithms of interesting kernels such as the FFT kernel. In other architectures [2–4], more comprehensive processors are used. Our approach reduces the silicon area and power consumption of the core but also limits the complexity of the algorithms that can be run on it. Thus, we rely on other processors that exist in the system to run more complex algorithms, and we restrict Versat to be a kernel accelerator only.

3 Programming

The Versat controller can be programmed using a small C/C++ subset using the Versat compiler. Certain language constructs are interpreted as DE configurations and the compiler automatically generates instructions that write these configurations to the CM. The Versat controller can also be programmed in assembly language, given its easy to apprehend structure. To the best of our knowledge, Versat is the only CGRA that can be programmed in assembly. Despite its simplicity, the Versat controller is able to execute rather complex kernels autonomously.

The purpose of this paper is to describe the Versat architecture, not the Versat programming tools. However, after describing the Controller, it is useful to show an example program to illustrate the features of the C++ subset that

```
int main(){
    //initiate data transfer into Versat using DMA
    dma.config(1024, 256, 256, 1);
    dma.run();

    //clear configuration register and create new configuration
    de.clearConfig();
    for(j = 0; j < 128; j++)
        mem1A[j] = mem0A[j*2] + mem0B[1+j*2];

    //wait for data transfer to finish
    dma.wait();

    //run the data engine
    de.run();

    //configure DMA to transfer result back to memory
    dma.config(2048, 2048+128, 256, 2);

    //wait for data engine to finish
    de.wait(mem1A);

    //transfer result back to memory using DMA
    dma.run();
    dma.wait();
}
```

Fig. 7. Example code.

can be used. The chosen example is the interleaved vector addition program shown in Fig. 7. Comments are added to help understand the code.

Note that the Versat C/C++ dialect does not yet support object or variable declarations. All objects and variables that can be used are predefined. *For* loops with expressions involving memory ports are interpreted as DE configurations and trigger partial reconfigurations. Two nested loops and multiple expressions in the loop body are also supported. Whenever possible, the compiler keeps track of the current state of the configuration register to perform a minimal number of partial reconfigurations needed to prepare the next configuration. Partial reconfigurations generate store instructions in the assembly code, one per configuration field. In many practical situations, the configurations are generated in a program loop where, in each iteration, only a few configuration fields change. A configuration generation loop has a much smaller code footprint than storing all the configurations in memory.

4 Results

Versat has been designed using a UMC 130 nm process. Table 2 compares Versat with a state-of the-art embedded processor and two other CGRA implementations. The Versat frequency and power results have been obtained using the Cadence IC design tools. The power figures have been obtained using the node activity rate extracted from simulating an FFT kernel.

Table 2. Integrated circuit implementation results.

Core	Node (nm)	Area (mm^2)	RAM (kB)	Freq. (MHz)	Power (mW)
ARM Cortex A9 [11]	40	4.6	65.54	800	500
Morphosys [3]	350	168	6.14	100	7000
ADRES [4]	90	4	65.54	300	91
Versat	130	4.2	46.34	170	99

Because the different designs use different technology nodes, to compare the results in Table 2, we need to use a scaling method [12]. A standard scaling method is to assume that the area scales with the square of the feature size and that the power density (W/m^2) remains constant at constant frequency. Doing that, we conclude that Versat is the smallest and least power hungry of the CGRAs. If Versat were implemented in the 40 nm technology, it would occupy about 0.4 mm^2, and consume about 44 mW running at a frequency of 800 MHz. That is, Versat is 10× smaller and consumes about 11× lower power compared with the ARM processor.

The ADRES architecture is about twice the size of Versat. Morphosys is the biggest one, occupying half the size of the ARM processor. These differences can be explained by the different capabilities of these cores. While Versat has a 16-instruction controller and 11 FUs (excluding the memory units), ADRES has a VLIW processor and a 4x4 FU array, and Morphosys has a RISC processor and an 8x8 FU array.

