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. This stems from the observation that competitive acceleration can be achieved with a smaller array and more frequent 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 these reasons CGRA architectures normally feature a processor core. For example, the Morphosys architecture [2] uses a RISC processor and the ADRES architecture [3] uses a VLIW processor.

This work started with two 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. Examples of target kernels are transforms (IDCT, FFT, etc), filter banks (FIR, IIR, etc), and others.

Therefore we propose a new architecture, Versat, which uses a relatively small number of functional units and a simpler controller. A smaller array limits the size of the data expressions that can be mapped to the CGRA but these expressions can be broken into smaller expressions which can be executed sequentially in the CGRA. Therefore Versat requires mechanisms for handling large numbers of configurations and frequent reconfigurations efficiently.

Versat is to be used as a co-processor featuring an API containing a set of useful kernels. Applications developers can use a commercial embedded processor with a rich ecosystem and drop in a Versat core for performance and power optimization. Versat programmers can create a set of useful kernels that application programmers will want to use. In this way, the software and programming tools of the CGRA are clearly separated from those of the application processor. This makes Versat suitable for supporting the Open Computing Language (OpenCL) standard or others.

A new compiler for Versat has been developed. The use of standard compilers such as *gcc* or *llvm* has been investigated. However, classical compilers are good at producing sequences of instructions, not sequences of hardware datapaths. For this reason, it has been decided that a specific compiler needed to be 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 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, 2, 4] 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 [5] and PACT [6]. RaPiD supports dynamic (cycle by cycle) partial reconfiguration for a subset of the configuration bitstream, which suggests that the loop body may take several cycles to execute. The reconfiguration process in PACT is reportedly slow and users are recommended to avoid it and resort to full reconfiguration whenever possible. We do not have data to make performance comparisons with these approaches, but, compared to [5], our partial reconfiguration happens between program loops instead of cycle by cycle, and the loop body executes in only one cycle. Compared to [6], our partial reconfiguration is fast and is used frequently.

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 per-

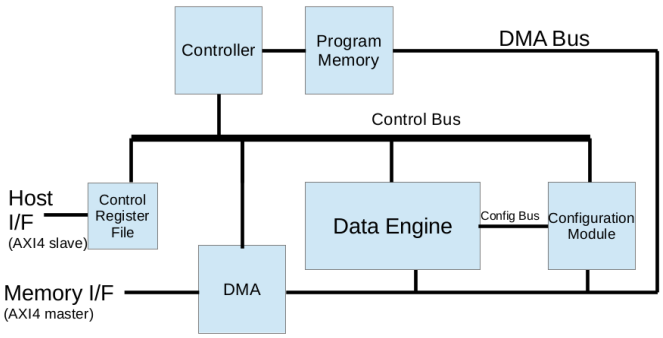

Fig. 1. Versat top-level entity

form 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. 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 load and execute programs. The host and the Versat controller communicate using the Control Register File. 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 embedded memories, 4 multipliers, 6 arithmetic and logic units. The Versat controller can read and write the output register of the FUs and can read and write to the embedded memories.

Each FU contributes its 32-bit output(s) to a wide Data Bus of 19x32 bits, and is able to select one 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. For example, an ALU can be configured to perform addition, subtraction, logical AND, maximum and minimum, etc. Unfortunately, there is no space in this paper to outline all the operations of each FU.

Therefore, there are direct connections from any FU to any other FU. This complete interconnect structure may be unnecessary but it greatly simplifies the compiler design as it avoids expensive place and route algorithms. More compact interconnect may be developed in the future simultaneously with compiler

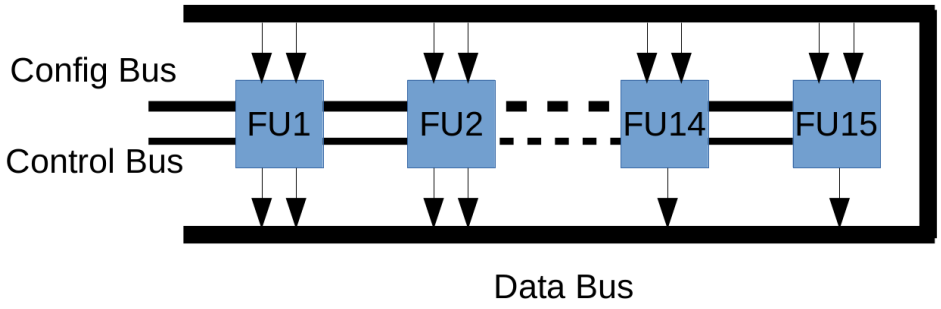

Fig. 2. Data engine

improvements. In any case, the interconnect does not consume much power since Versat is reconfigured only after a complete program loop is executed in the DE.

Each configuration of the DE corresponds to a hardware datapath. 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 Versat.

Each memory port is equipped with an address generator to access data from the embedded memories during the execution of a program loop. The discussion of the details of the address generator falls out of the scope of this paper. We will simply state the following properties: (1) two levels of nested loops are supported (reconfiguration after each inner loop would cause excessive reconfiguration overhead); (2) an address generator can start execution with a programmable delay, so that paths with different accumulated latencies can be synchronized.

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. 3, 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, 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. Fig. 3 shows 4 configuration spaces, FU0 to FU3, where each FUj has configuration fields FUj_i of varying lengths. A configuration memory

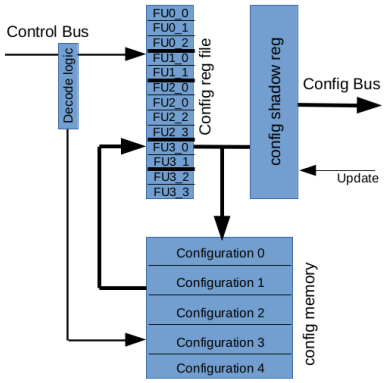

Fig. 3. Configuration module

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.

