TÉCNICO LISBOA

VERSAT, a Compile-Friendly Reconfigurable Processor - Architecture

João Dias Lopes

Introduction to the Research in

Electrical and Computer Engineering

Supervisor: Prof. José João Henriques Teixeira de Sousa

Examination Committee

Chairperson: Prof. Pedro Filipe Zeferino Tomás
Supervisor: Prof. José João Henriques Teixeira de Sousa

June 2016

Abstract

This report introduces Versat, a reconfigurable hardware accelerator to be used in an embedded system in order to optimize performance and power. Versat is a Coarse-Grain Reconfigurable Array architecture (CGRA), which implements self and partial reconfiguration by using a simple controller unit. Compared to other CGRAs, Versat has a smaller number of functional units with a fully connected graph topology for maximum flexibility. The idea is to operate in a region of the design space, where less computations can be mapped onto the array but the array itself is more frequently reconfigured. An original feature of Versat is the ability to map sequences of nested program loops instead of just one program loop. Reconfiguration happens when moving from one nested loop to the next. Experimental results are presented.

Keywords: reconfigurable computing, coarse-grain reconfigurable arrays, embedded systems

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the expansion in the number of features, modern embedded systems are becoming more and more power hungry. Therefore it is crucial to minimize power consumption. The main reason for this low power demand is the short battery life of most devices used daily.

Another problem is the device price. The smaller the circuit the cheaper it will be, and higher profit margins can be obtained from it. Therefore, if one can deliver the same functionality with a smaller and more power efficient device, one will have a more competitive product.

These problems have been tackled with smaller and smaller transistors which can work at higher frequencies without changing the architecture. However, with the eminent demise of Moore's law and the advent of the Internet of Things (IoT), this solution is not viable. Reconfigurable hardware, which is known to be extremely efficient in terms of power consumption and silicon area utilization, has been used in mid-range to high-end applications. Fine grain reconfigurable fabrics such as Field-Programmable Gate Arrays (FPGAs) can be an alternative but, compared to Coarse-Grain Reconfigurable Arrays (CGRAs), embedded FPGA cores consume significantly more silicon area and power.

A CGRA is a collection of programmable functional units and embedded memories connected by programmable interconnects. This structure is what is called the reconfigurable array. When given a set of configuration bits, the reconfigurable array forms a hardware datapath able to execute a certain task orders of magnitude faster than a conventional Central Processing Unit (CPU). CGRAs are used as hardware co-processors to accelerate algorithms that are time/power consuming in regular CPUs.

Normally, the reconfigurable array is used only to accelerate program loops and the non-loop code is run on an attached processor which has a conventional architecture. For this reason, CGRAs normally include a conventional processor. For example, the Morphosys architecture [1] integrates a small Reduced Instruction Set Computer (RISC) and the ADRES architecture [2] integrates a Very Large Instruction Word (VLIW) processor.

1.2 Problems addressed

The main problems that we have identified with existing CGRAs are their large size, limited reconfiguration control and difficulty in programming. Therefore, we propose some architectural improvements to address these problems which follow three basic ideas.

The first idea is to make the CGRA smaller and to use fully connected graph topology. Normally, graphs with constrained connectivity are employed in CGRAs to avoid decreasing the frequency of operation. However, low power devices will rarely choose to operate at a high frequency, so using a fully connected graph becomes a possibility. In terms of silicon area, fewer compute nodes can be used if all nodes are connected to one another but more routing resources are needed. Fortunately, these routing resources do not increase power consumption as our reconfiguration frequency is low. In fact, our reconfiguration process is more frequent compared to static arrays such as [3], but much less frequent compared to dynamic arrays such as [2]. Although we cannot build large datapaths with many functional units, because we use fewer, we can build *any* datapath with the functional units we do have. Finally, programming is drastically simplified as there is no need for *place & route* algorithms in the compilation flow such as in FPGAs and other fabrics.

The second idea is to make the configuration register addressable to the word level. The configuration is divided in spaces, which correspond to each functional unit, and the configuration spaces are further divided in configuration fields which are made individually addressable. Partial reconfiguration is useful to keep reconfiguration to a minimum, exploiting the similarity between successive configurations. Reducing reconfiguration time has a dramatic influence on improving the performance, which can compensate for a lower frequency of operation.

The third idea is to integrate a specialized and programmable controller in the CGRA to manage reconfigurations and data transfers to/from external memory. The controller is in charge of the main program flow of the accelerator, sequencing the configurations and using partial reconfiguration whenever possible. The controller can spawn data stream compute threads in the CGRA and data movement threads using a Direct Memory Access (DMA) unit. While these threads are running, the controller can prepare the next configurations.

1.3 Topic Overview

In this work we presente Versat, a new reconfigurable hardware accelerator which is suitable for low-cost low-power devices. Its architecture uses a relatively small number of functional units and a simple controller. A smaller array limits the size of the data expressions that can be mapped to the CGRA but large 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 Application Programming Interface (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.