A prototype has been built using a Xilinx Zynq 7010 FPGA, which features a dual-core embedded ARM Cortex A9 system. Versat is connected as a peripheral of the ARM cores using its AXI4 slave interface. The ARM and Versat cores are connected to an on-chip memory controller using their AXI master interfaces. The memory controller is connected to an off-chip DDR module. This FPGA prototype has only been used to measure the execution time in clock cycles. The performance and energy estimates discussed in the next paragraph have been obtained using the measured execution times combined with frequency and power estimates extrapolated from the results in Table 2.

Results for a set of kernels are summarized in Table 3. For both ARM and Versat, the program has been placed in on-chip memory and the data in an external DDR memory device. The Versat *Total* cycle counts include data transfer, processing, control and reconfiguration. The Versat *Unhidden* cycle counts

means the control and reconfiguration cycles that do not occur in parallel with the DE or DMA. The average number of FUs used and the code size are given for each kernel. The speedup and energy ratio have been obtained assuming the ARM is running at 800 MHz and Versat is running at 600 MHz in the 40 nm technology. The energy ratio is the ratio between the energy spent by the ARM processor alone and the energy spent by an ARM/Versat combined system using the power figures in Table 2.

Table 3. Kernel benchmark results.

Kernel	ARM Cortex A9 cycles	Versat cycles		Versat FUs used	Versat code size (bytes)	Speedup	Energy ratio
		Total	Unhidden				
`vec_add`	14726	4517	36	3	152	2.45	2.29
`iir1`	18890	7487	26	5	220	1.89	1.77
`iir2`	24488	10567	26	8	332	1.74	1.62
`cip`	25024	6673	26	10	408	2.81	2.63
`fft`	394334	16705	624	8.5	3028	17.70	16.55

In Table 3, `vec_add` is a vector addition, `iir1` and `iir2` are 1st and 2nd order IIR filters, `cip` is a complex vector inner product and `fft` is a Fast Fourier Transform. This kernel set occupies only 50% of the 8 kB program memory. All kernels operate on Q1.31 fixed-point data with vector sizes of 1024. The first 4 kernels use a single Versat configuration and the data transfer size dominates. For example, the `vec_add` kernel processing time is only 1090 cycles and the remaining 3427 cycles account for data transfer and control. The FFT kernel is more complex and goes through 43 Versat configurations generated on the fly by the Versat controller. The processing time is 12115 cycles and the remaining 4590 cycles is for data transfer and control. It should be noted that most of the reconfiguration and control is done while the data engine is running. In fact, only 605 cycles are unhidden reconfiguration and control cycles in the FFT kernel. In general, this is true for all kernels: unhidden reconfiguration and control cycles are about 1–5% of the total time. The number of FUs used is low for simple kernels like `vec_add` and `iir1` (which could have used a smaller array) but is over 50% for more complex kernels. However, the simpler kernels could have been designed in a more parallel fashion, using more FUs. For example, `vec_add` can use multiple adders in parallel but only one has been instantiated. These results show good performance speedups and energy savings, even for single configuration kernels.

Most of the 43 FFT configurations derive from two basic configurations that are partially changed many times. The two configurations implement a radix-2 FFT butterfly: one configuration performs the complex product and the other configuration performs the two complex additions in a butterfly. The variations of these configurations alternate the source and destination memories for the data (ping-pong processing), and the address generation constraints to read and write the data values for the various stages and blocks of the FFT. On average only 26% of the array is reconfigured each time. When the array is being configured from

the default values, on average 68% of the bits need be written. There is also one configuration to copy the FFT coefficients between two memories (so that they can be accessed from 4 memory ports simultaneously) and to reorder the data by bit reversing the addresses. Partial reconfiguration plays a key role in the FFT example: with full reconfiguration, the execution time grows about 7%, as there are many short loops that are not long enough to overlap with reconfiguration. If full reconfiguration, with all configurations produced at compile time, were used, like in [2–4], the FFT code would be 2.2× larger, penalizing memory bandwidth and capacity.