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.

Building a configuration of the DE for the first time requires several writes to the fields of the configuration spaces of the involved FUs. In most applications there is a high likelihood that one configuration will be reused again. It is also likely that other configurations will differ little from the current configuration. Thus, it is useful to save certain configurations in the configuration memory to later load them. A loaded configuration may be used as is or partially changed.

2.3 Controller

Versat uses of a minimal controller for reconfiguration, data transfer and simple algorithmic control. The instruction set contains just 16 instructions used to perform the following actions: (1) loads/stores; (2) basic logic and arithmetic operations; (3) branching.

The Versat controller can be programmed using a 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.

3 Results

Versat has been designed using a UMC 130nm process. Table 1 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, and the node activity rate extracted from simulating an FFT kernel.

Table 1. Implementation results

Core	Node(nm)	Area(mm^2)	RAM(KB)	Freq.(MHz)	Power(mW)
ARM Cortex A9 [7]	40	4.6	65.54	800	500
Morphosys [2]	350	168	6.14	100	7000
ADRES [3]	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 1, we need to use a scaling method [8]. A standard scaling method is to assume that the area scales with the square of the feature size and that the power density 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 40nm technology, it would occupy about 0.4 mm^2, and consume about 44mW running at a frequency of 800MHz. That is, Versat is 10x smaller and consumes about 11x less 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 core and Versat are connected to an on-chip memory controller using their AXI master interfaces. The memory controller is connected to an off-chip DDR module.

Results on running a set of kernels on Versat and on the ARM Cortex A9 are summarized in Table 2. For both the ARM and Versat, the program has been

placed in on-chip memory and the data in an external DDR memory device. Cycle counts include processing, reconfiguration and data transfer times. The speedup and energy ratio have been obtained assuming the ARM is running at 800 MHz and Versat is running at 600MHz in the 40nm 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 1.

Table 2. Cycle counts, speedup and energy ratio

Kernel	ARM Cortex A9 cycles	Versat cycles	Speedup	Energy Ratio
`vec_add`	14726	4517	2.45	2.29
`iir1`	18890	7487	1.89	1.77
`iir2`	24488	10567	1.74	1.62
`cip`	25024	6673	2.81	2.63
`fft`	394334	16705	17.70	16.55

In Table 2, `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. All kernels operate on Q1.31 fixed-point data with a 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 control is done while the data engine is running. In fact only 638 cycles are unhidden control cycles in the FFT kernel. These results show good performance speedups and energy savings, even for single configuration kernels.

We can compare Versat with Morphosys since it is reported in [9] 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 4.6x faster. This is not surprising since Morphosys has 64 FUs compared to 11 FUs in Versat. However, our point is whether an increased area and power consumption is justified when the CGRA is integrated in a real system. Note that, if scaled to the same technology, Morphosys would be 5x 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 published CGRA architectures.

4 Conclusion

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

other CGRAs with larger arrays, Versat requires more configurations per kernel and a more sophisticated reconfiguration mechanism. Thus, the Versat controller can generate configurations and uses partial reconfiguration whenever possible. The controller is also in charge of data transfers and basic algorithmic flows.

Versat can be programmed in a C++ dialect and is suitable to be used by a host processor by means of a standard interface such as OpenCL.

Results on a VLSI implementation show that Versat is competitive in terms of silicon area, frequency of operation and power consumption. Performance results show that a system combining a state-of-the-art embedded processor and the Versat core can be 17x faster and more energy efficient than the embedded processor alone.

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. Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. Coarse-grained reconfigurable array architectures. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 449–484. Springer US, 2010.
2. Ming hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, and Fadi J. Kurdahi. Design and implementation of the MorphoSys reconfigurable computing processor. In *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology*. Kluwer Academic Publishers, 2000.
3. Bingfeng Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins. Architecture exploration for a reconfigurable architecture template. *Design & Test of Computers, IEEE*, 22(2):90–101, March 2005.
4. R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Mapping applications onto reconfigurable Kressarrays. In Patrick Lysaght, James Irvine, and Reiner Hartenstein, editors, *Field Programmable Logic and Applications*, volume 1673 of *Lecture Notes in Computer Science*, pages 385–390. Springer Berlin Heidelberg, 1999.
5. Carl Ebeling, Darren C. Cronquist, and Paul Franklin. Rapid - reconfigurable pipelined datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, FPL '96, pages 126–135, London, UK, 1996. Springer-Verlag.
6. V. Baumgarte, G. Ehlers, F. May, A. Nckel, M. Vorbach, and M. Weinhardt. PACT XPP – a self-reconfigurable data processing architecture. *The Journal of Supercomputing*, 26(2):167–184, 2003.
7. Wei Wang and Tanima Dey. A survey on ARM Cortex A processors. http://www.cs.virginia.edu/skadron/cs8535s11/armcortex.pdf. Accessed 2016-04-16.
8. W. Huang, K. Rajamani, M. R. Stan, and K. Skadron. Scaling with design constraints: Predicting the future of big chips. *IEEE Micro*, 31(4):16–29, July 2011.
9. A. H. Kamalizad, C. Pan, and N. Bagherzadeh. Fast parallel FFT on a reconfigurable computation platform. In *Computer Architecture and High Performance Computing, 2003. Proceedings. 15th Symposium on*, pages 254–259, Nov 2003.