1.4 Objectives

The main objective of this new architecture is to get acceleration with low power consumption and small silicon area.

In fact, Versat can replace a number of dedicated hardware accelerators in a System on Chip (SoC), making it smaller, more power efficient and safer to design (the development risk of designing dedicated hardware accelerators is eliminated).

Digital signal processing applications are targeted: biometrics, speech recognition, artificial vision, security, etc. The overall goal of the project is to create an Intellectual Property (IP) core and a library of useful procedures. A clean procedural interface to a host becomes possible. With such an interface the host can have tasks executed in the CGRA by simply calling procedures and passing arguments.

1.5 Author's Work

The work presented here is the result of the work of a few people. When I started working on this project, in the summer of 2014, there was a preliminary, non-functional version of the system. I contributed the multi-threading idea, where the state machines of the address generators work independently, I pipelined parts of the implementation to improve the frequency of operation, I wrote the boot Read Only Memory (ROM) software and its hand shaking protocol with a host system. I also implemented the DMA design, master and slave Advanced Extensible Interface (AXI) interfaces, contributed to the Application-Specific Integrated Circuit (ASIC) implementation, and, finally, I wrote all assembly kernels used for tests and implemented a regression test system where it is easy to add or remove new tests. As a result of this work, the team has already a paper accepted for publication [4].

1.6 Report Outline

This report is composed by a few more chapters. In the second chapter, the advantages and disadvantages of other architectures are discussed. In the third chapter, Versat's architecture is fully described. In the fourth chapter, some experimental results are presented. In the fifth and final chapter, our achievements are pointed out and some future work is outlined.

Chapter 2

Background

CGRAs have gained increasing attention in the last 2 decades both in academia and industry [2, 1, 5, 6, 7]. CGRAs are programmable hardware mesh structures that operate on word-length variables and are efficient in terms of operations per unit of silicon area. CGRAs can be built with simple components such as adders, subtractors, multipliers, shifters, among others [8, 7]. CGRAs are a suitable architecture for a vast range of low power devices.

2.1 Important problems

A critical aspect for achieving performance speedups is dynamic reconfiguration of the CGRA. Static reconfiguration, where the array is configured once to run a complete kernel, has poor flexibility [9]. Some arrays are dynamically reconfigurable but they only iterate over a fixed sequence of configurations [6, 2, 1]. These configurations must be moved to the CGRA from external memory and stored inside the CGRA, which costs memory bandwidth and storage space in the CGRA. Following a fixed sequence of configurations means that it is impossible to conditionally run a configurations. This limits the programming flexibility of the CGRA.

Another important problem is that of the interconnect topology [10]. Fully connected graph topologies have been avoided as they scale poorly in terms of area, wire delays and power consumption. Since large arrays have been preferred over smaller arrays, it has been important to keep a lean interconnection structure. However, in this work a relatively smaller array is used and the impact of a fully connected topology is assessed in terms of silicon area, power and programming flexibility.

2.2 Reconfiguration

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 [2, 1, 3] 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 [11] and PACT [5]. 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 [11], our partial reconfiguration happens between program loops instead of cycle by cycle, and the loop body executes in only one cycle. Compared to [5], our partial reconfiguration is fast and is used frequently.

2.3 Address Generation

The main contribution in [7] was the invention of an address generation scheme able to support groups of nested loops in a single machine configuration. The idea, aimed at reducing reconfiguration time, was inspired by the use of cascaded counters for address generation [12]. This represented a major improvement from other works that focus exclusively in supporting the inner loops of compute kernels [13]. However, as this report shows, more can be done in terms of address generation to reduce the reconfiguration overhead.

2.4 Heterogeneity versus Homogeneity

The question of heterogeneity versus homogeneity of the functional units inside a CGRA is an important one. Some CGRAs are homogeneous [11], i.e., they only have one type of functional unit, whereas others are heterogeneous and support a diversity of functional units [14]. A careful analysis in [15] has favored heterogeneous CGRAs as the performance degradation when going from homogeneous to heterogeneous is greatly compensated by the silicon utilization rate and power efficiency of heterogeneous solutions. Thus, we adopt heterogeneous CGRAs in this project.

2.5 Compiler

Compiler support for CGRAs is probably the most difficult aspect. Not only a compiler has to make use of standard compilation techniques, especially the well known modulo scheduling approach used in VLIW machines [16], but also Computer-Aided Design (CAD) techniques are needed, such as those used in FPGA compilation [17]. One attempt to circumvent the compiler difficulties is to formulate CGRAs as vector processors [18, 19, 20]. In those approaches, instructions are the equivalent of small configurations, and their authors claim several orders of magnitude speedup in certain applications. However, the user has to work at a very low level to make use of vector instructions. We propose as a solution to this problem the adoption of programming interfaces such as Open Computing Language (OpenCL) [21], now very popular for Graphics Processing Units (GPUs) and FPGAs in heterogeneous computing environments.

