Versat, a Runtime Partially Reconfigurable Coarse-Grain Reconfigurable Array using a Programmable Controller

***Abstract*—Integrating a Coarse-Grain Reconfigurable Array (CGRA) in a System-on-Chip (SoC) is often a challenging endeavor, especially because of software integration issues. In this paper we show that a runtime partially reconfigurable CGRA with a most basic controller can be used as an accelerator by any embedded processor with minimal changes to the original software. The CGRA itself is easy to program and hides all its specificities like reconfiguration and data transfers from the host processor. Yet, it is capable of implementing rather complex kernels, which makes the integration job easier. We propose a CGRA architecture and a programing paradigm to support runtime partial reconfiguration. Experimental results are presented.**

Keywords*—*econfigurable computing, coarse-grain reconfigurable arrays, heterogeneous systemseconfigurable computing, coarse-grain reconfigurable arrays, heterogeneous systemsr

I. Introduction

A suitable type of reconfigurable hardware for embedded devices is the Coarse Grain Reconfigurable Array (CGRA) [1], due to its compact size and low power consumption. Fine grain reconfigurable fabrics such as FPGAs can be an alternative but, compared to 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 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 [2] integrates a small Reduced Instruction Set Computer (RISC) and the ADRES architecture [3] integrates a Very Large Instruction Word (VLIW) processor.

When programming a CGRA, it is necessary to partition the code between the reconfigurable array and the processor. Such partitioning is normally done manually, as compilers that are able to do this well have remained elusive despite two decades of research on the subject. A second problem is how the data is shared between the processor and the reconfigurable array. In Morphosys [2], a DMA block is used to transfer data between the reconfigurable array and a shared memory (loose coupling). In ADRES [3], shared registers are used between the VLIW processor and the reconfigurable array. They have mutually exclusive operation: the system is either in VLIW mode or reconfigurable array mode but both blocks have instantaneous access to the data in the shared registers. A third problem is how the reconfigurable array is configured. Both Morphosys and ADRES have a configuration memory which can store a certain number of configuration contexts. Each configuration context corresponds to a particular hardware datapath used for acceleration. If the application requires more configuration contexts than the number of configuration contexts that can fit in the configuration memory, then the extra configuration contexts have to be fetched from external memory, incurring on a performance penalty similar to a cache miss in a conventional computer architecture.

For the above reasons, it becomes quite difficult to program a CGRA. In the case of Morphosys, the programmer must separately code the context words and the main program running on the RISC processor, which includes the DMA transfers for data and configuration contexts. Also, the main program has to manage the reconfigurable array, initiating its execution and monitoring when it completes execution. These low level programming tasks can be quite distracting, as the programmer should concentrate on the application itself.

In this work we adopt a different perspective. First of all, we observe that the tasks that need to be run on a CGRA are much simpler than what is normally considered. In the DSP world many examples of these tasks can be found: Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) filters, transforms such as the Fast Fourier Transform (FFT) and the Discrete Cosine Transform (DCT), and others. These well defined *kernels* have simple control code but are very intensive in terms of data processing. Therefore we propose to replace the coupled processor in a CGRA with a much simpler controller. For example, the controller should be complex enough to control a motion estimation engine, but it does not need to be as complex as to run an entire video encoder algorithm. Moreover, the controller needs low latency access to the reconfigurable array, which may be difficult to guarantee with a more complex solution.

In our view, the CGRA is only used for the compute intensive kernels. For the rest of the code, an off-the-shelf solution such as a general purpose embedded CPU should be used. The CPU uses the CGRA as an acceleration engine and should interact with the CGRA using a application programming interface like the Open Computing Language (OpenCL) [4]. Using an off-the-shelf processor has the advantage of not having to develop a compiler for it.

In any case, a compiler is needed for the CGRA. The main difficulty in developing this compiler is supporting the reconfigurable array. Since we made the CGRA controller simple, the controller code does not need sophisticated compilation techniques and optimizations. In this way, we can focus in developing a compiler that excels in the reconfigurable array part and is very basic for the controller part. This design decision avoids the expensive integration of the compiler in a more general framework such as *gcc* or *llvm*.

In other approaches like [2], [3], the configuration contexts are prepared or compiled beforehand and are applied at runtime. One limitation of this approach is the inability to create or modify configurations during the execution of the CGRA. However, generating configurations on the fly can be interesting in the following situations: (1) supporting input parameters and (2) creating a new configuration by modifying a similar configuration already in memory.