We can compare Versat with Morphosys since it is reported in [13] that the processing time for a 1024-point FFT is 2613 cycles. Compared with the 12115 cycles taken by Versat, this means that Morphosys was 4.6× faster. This is not surprising since Morphosys has 64 FUs compared to 11 FUs in Versat. However, our point is that an increased area and power consumption is not justified when the CGRA is integrated in a real system. Note that, if scaled to the same technology, Morphosys would be 5× the size of Versat. Unfortunately, comparisons with the ADRES architecture have not been possible, since we have not found any cycle counts published, despite ADRES being one of the most cited CGRA architectures.

It would be nice if we implemented the other approaches in our setup to have a fairer comparison, instead of using published results. However, those are complex cores and implementing them ourselves, besides representing a formidable effort, would carry the risk of us missing some important details that could distort the results. It would also be nice to study Versat's performance in a real application, where it has to be reconfigured periodically to different kernels. We leave that for a more mature state of our evaluation.

5 Conclusion

In this paper we have presented Versat, a minimal CGRA with 4 embedded memories and 11 FUs, a fine partial reconfiguration scheme and a basic 16-instruction controller.

Versat has fewer processing elements compared to other CGRAs (eg. RPU [2]) but uses a full mesh interconnect topology. Another unique feature is a fully addressable configuration register combined with a configuration memory to support partial configuration. The simple Versat controller is used for reconfiguration, DMA management and simple algorithm control, dispensing with complex RISC or VLIW processors proposed in other approaches. The controller generates Versat configurations on the fly, instead of using pre-compiled configurations like other CGRAs. This saves configuration storage space and memory bandwidth. Versat can be programmed in a C/C++ dialect and can be used by host processors by means of an API containing a set of useful kernels.

Results on a VLSI implementation show that Versat is competitive in terms of silicon area (2× smaller than ADRES [4]), and energy consumption (3.6× lower compared with Morphosys [3]). Performance results show that a system

combining a state-of-the-art embedded processor and the Versat core can be 17× faster and more energy efficient than the embedded processor alone when running the FFT algorithm.

Acknowledgment. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

References

1. 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, Heidelberg (2010)
2. Liu, L., Wang, D., Zhu, M., Wang, Y., Yin, S., Cao, P., Yang, J., Wei, S.: An energy-efficient coarse-grained reconfigurable processing unit for multiple-standard video decoding. IEEE Trans. Multimed. **17**(10), 1706–1720 (2015)
3. Lee, M.H., Singh, H., Lu, G., Bagherzadeh, N., Kurdahi, F.J.: Design and implementation of the MorphoSys reconfigurable computing processor. J. VLSI Signal Process. Syst. Signal Image Video Technol. **24**, 147–164. Kluwer Academic Publishers (2000)
4. 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)
5. de Sousa, J.T., Martins, V.M.G., Lourenco, N.C.C., Santos, A.M.D., do Rosario Ribeiro, N.G.: Reconfigurable coprocessor architecture template for nested loops and programming tool. US Patent 8,276,120 (2012)
6. 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). doi:10.1007/978-3-540-48302-1_42
7. 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). doi:10.1007/3-540-61730-2_13
8. 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)
9. Farahini, N., Hemani, A., Sohofi, H., Jafri, S.M.A.H., Tajammul, M.A., Paul, K.: Parallel distributed scalable runtime address generation scheme for a coarse grain reconfigurable computation and storage fabric. Microprocess. Microsyst. **38**(8), 788–802 (2014)
10. Liu, D., Yin, S., Liu, L., Wei, S.: Polyhedral model based mapping optimization of loop nests for CGRAs. In: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–8 (2013)
11. Wang, W., Dey, T.: A survey on ARM Cortex A processors. http://www.cs.virginia.edu/skadron/cs8535s11/armcortex.pdf. Accessed 6 Apr 2016
12. Huang, W., Rajamani, K., Stan, M.R., Skadron, K.: Scaling with design constraints: predicting the future of big chips. IEEE Micro **31**(4), 16–29 (2011)
13. Kamalizad, A.H., Pan, C., Bagherzadeh, N.: Fast parallel FFT on a reconfigurable computation platform. In: 15th Symposium on Computer Architecture and High Performance Computing, Proceedings, pp. 254–259 (2003)