A new compiler for Versat has been developed in parallel with this work. 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 its functionality has been restricted 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 report whose main thrust is the description of the architecture and Very-Large-Scale Integration (VLSI) implementation.

Chapter 3

Architecture

Versat is designed to do signal processing, so its architecture has a Data Engine (DE) unit that performs data computation. Since Functional Units (FUs) are fully connected, that means there is more than one way to make a datapath. This feature has been implemented with the objective of simplifying the compiler technology, which can be very complex. A configuration module holds the configuration of the DE, i.e., it specifies the current datapath, and can also temporarily store tens of other configurations, which can be switched at runtime. The Versat top-level entity is represented in figure 3.1.

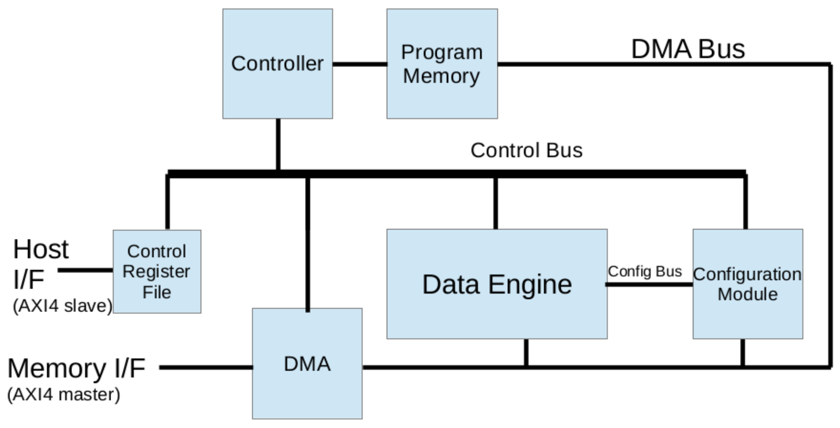

Figure 3.1: Versat top-level entity

In order to have less dependency from the host CPU, it has been decided that Versat should have a simple controller. This way, it can do self-reconfiguration without engaging the host, which becomes free for other more useful tasks. The controller is programmable and has an instruction memory where Versat programs or kernels are stored. To communicate with the host processor, Versat has the Control Register File (CRF), which is shared with the host.

The controller can write partial configurations to the configuration module, command the configuration module to save a configuration or to restore a saved configuration.

Versat as two interfaces that can be selected at compile time: a Serial Peripheral Interface (SPI) and a parallel bus interface. The SPI interface is used when an off-chip device is the host. Versat is a slave SPI device and the host is a master SPI device. The parallel bus interface is used when the host is some embedded processor. This bus had a generic format which has been recently replaced with the Advanced Extensible Interface - AXI4. It is an interface designed by ARM, which derives from the Advanced Microcontroller Bus Architecture (AMBA). Associated with the parallel interface is a DMA module used for data transfers from/to an external memory.

3.1 Data Engine

The Data Engine (DE) is designed to do vector computations. To accomplish this goal, instead of regular registers, it has 4 Random Access Memories (RAMs - with the size of 2048 words of 32 bits), which work as vector registers.

It also contains several FUs that actually perform data computation: 2 ALUs, 4 smaller ALUs (AluLite), 4 multipliers and 1 barrel shifter. The FUs are connected to each other by a large bus, formed by the output of every FU – the data bus. Each FU input can then select a section of this bus by using multiplexers (MUXs). A simple diagram in figure 3.2 illustrates the DE.

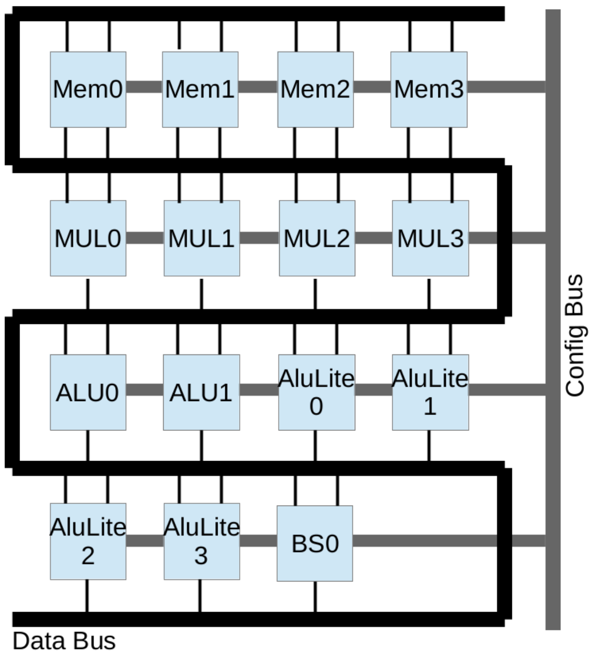

Figure 3.2: Data engine

The data bus, in addition to the output from the FUs, has 2 additional values: the constants 0 and 1, which are commonly used in many datapaths, so there is no need to store these constants. Another bus is used to configure the DE to create datapaths: the configuration bus.