Most CGRAs are only fully reconfigurable [3], [5], [2], [6] and do not support partial reconfiguration. The disadvantage of full reconfiguration is the amount of configuration data that must be kept and/or moved to/from external memory. Partial reconfiguration exploits the similarity between consecutive configurations, and can reduce the amount of configuration storage/bandwidth at the cost of a more complex reconfiguration hardware infrastructure. The RaPiD architecture supports partial reconfiguration but only for a subset of the configuration bitstream [7]. The PACT architecture [8] also supports self and partial reconfiguration but the reconfiguration process is significantly slow and users are recommended to avoid it and resort to full reconfiguration whenever possible.

In this paper we propose Versat, a CGRA that supports efficient runtime partial reconfiguration. Like other CGRAs, Versat features a controller module. However, unlike other approaches, the Versat controller is not meant to execute the code that cannot be run on the reconfigurable array. The purpose of the Versat controller is to efficiently deal with reconfiguration and data transfers, so that the reconfigurable array becomes a very flexible hardware accelerator that can be used by any embedded processor. The code that is not suitable for the reconfigurable array is run on a separate embedded processor for which a good set of development tools already exist. In this work, we make the reconfigurable array much more powerful and dispense with the attached processor. Compared to reconfigurable arrays that exist in other CGRAs, Versat has the following advantages:

- It can map nested loops rather than just simple loops.
- It can map sequences of nested loops rather than just single nested loops.

In order to achieve these distinctive features, we designed a minimal controller with just 16 instructions. In its memory map, the controller can see the configuration register file, which is organized in configuration spaces and fields to support partial reconfiguration. It can also see the DMA registers to be able to control data transfers to and from external memory. The controller has memory mapped access to the registers and memories of the reconfigurable array. Finally, the controller has access to a special register file used to communicate with host processors which are using Versat as an accelerator. Versat

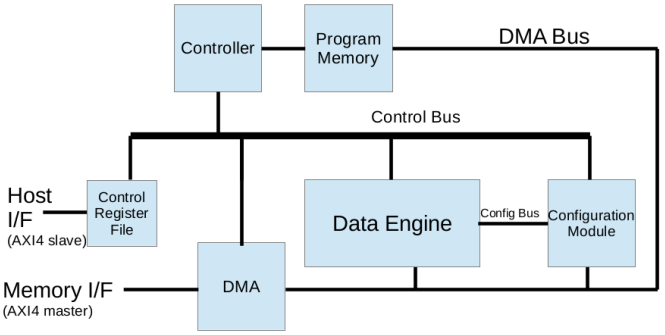

Fig. 1. Versat top-level entity

can be programmed in assembly language or in a higher level language such as C or C++. An assembler written in Python and a lightweight C/C++ compiler have been developed. To the best of our knowledge it is the only CGRA that can be programmed in assembly, thus permitting low-level code optimizations. The assembly programming is made easy by the simple structure of the reconfigurable array.

II. Architecture

Some CGRAs are homogeneous [7], i.e., they consist of a collection of identical functional units, whereas others are heterogeneous and support a diversity of FUs [9]. A careful analysis published in [10] has favored heterogeneous CGRAs as having better silicon utilization and power efficiency when compared to homogeneous solutions. Thus, we have adopted a heterogeneous CGRA architecture in this project.

The top level entity of the Versat module is shown in Fig. 1 and it looks rather standard. However, each of the modules is originally designed to meet the overall objective of a minimally controlled and yet effective CGRA. Versat is designed to carry out computations on data arrays using its Data Engine (DE). To perform computations the DE needs to be configured using the Configuration Module. The data to be processed is read from an external memory using a DMA engine, also used to store the processed data back in the memory. A typical computation uses a number of DE configurations and a number of external memory transactions. The Controller executes programs stored in the Program Memory. Each program executes one or more algorithms, coordinating the reconfiguration and execution of the DE, as well as the external memory accesses. The controller accesses the modules in the system by means of the Control Bus.

The Versat top-level entity has a host interface and a memory interface. The host interface (AXI4 slave) is used by a host system to load a Versat program containing a set of procedures or kernels, and to command Versat to execute those kernels. Using this interface, the host issues commands to Versat, and Versat reports status information and returns little amounts of data. The memory interface (AXI4 master) is used to access data from an external memory using the DMA. Large data exchanges are reserved for the memory interface. Exceptionally, if the amount of data is small, the host interface may be used for data exchange. For example, debug data is likely to use the host interface.

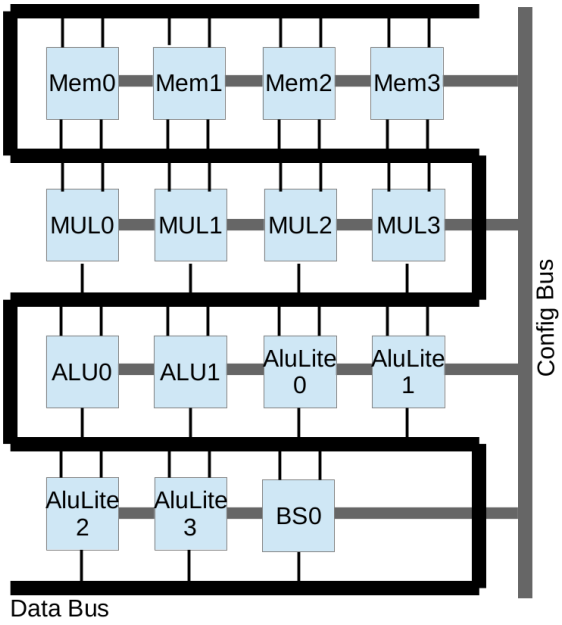

Fig. 2. Data engine

A. Data engine

The Data Engine (DE) currently has a fixed topology using 15 functional units (FUs) as shown in Fig. 2. However, it is relatively easy to change its structure to better accommodate a particular application. It is a 32-bit architecture and contains the following types of FUs: embedded memories (MEM), multipliers (MUL), arithmetic and logic units (ALU), including the lightweight versions (AluLite). 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 into a wide Data Bus. Additionally the data bus has two fixed entries for driving the constants 0 and 1, which are needed in many datapaths. Each FU is able to select one data bus entry for each of its inputs.

The FUs are configured by the Config Bus. Each FU is configured with a mode of operation and with its input selections. There are no global configurations – configuration data is always linked to a particular FU. The Config Bus is divided in configuration spaces, one for each FU. Each configuration space contains several configuration fields. Configuring a set of FUs results in a custom datapath for a particular computation.

Datapaths can have parallel execution lanes to exploit Data Level Parallelism (DLP) or pipelined paths to exploit Instruction Level Parallelism (ILP). An important type of FU is the dual-port embedded memory. Each port is equipped with an address generator to enable data-flow computing. If two memories are part of the same datapath, they will have its address generators working in a synchronized fashion. Given enough resources, multiple datapaths can operate in parallel with independent address generation. This corresponds to having multiple concurrent threads in Versat – Thread Level Parallelism (TLP).

The discussion of the details of address generation in Versat falls out of the scope of this paper. We will simply state the following properties of the address generators: (1) the address generators are compact enough to be dedicated to each memory port; (2) only two levels of nested loops, where the inner loop is shorter, are supported (reconfiguration after each short inner loop would cause excessive reconfiguration overhead); (3) if the inner loop is long then Versat will work in single loop mode and rely on its fast reconfiguration time to implement the outer loops.

B. Program memory

The instruction memory is designed to contain 2048 instructions. This is deemed enough for most compute tasks (kernels) that we have in mind. In fact, the idea is to place multiple compute kernels in this memory. However, historically, memory needs always increase beyond expected. As we develop compiler tools, we notice that it is likely that the memory capacity to contain compiled programs may need to be larger. Note that compiler produced code is never as optimized as the code that results from handwritten assembly instructions.

The program memory is divided in two parts: boot ROM and execution RAM. The boot ROM contains a hardwired program which runs after hardware reset. The execution RAM is where the user program to be run is loaded. The Versat controller can write to the program memory to load it and then can read each instruction to execute it; it cannot read words from the program memory into its data register. Alternatively, the program memory can be loaded using the DMA.

The code in the boot ROM is called the boot loader. It is used for loading programs and data files into Versat, according to instructions from the host; it is also used to download data files from Versat and to start running programs previously loaded in the execution RAM. If the size of the programs increase, it is likely that the execution RAM will evolve into an instruction cache, which will share the external memory with the DMA.

C. Control register file

Host and Versat exchange information and synchronize using the Control Register File (CRF). Both can read and write to the CRF which has 16 general purpose registers.

The CRF is used by the host to pass the address of a program to be run by Versat, along with a set arguments needed by that program. Versat uses the CRF to return status information to the host and a few data words.

D. Controller

The central claim of this paper is that runtime partially reconfigurable CGRAs can operate with a very basic controller and still be able to execute compute kernels that are normally run on this kind of machines. In order to demonstrate this, it is important to give a detailed description of the controller used. The Versat controller has a minimal architecture which supports basic program flow, loads and stores and basic arithmetic and logic operations. Its architecture is shown in Fig. 3.

The controller architecture contains 3 main registers: the program counter (register PC), the accumulator (register A) and the data pointer (register B).