An ALU can do any of the following operations:

- logical OR;
- logical AND;
- logical AND with one of the operands negated;

- logical XOR;
- addition;
- subtraction;
- sign extension of 8 bits;
- sign extension of 16 bits;
- arithmetic right shift;
- logic right shift;
- unsigned comparison;
- signed comparison;
- counting leading zeros;
- maximum;
- minimum;
- absolute value.

The AluLite units can perform only the first six operations. This way, the AluLite units are smaller and more power efficient. Both ALU types have 2 clock cycles of latency.

The multiplier produces a 64-bit result from two 32-bit operands and has two configuration parameters. One parameter allows selecting the lower or higher 32 bits of the result. The other parameter forces the multiply result to be left-shifted by 1 bit. This configuration is useful when operands are in the common Q1.31 fixed-point format. Setting the first parameter to select the high part of the result and the second parameter to shift left by 1, allows the multiplication of two Q1.31 operands to yield Q1.31 result. This unit has 3 clock cycles latency.

The barrel shifter can perform left shifts and logic and arithmetic right shifts. The number of bits to shift is one operand and the value to be shifted is the other operand. This FU has only 1 clock cycle of latency.

All FUs have the possibility to store some value in their output register to be used in the datapath. To do that it is required that the FU is configured as disabled. This is useful in case the computation datapath has one or more constants.

The DE memories are dual-port RAMs and each port has attached an address generator. The address generator is the block that allows Versat to execute two nested loops with just one configuration, with the restriction that the inner loop has a maximum of 32 iterations and the outer loop has a maximum of 2048 iterations.

3.2 Configuration Module

The configuration module is composed by a configuration register, where the next configuration is set, and the configuration memory, where some configurations are stored to be used later, as it is visible in figure 3.3.

The configuration register is a register of 660 bits, fully addressable, which means that it is possible to change only one field of this register. This feature takes advantage that in most applications there is a high likelihood that one configuration will be reused again in a nearby instant of time (time locality). Therefore, it is possible to do partial reconfiguration in a few clock cycles.

The configuration module also has a shadow configuration register. This register holds the configuration that the DE is executing, so another one can be generated in parallel with the DE execution.

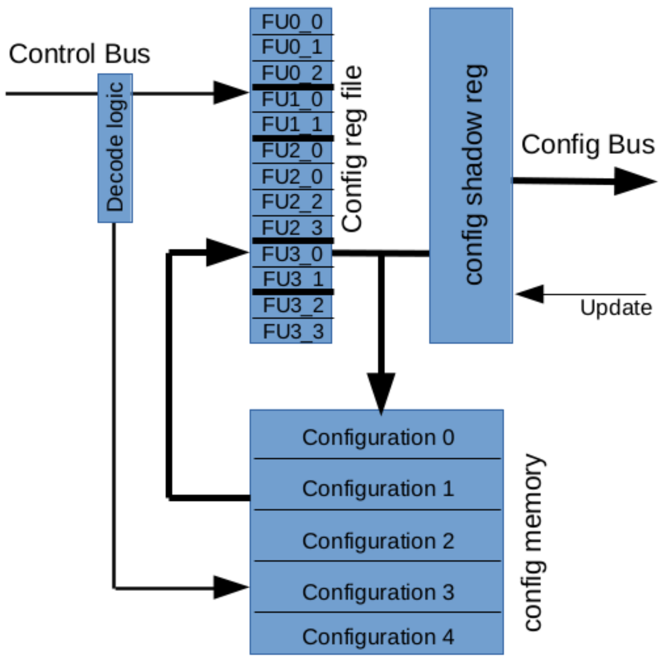

Figure 3.3: Configuration Module

The configuration memory has 64 positions, used to keep some configurations previously generated to reuse them later. This memory is a dual-port memory. One port has 660 bits (configuration width), which means that configurations can be loaded/stored from/to the configuration register in just 1 clock cycle. The other is a 32-bit port used to load and store configurations in the external memory through the DMA. This second port can expand the configuration memory beyond 64 configurations. This scheme is designed so that one can study the difference between working with pre-built configurations stored in external memory and generating configurations using the Versat controller.

3.3 Controller

The Versat controller has a minimal architecture (Fig. 3.4) to support reconfiguration, data movement and host interaction. It contains 3 main registers: the program counter (register PC), the accumulator (register A) and the data pointer (register B). The instruction whose address is pointed by the PC is decoded so that an opcode and an immediate value (often a memory address) are extracted from it.