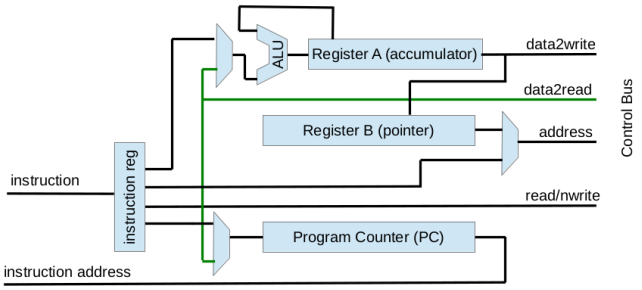

Fig. 3. Controller

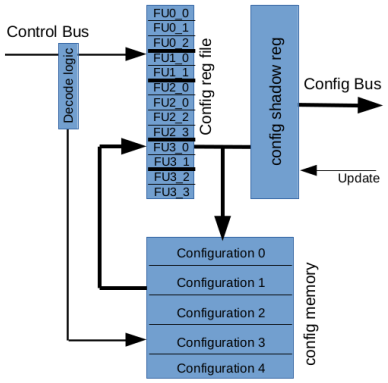

Fig. 4. Configuration module

Register A, the accumulator, is the destination of operations having as arguments register A itself (implicitly) and an immediate or addressed value. Register B is used to keep addresses used in indirect loads and stores.

The controller is the master of a very simple bus called the Control Bus. Fig. 3 shows the Control Bus signals for which an explanation is given in Table I.

TABLE I. SIGNALS OF THE CONTROL BUS

Name	Direction	Description
address[13:0]	OUT	Address to be read or written
read/nwrite	OUT	Read not write
data2write[31:0]	OUT	Data to be stored
data2read[31:0]	IN	Data to be loaded

The instruction set contains just 16 instructions used to perform the following actions: (1) loads/stores to/from the accumulator; (2) basic logic and arithmetic operations; (3) branching. There is only one instruction type illustrated in Table II. In fact, the instruction set is so small that it makes sense to outline it in Table III. Brackets are used to represent memory pointers. For example, (Imm) represents the contents of the memory position whose address is Imm.

TABLE II. INSTRUCTION FORMAT.

Bits	Description
31-28	Operation code (opcode)
27-16	Reserved
15-0	Immediate constant

TABLE III. INSTRUCTION SET

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

E. 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. 4, 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, copying it 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. 4 shows 4 configuration spaces, FU0 to FU3, where each FUj has configuration fields FUj_i of varying lengths. 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.

If the CM is being addressed by the Controller, the decode logic in the CM 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 reload them. A reloaded configuration may be used as is or partially changed.

III. Programming

Versat can be programmed in its assembly language and in its C/C++ dialect. In this section we briefly explain how to use these two languages.

A. Programming in assembly

The Versat assembler is implemented in the Python language and converts assembly code into the 16 machine code instructions supported by the Versat controller. The assembler reads a text dictionary generated by running a special Verilog testbench. The dictionary contains the names of all constants that control the hardware. In this way, one is free to change the hardware without need to change the assembler. Typically, a Versat kernel has 3 parts:

1) Data loading
2) Execution
3) Data saving

For efficiency reasons, data loads and data saves are normally performed using the DMA. The following code configures and runs the DMA:

```
#configure external address
        rdw R1
        wrw DMA_EXT_ADDR_ADDR
#configure internal address
        ldi 0x3000
        wrw DMA_INT_ADDR_ADDR
#configure transfer size
        ldi 256
        wrw DMA_SIZE_ADDR
#configure  direction and run the DMA
        ldi 1
        wrw DMA_DIRECTION_ADDR
```

After reading the dictionary the assembler can associate labels such as R1, DMA_EXT_ADDR_ADDR, DMA_INT_ADDR_ADDR, etc, with actual memory mapped addresses. Note that the first instruction loads the DMA external memory address from the CRF register R1 into the accumulator and the second instruction writes it to the respective DMA configuration register. It might have been the host system that specified the external memory address by writing it to R1. The third instruction loads the DMA internal memory address into the accumulator as an immediate value and the fourth instruction writes it to the respective DMA configuration register. In the same way, the DMA configuration registers for the transfer size and direction are configured in the next instructions (direction=1 indicates a transfer from the external memory to the internal memory). As soon as the direction is configured the DMA starts executing.

Now, one may want to take the time while the DMA is working to configure the data engine (DE). This can be accomplished by the following code

```
ldi smem2A
wrc MEM0A_CONFIG_ADDR,MEM_CONF_SELA_OFFSET
ldi smem2B
wrc MEM0B_CONFIG_ADDR,MEM_CONF_SELA_OFFSET
ldi 1024
wrc MEM0B_CONFIG_ADDR,MEM_CONF_START_OFFSET
...
```

The first two instructions are configuring memory port A of memory 0 to take as input port A of memory 2. Note that `wrc` is an alias for `wrw`, which takes the sum of its two parameters (`MEM0A_CONFIG_ADDR+MEM_CONF_SELA_OFFSET`) as the store address. `MEM0A_CONFIG_ADDR` is the configuration base address for port A of memory 0 and `MEM_CONF_SELA_OFFSET` is the address offset of the input selection field.

After configuring the DE, it is time to run it. The following code does just that.

```
ldi 0xC002
ldih 0x1D
wrw ENG_CTRL_REG
```

The first two instructions load register A with a command word, which specifies which functional units to initialize and which address generators to run. The contents of the command word is not explained in detail.

While the DE runs, one may configure the DE with the next configuration. Note that the running configuration is in the configuration shadow register, which is not affected if one writes to the main configuration register. As a result, if one manages to create the next configuration before the DE completes execution, the configuration time is completely hidden.

Now it is time to check whether the DE has finished running. This can be done by the following code:

```
wait ldi 0x0002
and ENG_STATUS_REG
beqi wait
```

The first instruction loads register A with a mask to be applied to the DE status register. The second instruction performs the logical AND of the DE status register and the mask. The mask used in this example has a single non-zero bit to check if the second LSB of the status register is set. That bit indicates whether the address generator of port A of memory 0 has finished execution. The third instruction branches to the first instruction if that bit is still reset, thus implementing a loop that waits for the DE to stop.

B. Programming in C/C++

Programming Versat in C/C++ is a lot more convenient than using the assembly language. However, Versat only supports a small subset of C/C++, deemed enough for the Versat programmer. In this section we will provide some examples of using the Versat C/C++ dialect. The Versat compiler has been implemented in C++ itself, using the `bison` and `flex` utilities to build the parser.

The first observation that needs to be made is 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 example, consider the following expression in the Versat C++ language:

```
R5 = R4+R3+(2-R6);
```

The variables R3-R6 refer to CRF registers and need not be declared by the user. The assembly code generated by the above expression tree is the following:

```
  ldi 2
sub R6
wrw RB
rdw R4
add R3
add RB
wrw R5
ldi 0
beqi 0
nop
```

This example illustrates the power of the compiler in handling arithmetic expressions. In future versions of the compiler, declaration of variables and objects and their automatic mapping to memory addresses or registers will be considered. In spite of this and other limitations, the Versat compiler has very advanced features when it comes to configuring the data engine. For example, the following code configures and runs the DMA:

```
dma.config(R1, 0x00003000, 256, 1);
dma.run();
dma.wait();
```

The predefined `dma` object has methods for configuring, running and waiting for the DMA to finish. The `dma.config` method takes as arguments the external memory address, the internal memory address, the size of the transfer in 32-bit words and the direction of the transfer, by this order. The `dma.run` method sets the DMA running and the `dma.wait` method monitors the DMA status register and exits when the DMA has completed execution. Note that one could have performed other actions in parallel with the DMA execution, and only wait for the DMA later.

The most powerful feature of our compiler is the ease in describing engine configurations. The following code configures the entire data engine to implement a product of two vectors of complex elements.

```
de.clearConfig();
for(j=0;j<R6;j++) {
  for(i=0;i<R14;i++) {
    mem2A[R7+j*R13+i] = mem0A[R7+j*R13+i]
                        +mem0B[R1+j*R13+i];
    mem3A[R7+j*R13+i] = mem1A[R7+j*R13+i]
                        +mem1B[R1+j*R13+i];
    mem2B[R1+j*R13+i] = mem0B[R1+j*R13+i]
                        -mem0A[R7+j*R13+i];
    mem3B[R1+j*R13+i] = mem1B[R1+j*R13+i]
                        -mem1A[R7+j*R13+i];
  }
}
```

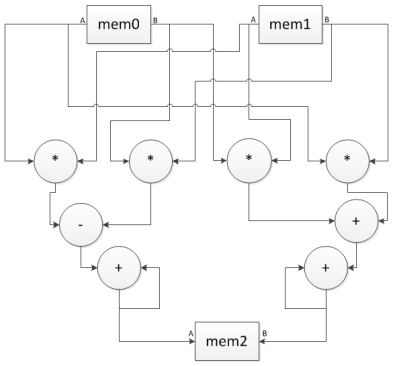

Fig. 5. Complex vector product

The predefined `de` object, which represents the data engine, has useful methods for various purposes which will be described as needed. The `de.clearConfig` method restores the engine's the default configurations. The default configurations are very likely to be used, so starting from the default values and just adding the configurations that differ from the default configurations is the fastest way to configure the engine. In fact, the compiler does this automatically. It keeps track of the state of the configuration register and only issues assembly instructions to make the configurations that differ from the current configurations. This is one way the compiler makes use of partial reconfiguration.