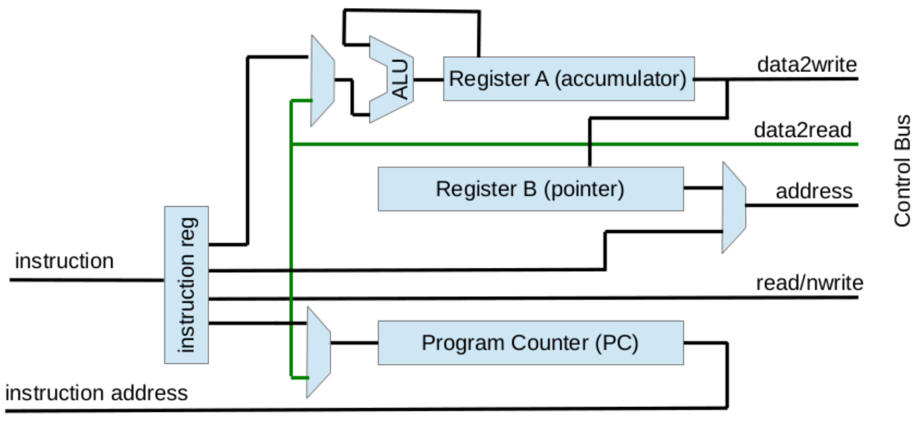

Figure 3.4: Controller

Register A is the destination of all operations that the controller performs, and is also often one of the operands (accumulator architecture).

On the other hand, register B, which is addressable by the controller, is used to implement indirect loads and stores. Its contents is the load/store address.

The controller has an instruction set of only 16 instructions (opcode of 4 bits, immediate value of 16 bits). These allow the controller to perform the following actions: loads/stores to/from the accumulator, arithmetic and logic operations and branches. There are three types of load instructions: of immediate constants, direct (from an immediate address) and from an indirect address stored in register B. The stores can be direct or indirect.

In order to increase frequency, by reducing the critical path, the controller takes 2 clock cycles to fetch one instruction, in pipeline. For simplicity, it executes every instruction that is fetched. In other words, it has 2 delay slots in case of a branch instruction. These delay slots can be filled with a no-operation instruction (NOP), but the compiler/programmer can also use them to execute useful instructions. For instance, in the case of a `for` loop, the delay slots can be used to write the iteration count to some register.

The controller can handle host procedure calls. For each procedure it is necessary to read parameters from the control register file and configure and execute the DMA and DE multiple times. DMA and DE threads can be spawned, hiding part of the controller execution time, as will be shown later.

3.4 DMA

One of the crucial factors to guarantee acceleration is the rapidity at which data is moved in and out of Versat. Accessing data words one by one in the external memory is out of the question. Data must be moved in blocks to amortize the memory latency that is always present when an external memory device has to be used.

The DMA engine is operated by the Versat controller to transfer a data burst from external memory to one of the Versat's data engine memories, or from one of Versat's data engine memories to the external memory. It also possible to move data to the instruction memory or to/from the configuration memory.

From the Versat controller point of view the DMA is memory mapped and the following DMA registers can be accessed: the external address register, the internal address register, the size register, the direction register and the status register. The external address register holds the transfer start address in the external memory (32 bits), and the internal address register holds the transfer starts address in Versat (14 bits). The size register has the number of words to be transferred (8 bits to support transactions of up to 256 words, which is the maximum for AXI4). The direction register tells the DMA state machine if it is a transfer form Versat to the external memory or vice-versa. When this register is written with a value different from zero the transfer starts. At the end of the transfer it self clears. The status register tells the controller whether the DMA is busy or ready to initiate a new transaction; it also informs if an error has occurred.

3.5 Program memory

The program memory is divided in two parts: the boot ROM (256 words of 32 bits) and the instruction memory RAM (2048 words of 32 bits). However, it is addressable as a single memory with the value stored in the PC register. The first 256 addresses are used to address the boot ROM, the others to address the instruction memory.

The boot ROM holds the instructions that allow Versat to communicate with the host processor. Basically is is used for the host to call Versat kernels and to load/store values in any addressable memory position within Versat.

The instruction memory, that needs to be loaded, contains the instructions related with the kernel. This memory can not be read (except to execute instructuions) but it can be written to load the kernel instructions.

3.6 Control Register File

The CRF has 16 registers of 32 bits each, and is implemented with a dual-port register file (one port for the host and the other port for Versat). These registers are used to establish the communication between the two processors. To perform that, both can read and write to the CRF.

Chapter 4

Results

This chapter presents results on implementing Versat on FPGA, as well as on ASIC technology. Performance results comparing Versat to an FPGA embedded processor (Microblaze, from Xilinx) and to an ARM Cortex A9 system are also presented.