The two nested `for` loops in the above code describe the DE configuration. Recall that Versat's address generators support two nested loops. Memory ports are predefined objects such as `mem0A`, `mem1B`, etc. A memory port address generator is configured by its index expression and the loop limits. For example, `mem2A[R7+j*R13+i]` configures port A of memory 2 to start with the address given by R7, to increment by 1 every cycle (1 is the coefficient of loop variable i), to increment by `R13` every `R14` cycles (`R14` is the inner loop limit), and to repeat the inner loop `R6` times. The body of the nested loops contains expressions where memory ports appear as variables, and create a hardware datapath that is mapped to the DE. For the above code the datapath created is depicted in Fig. 5.

More interesting is when two nested loops are placed within a third loop. The address generators do not support three nested loops but nested loops of arbitrary depths are supported via the controller. The following code shows this:

```
do { //partial reconfiguration until done

  R1=R8+R7;

  for(j=0;j<R6;j++) {
    for(i=0;i<R14;i++) {
      mem2A[R7+j*R13+i] = mem0A[R7+j*R13+i]
                          +mem0B[R1+j*R13+i];
      mem3A[R7+j*R13+i] = mem1A[R7+j*R13+i]
                          +mem1B[R1+j*R13+i];
      mem2B[R1+j*R13+i] = mem0B[R1+j*R13+i]
                          -mem0A[R7+j*R13+i];
      mem3B[R1+j*R13+i] = mem1B[R1+j*R13+i]
                          -mem1A[R7+j*R13+i];
```

```
    }
  }

  de.wait(mem2A);
  de.run();

  R7=R7+R6+R6;
  R12=R12-1;

} while(R12!=0);
```

In the above code the `do while` outer loop is executed by the controller. In the body of this loop the two nested loops that run on the DE are placed. Note that the configuration of the DE changes for every iteration of the outer loop, since it depends on the values of registers `R1` and `R7`, which are updated inside the outer loop. If the two DE nested loops take a significant time to run, as we expect they do, then doing the outer loop on the controller, introduces a comparatively small overhead.

For each iteration of the outer loop, only the configurations that depend on R1 and R7 change. These represent the start addresses for the various memory ports. In other words, for each iteration of the outer loop one starts reading and writing to the memories at different addresses. This is another illustration of the runtime partial reconfiguration capability.

After describing the DE configuration, by means of the two nested loops, one commands the DE to wait until the address generator of port A of memory 2 has finished generating the write addresses. This is done by means of the `de.wait` method. In fact, memory 2 and memory 3 work in parallel, so one could have waited for memory 3 instead, or for both. The interesting part is why one waits for the DE to finish before we command it to run by means of the `de.run` method. The answer is that we want to wait for the DE run of the previous iteration of the outer loop to finish. In the first iteration, the `de.wait` method returns immediately as the DE has never been started. In the following iterations, the partial reconfiguration of the DE is performed by the nested loops, and only after one waits for the previous DE configuration to finish execution. Thus, DE reconfiguration occurs in parallel with DE execution. If the partial reconfiguration time is shorter than the execution of the previous configuration, the reconfiguration time is completely hidden and one will have to wait for a while before running the current configuration. However, if the partial reconfiguration takes longer than the previous configuration run, there is a reconfiguration overhead. From this explanation, the importance of partial reconfiguration in mitigating the reconfiguration overhead should be clear. If full reconfiguration were performed all the time, the chances to incur in reconfiguration overhead would be much higher.

Another way to reduce reconfiguration time is to temporarily save configurations in the configuration memory, in order to reuse them later. This will be shown in the next example. The following datapath implements simultaneous addition/subtraction of 4 vector elements.

```
de.clearConfig();
for(j=0;j<R6;j++) {
  for(i=0;i<R14;i++) {
    mem2A[i] = mem0A[i]+mem0B[i];
    mem3A[i] = mem1A[i]+mem1B[i];
    mem2B[i] = mem0B[i]-mem0A[i];
    mem3B[i] = mem1B[i]-mem1A[i];
  }
}
de.saveConfig(1);
```

Note that the vector indices do not depend on the j loop variable. That looks strange as this configuration would just repeat the inner loop `R6` times, since `R6` is the limit of the outer loop. However, this configuration is not intended to be run but rather to be saved in the configuration memory at position 1. It will be reloaded later and tweaked to implement the desired datapath. The following code illustrates that:

```
de.loadConfig(1);

mem2A.setIter(R6);
mem2A.setPer(R14);
mem2A.setDuty(R14);
mem2A.setShift(R13-R14);
...

do { //partial reconfiguration until done
  mem0A.setStart(R7);
  mem0B.setStart(R8+R7);
  mem1A.setStart(R7);
  mem1B.setStart(R8+R7);
  ...

  de.wait(mem2A);
  de.run();

  R7=R7+R6+R6;
  R12=R12-1;
} while(R12!=0);
```

After the configuration is reloaded via the `de.loadConfig` method, several manual changes to the configuration are effected by means of methods such as tt mem0A.setStart, which sets the start address of port A of memory 0. This highlights a different way of programming Versat – structural programming. Using structural programming, datapaths can be described like hardware netlists, and FUs configured individually as shown above. Below we provide a complete description of a first order IIR filter using structural programming.

```
int main() {
  de.clearConfig();
  Ralu1 = 0x73333333;
  RaluLite2 = 0x0CCCCCCD;
  mem0A.setIncr(1);
  mem0A.setIter(512);
  mem0A.setPer(5);
  mem1A.connect(alu0);
  mem1A.setDelay(6);
  mem1A.setPer(5);
  mem1A.setStart(1);
  mem1A.setIncr(1);
  mem1A.setIter(512);
```

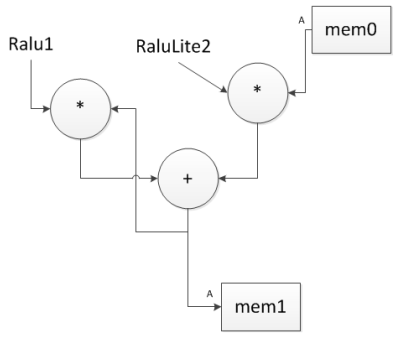

Fig. 6. First order IIR filter

```
  mult0.connectPortA(alu1);
  mult0.connectPortB(alu0);
  mult1.connectPortA(aluLite2);
  mult1.connectPortB(mem0A);
  alu0.setOper('+');
  alu0.connectPortA(mult0);
  alu0.connectPortB(mult1);
  de.run(mem0, mem1, bs0, mult0, mult1,
         mult2, mult3, aluLite3, aluLite1,
         aluLite0, alu0);
  de.wait(mem1A);
}
```

In the above description one should notice the `Ralu1` and `Ralulite2` predefined variables, which represent the output registers of the `alu1` and `alulite2` FUs, respectively. In this example these registers are used for storing filter coefficients. The connection methods such as `mem1A.connect`, `mult0.connectPortA`, etc, should also be noticed. For example `mem1A.connect(alu0)` connects portA of memory 1 to the output of `alu0`. The datapath corresponding to the above description is given in Fig. 6.

C. Packaging and using Versat kernels

Having shown how to program Versat, we will now show how can host processors use Versat kernels. In terms of hardware, Versat's standard AXI interfaces makes it ready for inclusion in a great variety of embedded systems. The Versat hardware should be integrated and host processors should take note of its base memory mapped address. In fact, running Versat kernels from a host processor is as simples as illustrated below:

```
#include "versat.h"
   ...
   //load versat program
   versat_load();
   ...
   //call a versat kernel
   versat_run(KERNEL_ID, arg1, arg2, ...);
   //may insert user code here
   ...
   //wait for versat to finish
   versat_wait();
   ...
```

The `versat_load`, `versat_wait` and `versat_run` functions are defined in the `versat.c` module, which should be compiled along with the user program. The files `versat.h` and `versat.c` constitute the Versat driver. The `versat.h` file includes a `versat_prog.h` file provided by the Versat compiler, which contains the Versat program machine code and the definition of the kernel ID's. In fact, the `versat.h` file is as simple as given below.

```
#include "versat_prog.h"
#define VERSAT_BASE_ADDR 0x7F400000
void versat_load();
void versat_wait();
void versat_run(int kernel, ...);
```

The `versat_load` function initially loads the versat program in its program memory. Since we assume the kernels in the program will be used multiple times by the host program, then the time to initially load the Versat program should dilute into a small overhead. The `versat_wait` function, as the name indicates, blocks the host program until Versat is free to run a kernel. The `versat_run` function sets Versat running and is a non-blocking function. User code can execute in parallel with Versat. When the host needs Versat again, it should call the `versat_wait` function again.