4.1 FPGA implementation results

Versat's FPGA implementation results are given in Table 4.1. These results show that in terms of size, Versat can fit in the smallest FPGAs: in a Xilinx XC5VLX50T device of the Virtex V family, Versat occupies half of the logic resources; in an Altera Cyclone IV EP4CE22F17C6N device, Versat takes about 80% of the logic resources. The Xilinx implementation makes better use of the RAM resources for implementing the configuration memory than the Altera device. This explains why the Xilinx implementation consumes more RAM memory but less logic (LookUp Tables or LUTs) and the Altera device consumes more Logic Elements (LEs) and less RAM. This is also reflected in the frequency of operation, although the fact that we are comparing two different architectures also plays an important role in this difference, since the Virtex V architecture is known to achieve higher frequencies of operation compared to the Cyclone IV architecture. Nevertheless, it is a fact that the current Versat implementation is not very well optimized in terms of maximum frequency of operation and more work can be done on this once it becomes a priority.

Table 4.1: FPGA implementation results

Architecture	Logic	Regs	RAM(KB)	Mults	Fmax(MHz)
Cyclone IV	19366 LEs	4673	43.88	32 (9 bits)	64
Virtex V	12510 LUTs	4396	132.75	16 (18 bits)	102

4.2 ASIC implementation results

Versat has been designed using a UMC 130nm process. Table 4.2 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, and the node activity rate extracted from simulating an FFT kernel.

Table 4.2: ASIC implementation results

Core	Node(nm)	Area(mm^2)	RAM(KB)	Freq.(MHz)	Power(mW)
ARM Cortex A9 [22]	40	4.6	65.54	800	500
Morphosys [1]	350	168	6.14	100	7000
ADRES [2]	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 4.2, we need to use a scaling method. 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.

4.3 Execution results

The execution results of running a set of example kernels on Versat and on a embedded processor are divided in two parts: 1) Versat is compared with Microblaze (Xilinx embedded processor); 2) Versat is compared with an ARM Cortex A9 system embedded in a Zynq FPGA. In both cases a hardware timer has been used to measure the time in elapsed clock cycles. All kernels used for tests operate on vector sizes of 1024. The `lpf`, `lpf2` and `fft` kernels use Q1.31 fixed-point format.

4.3.1 Comparing with the Microblaze processor

The execution results using Microblaze for comparison are given in Table 4.3. The Microblaze and Versat programs are initially placed in their local on-chip memories. Microblaze is configured with a 32KB one-way data cache. In Table 4.3, column *MB1* gives the clock cycle counts for MicroBlaze when the data is placed in external memory, column *MB2* gives the clock cycle counts for MicroBlaze when the data is placed in the cache, column *Versat1* gives the cycle counts for Versat when the data is placed in external memory and column *Versat2* gives the Versat cycles when the data is placed in the DE memories. Two speedup results are presented: Speedup1 is the speedup including data transfer time and Speedup2 is the speedup excluding data transfer time. Comparing these two speedups one can have an idea of the overhead of moving data.

Table 4.3: Execution results using Microblaze for comparison

Kernel	MB1	MB2	Versat1	Versat2	Speedup1	Speedup2
`vec_add`	11356	6147	4517	1090	2.51	5.64
`lpf1`	15311	10756	7487	5205	2.05	2.07
`lpf2`	20644	16388	10567	8310	1.95	1.97
`cdp`	25110	12292	6673	2185	3.76	5.63
`fft`	260226	238749	16705	12115	15.58	19.71

Kernel `vec_add` is a vector addition, `lpf1` and `lpf2` are 1^{st} and 2^{nd} order IIR filters, `cdp` is a complex dot (inner) product and `fft` is a Fast Fourier Transform. The first 4 kernels use a single Versat configuration. The `fft` kernel uses multiple Versat configurations.

In terms of performance results, kernel `vec_add` is a fully pipelined kernel and produces one vector element result per cycle. It can be accelerated 5.64x in Versat but if the data is transferred into Versat just for running this kernel then the acceleration is only 2.51x. The processing time is only 1090 cycles and the remaining 3427 cycles account for data transfer and control. This means that if the data is already in the Microblaze data cache it may not be worth it to run this kernel on Versat. On the other hand, if this datapath is part of a larger kernel where the data is already in the Versat memories, it becomes advantageous to use Versat.

Kernels `lpf1` and `lpf2` have similar acceleration of about 2x with or without the data in cache. This is because the acceleration in moving the data is similar to the processing acceleration. Thus it always pays off to have these kernels executed in Versat. The modest yet effective speedup is due to the feedback loops needed to implement the filters (loop carried dependencies); they produce new vector elements every 5 and 8 cycles, respectively.

Kernel `cdp` is more complex with 4 multipliers in parallel followed by two adders. Despite the deeper pipeline, the processing speedup is not better than for the `vec_add` kernel. Unfortunately, the adders can not do an accumulation per cycle and need an external feedback loop which causes a new vector element result is accumulated every other cycle. Adding an accumulator mode to the ALUs is straightforward and will be considered in the future in order to double the processing speedup.

The `fft` kernel is the most complex kernel 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. Note that most of the control is done while either the DMA or the data engine is running, and the controller runs alone for only 566 cycles. The processing time is almost 20x smaller compared to Microblaze. Thus, a slower acceleration in data movement brings the overall speedup down to about 16x.

4.3.2 Comparing with ARM

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.

The execution results using an ARM Cortex A9 core for comparison are given in Table 4.4. In column *ARM*, the clock cycle counts is given, including the time to move the data between external DDR and the data cache. The total cycle counts for Versat are given in the *Versat* column, including the time to move the data between the Versat memories and the DDR. Column *Speedup* column gives the measured speedup and the *Energy Ratio* column compares the energy spent by an ARM system to the energy spent by ARM+Versat system. 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 4.2.

Table 4.4: Execution results using ARM for comparison

Kernel	ARM	Versat	Speedup	Energy Ratio
`vec_add`	14726	4517	2.45	2.29
`lpf1`	18890	7487	1.89	1.77
`lpf2`	24488	10567	1.74	1.62
`cdp`	25024	6673	2.81	2.63
`fft`	394334	16705	17.70	16.55

Surprisingly, despite all its advanced features, the ARM system does not do much better compared to the Microblaze results in Table 4.3. In fact for the `fft` kernel Microblaze slightly outperforms the ARM system. If this result can be generalized, it suggests that simpler processor architectures combined with accelerators like Versat would yield even better silicon area and power results. Note that, if implemented in silicon with roughly the same amount of embedded RAM, a Microblaze core would be similar in size to a Versat core if not smaller.

The ARM processor has much more hardware than Microblaze. ARM can fetch two instructions in paralell and dispatch four instructions in parallel. It has two pipelined multipliers, while Microblaze has only one. The reason why it underperforms the MicroBlaze in the `fft` kernel, can be explained by a number of factors: one factor may be the memory hierarchy, which in the ARM case is composed by two cache levels instead of one level in the Microblaze case; Microblaze runs the instructions from the local memory (with 1 clock cycle latency), while the ARM uses its on-chip memory which is in parallel with the level 2 cache and still has to go through level 1 cache; finally, there may be compiler differences that result in two different machine codes.

4.3.3 Comparing with other CGRAs

Comparing Versat with Morphosys is possible since it is reported in [23] 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.

Chapter 5

Conclusions

In this report we have introduced a new CGRA architecture named Versat and we have compared it with some existing CGRAs architectures. The main difference is that Versat can take care of the reconfiguration process itself as well as the data movement operations to and from an external memory. For that, runtime partial reconfigurations triggered by an internal controller are used. This allows Versat to accelerate complex kernels such as FFTs without using too much hardware.

Versat is a minimal CGRA with 4 embedded memories, 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.

5.1 Achievements

In this first version of Versat, the results show that the speedup improves with the kernel complexity. The kernel complexity depends on the number of datapaths that can be run sequentially using the data already in the Versat memories. In fact, our results show that single datapath kernels achieve speedups in the order of 2x while multiple datapath kernels can achieve speedups an order of magnitude higher.

Unlike other CGRAs, which are designed to accelerate one program loop, Versat is designed to accelerate a sequence of chained program loops, where the results produced in one loop are consumed by the next loop. It has been explained that most of the times the next configuration can be generated while the DMA or the current configuration is running on the data engine. Because the Versat controller can generate configurations, these do not need to be stored in external memory and then moved into Versat. In general, the code to generate configurations is much smaller than the configurations themselves.

Versat has been implemented in FPGA and in ASIC technology. In terms of silicon area, Versat is comparable to a basic low range CPU. Results on a VLSI implementation show that Versat is competitive in terms of silicon area, frequency of operation and power consumption. Performance results on running a 1024-point FFT 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. Overall performance depends of course on how often such kernels are used in a real system. However, knowledge of multimedia algorithms can reveal that between 40% to 80% of the execution time in a regular CPU can use an accelerator such as the one described here. This means that overall speedups of 2x to 5x can be expected in real systems.

5.2 Future Work

The work on Versat can be continued in many different fronts. First of all, more examples as compelling as the FFT example need to be implemented to better asses Versat's capabilities. In fact, complete applications using several such kernels should be developed.

A compiler is already under development and needs to be constantly evolved. Software libraries for host processors to use Versat are also needed. For example, to run the FFT kernel, the host program would just call a function `fft(int*x,int *X, int n)`, where x is a pointer to the input vector, X is a pointer to the result vector and n is the size of the vectors.

Finally, more work on the architecture is needed. One would like to support fast changes to Versat's data engine and that such changes are automatically reflected on the assembler and compiler tools. This would allow targeting new classes of applications in a short period of time. Another area is to incorporate floating-point units in the data engine. Other than that, many small hardware optimizations can be effected to improve the area and frequency of operation of certain blocks.