IV. Experimental Results

This section presents results on implementing Versat on FPGA and as an IP core for an ASIC. Performance results comparing Versat to FPGA and VLSI embedded processors are also shown.

A. FPGA results

FPGA implementation results are given in Table IV. These results show that Versat can be implemented in FPGA using a reasonable amount of resources. The frequency of operation in FPGA is somewhat low, which indicates that more work on improving timing is necessary. However, as the performance results will show, Versat can execute kernels using many fewer clock cycles than typical embedded processors.

TABLE IV. FPGA implementation results

Architecture	Logic	Regs	RAM(kbit)	Mults	Fmax(MHz)
Cyclone IV	19366 LEs	4673	351	32 (9 bits)	64
Virtex V	12510 LUTs	4396	1062	16 (18 bits)	102

Now we present execution results of running a set of example kernels on Versat and on the embedded processors Microblaze and Nios II. Hardware timers have been used to measure the time in clock cycles and the results are summarized in Table V.

Whether using Versat or the embedded processors, 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 results for the FPGA embedded processors have been obtained on a Xilinx ML505 board and on an Altera DE0 Nano board.

The purpose of these results is to show that Versat is capable of acceleration. However, only one example, the `fft`

example, illustrates the partial self-reconfiguration capabilities of Versat. The other four examples are single configuration kernels. Execution times of C implementations running on the Microblaze and NIOS processors have been obtained and are very similar. Thus, in column *Processor* in Table V, the average cycle counts for these two processors is given. The total cycle counts for Versat are given in the *Versat* column, and the *Unhidden* column gives the unhidden controller cycles, i.e., the number of clock cycles when the Versat controller is running but the data engine is not. This happens when it is not possible to hide the execution of the controller. The *Speedup* column is simply the ratio between columns *Processor* and *Versat*.

TABLE V. EXECUTION RESULTS

Kernel	Processor	Versat	Unhidden	Speedup
fft			638	33,93
vec_add	13355	4517	41	2,95
lpf1	36955	7487	59	4,93
lpf2	44187	10567	86	4,18
cip	566860	6673	106	17,25
fft	115114	16705	638	33,93

In Table V, `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 vector sizes of 1024.

Kernel `vec_add` is very simple and shows the lowest speedup in Versat. In Versat it produces one vector element result per cycle but the overhead of loading the data and its single configuration is significant. Moreover, in a regular processor, kernels like these, without loop carried dependencies, can undergo powerful compilation optimizations such as loop unrolling. Kernels `lpf1` and `lpf2` also show modest speedups due to the feedback loops needed to implement the filters in Versat; they produce new vector elements every 5 and 8 cycles, respectively. Kernel `cip` is more complex, using 4 multiplies in parallel followed by pipelined adders. Due to a deeper pipeline, the speedups for the `cip` kernel are greater, despite the feedback loop needed to implement the accumulation in the end; a new result vector element is accumulated every other cycle.

In Table V, 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.

B. ASIC results

Versat has been designed using a UMC 130nm process. Table VI 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 VI. IMPLEMENTATION RESULTS

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

Because the different designs use different technology nodes, to compare the results in Table VI, 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 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 $0.4mm^2$, 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 our set of kernels on Versat and on the ARM Cortex A9 are summarized in Table VII. 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 VI.

TABLE VII. 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

The results in Table VII show that, for an ASIC implementation, good performance speedups and energy savings can be obtained, even for single configuration kernels.

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 4.6x 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.

V. Conclusion

In this paper we have presented Versat, a new reconfigurable architecture that can take care of the reconfiguration process itself, as well as the data movement operations to and from an external memory. Versat is programmed in its own assembly language and in its own C/C++ dialect. It is designed to handle host procedure calls using a clean interface.

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.

The experimental results show that in general Versat can accomplish speedups even if the kernels use a single Versat configuration. For kernels that require multiple configurations, Versat can achieve speedups of one order of magnitude. This is because reconfiguration times are small and can be hidden when more than one configuration is run in a kernel.

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, in certain compute intensive kernels.

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] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, May 2010.

[5] G. Burns and P. Gruijters. Flexibility tradeoffs in soc design for low-cost sdr. In *Proceedings of SDR Forum Technical Conference*, 2003.

[6] 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.

[7] 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.

[8] 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.

[9] P.M. Heysters and G.J.M. Smit. Mapping of DSP algorithms on the MONTIUM architecture. In *Proceedings of the International Parallel and Distributed Processing Symposium, 2003*, pages 6–, April 2003.

[10] Yongjun Park, J.J.K. 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.

[11] 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.

[12] 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.

[13] 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.