Bibliography

[1] M. hau Lee, H. Singh, G. Lu, N. Bagherzadeh, and F. 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.

[2] B. 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. ISSN 0740-7475. doi: 10.1109/MDT.2005.27.

[3] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Mapping applications onto reconfigurable Kressarrays. In P. Lysaght, J. Irvine, and R. Hartenstein, editors, *Field Programmable Logic and Applications*, volume 1673 of *Lecture Notes in Computer Science*, pages 385–390. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66457-4. doi: 10.1007/978-3-540-48302-1_42. URL `http://dx.doi.org/10.1007/978-3-540-48302-1_42`.

[4] J. T. de Sousa and J. D. Lopes. Versat, a minimal coarse-grain reconfigurable array. In *High Performance Computing for Computational Science, 12th International Meeting on*, 2016.

[5] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP – a self-reconfigurable data processing architecture. *The Journal of Supercomputing*, 26(2):167–184, 2003. ISSN 0920-8542. doi: 10.1023/A:1024499601571. URL `http://dx.doi.org/10.1023/A%3A1024499601571`.

[6] M. Quax, J. Huisken, and J. Van Meerbergen. A scalable implementation of a reconfigurable WCDMA RAKE receiver. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2004*, volume 3, pages 230–235 Vol.3, Feb 2004. doi: 10.1109/DATE.2004.1269235.

[7] J. de Sousa, V. Martins, N. Lourenco, A. Santos, and N. do Rosario Ribeiro. Reconfigurable coprocessor architecture template for nested loops and programming tool, Sept. 25 2012. URL `http://www.google.com/patents/US8276120`. US Patent 8,276,120.

[8] J. L. Tripp, J. Frigo, and P. Graham. A survey of multi-core coarse-grained reconfigurable arrays for embedded applications. *Proc. of HPEC*, 2007.

[9] R. Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 564–570,

New York, NY, USA, 2001. ACM. ISBN 0-7803-6634-4. doi: 10.1145/370155.370535. URL http://doi.acm.org/10.1145/370155.370535.

[10] H. Park, Y. Park, and S. Mahlke. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 370–380, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-798-1. doi: 10.1145/1669112.1669160. URL http://doi.acm.org/10.1145/1669112.1669160.

[11] C. Ebeling, D. C. Cronquist, and P. 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. ISBN 3-540-61730-2. URL http://dl.acm.org/citation.cfm?id=647923.741212.

[12] S. M. Carta, D. Pani, and L. Raffo. Reconfigurable coprocessor for multimedia application domain. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 44(1-2):135–152, 2006. ISSN 0922-5773. doi: 10.1007/s11265-006-7512-7. URL http://dx.doi.org/10.1007/s11265-006-7512-7.

[13] B. De Sutter, P. Raghavan, and A. Lambrechts. Coarse-grained reconfigurable array architectures. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*, pages 449–484. Springer US, 2010. ISBN 978-1-4419-6344-4. doi: 10.1007/978-1-4419-6345-1_17. URL http://dx.doi.org/10.1007/978-1-4419-6345-1_17.

[14] P. Heysters and G. Smit. Mapping of DSP algorithms on the MONTIUM architecture. In *Proceedings of the International Parallel and Distributed Processing Symposium, 2003*, pages 6–, April 2003. doi: 10.1109/IPDPS.2003.1213333.

[15] Y. Park, J. Park, and S. Mahlke. Efficient performance scaling of future CGRAs for mobile applications. In *International Conference on Field-Programmable Technology (FPT), 2012*, pages 335–342, Dec 2012. doi: 10.1109/FPT.2012.6412158.

[16] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 63–74, New York, NY, USA, 1994. ACM. ISBN 0-89791-707-3. doi: 10.1145/192724.192731. URL http://doi.acm.org/10.1145/192724.192731.

[17] V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0792384601.

[18] A. Severance and G. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, Sept 2013. doi: 10.1109/CODES-ISSS.2013.6658993.

[19] A. Severance, J. Edwards, H. Omidian, and G. Lemieux. Soft vector processors with streaming pipelines. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 117–126, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2671-1. doi: 10.1145/2554688.2554774. URL http://doi.acm.org/10.1145/2554688.2554774.

[20] M. Naylor and S. Moore. Rapid codesign of a soft vector processor and its compiler. In *24th International Conference on Field Programmable Logic and Applications (FPL), 2014*, pages 1–4, Sept 2014. doi: 10.1109/FPL.2014.6927425.

[21] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, May 2010. ISSN 0740-7475. doi: 10.1109/MCSE.2010.69. URL http://dx.doi.org/10.1109/MCSE.2010.69.

[22] W. Wang and T. Dey. A survey on ARM Cortex A processors. http://www.cs.virginia.edu/skadron/cs8535s11/armcortex.pdf. Accessed 2016-04-16.

[23] 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. doi: 10.1109/CAHPC.2003.1250345.