Compiler for the Versat reconfigurable architecture

Rui Santiago
INESC-ID Lisboa / Técnico Lisboa
University of Lisbon
Email: rui.santiago@ist.utl.pt

João D. Lopes
INESC-ID Lisboa / Técnico Lisboa
University of Lisbon
Email: joao.d.lopes@ist.utl.pt

José T. de Sousa
INESC-ID Lisboa / Técnico Lisboa
University of Lisbon
Email: jts@inesc-id.pt

***Abstract*—This work describes the implementation of a compiler for Versat, a Coarse Grained Reconfigurable Array (CGRA). Before this work, Versat was only programmable in its assembly language. The developed compiler uses a simple and *high-level* Intermediate Representation (IR), contrasting with the complex and low-level IR found in compiler frameworks such as GCC or LLVM. Our IR is more easily translated into hardware datapaths, which are mapped to Versat partial reconfiguration instructions. The language syntax is a small subset of the C++ language, for the compiler is used only for sequences of loop nests containing operations on data arrays, as found in the target applications: digital filters, transforms and big data algorithms such as deep learning and k-means clustering. Experimental results show fast compilation time, and code size / execution time similar to handwritten assembly code.**

***Keywords*—reconfigurable computing, coarse-grained reconfigurable arrays, compilers**

I. Introduction

CGRA architectures have gained attention in the last two decades [1], [2], [3], [4], as a means to produce smaller and more power efficient reconfigurable architectures compared to FPGA architectures. CGRAs are also potentially faster to compile and to reconfigure. They consist of higher granularity processing units such as ALUs, multipliers, shifters, etc, in smaller numbers and thus using less programmable connections.

A main disadvantage of CGRA architectures is the fact that they can only execute program loops [5]. It is necessary to use a more conventional processor in conjunction with the CGRA in order to run complete applications. The data needs to be shared between the two blocks, which creates latency in the system, and normally the conventional processor is also responsible for managing the reconfiguration of the CGRA. One of the earliest CGRAs, the KressArray [6], uses static reconfiguration: the CGRA is configured once to run a complete function. In static reconfiguration, the CGRA may not have enough hardware resources to execute certain functions. Dynamic reconfiguration solves this problem by spreading the computation over several configurations. The Morphosys system [2] uses a loosely coupled RISC processor and dynamic reconfiguration. The ADRES system [1] uses a tightly coupled VLIW processor, to solve the system latency problem, and dynamic (cycle by cycle) reconfiguration. However, dynamic reconfiguration causes significant power dissipation, as large numbers of configuration registers and routing circuits are switching at every cycle.

The Versat architecture [7] turns the disadvantage of CGRAs into an opportunity to simplify its hardware and compiler. Since CGRAs can only process program loops, Versat uses just a basic 16-instruction controller for managing reconfiguration, data transfers and basic control. The target application domains, digital filters, transforms and big data algorithms such as deep learning and k-means clustering, do not require a more sophisticated controller. The host processors in a System on Chip can use Versat as a co-processor for accelerating these and other functions.

Versat solves the constrained resources problem by breaking a program loop into multiple loops with smaller bodies. This also solves the power dissipation problem because a new configuration is only switched on after the execution of a loop is terminated. By supporting loop nests, Versat reduces the reconfiguration switching frequency even further by eliminating reconfigurations in the outer loops [3]. Versat uses this technique but is limited to two levels of nesting. Loop nests having more than two levels will need reconfiguration.

While the Versat CGRA is running, its controller can algorithmically generate future configurations, which do not have to be statically produced by a compiler and stored in the hardware. Versat's reconfiguration strategy is based on the observation that, in the target applications, program loops rarely occur in isolation. Most of the times, program loops follow other program loops, where each loop uses the results of the previous loop. Because Versat can generate and schedule configurations, it can execute rather complex kernels, making the loose coupling with a host processor less of a problem. The generated configurations need to be written word by word in the configuration register file. To be able to do this quickly, and before the CGRA has finished running the active configuration, which is kept in a shadow register, partial reconfiguration is implemented: new configurations can be prepared by changing just a few configuration words of a previously generated configuration. Finally, since the active configuration switching rate is low, it does not matter, in terms of power, if the configuration register and routing logic is large. Exploiting this fact, Versat has a full mesh topology implemented by a full crossbar, which tremendously simplifies the compiler design by eliminating the need for Place and Route (P&R) techniques similar to those used in FPGAs [8]. In fact, not needing P&R makes Versat easily programmable in assembly language, which is very useful and, to the best of our knowledge, unique in CGRAs.

The Versat VLSI implementation in the UMC 130 nm technology [7] features a silicon area of 4.2mm^2, a post place and route operation frequency of 170MHz and a power consumption of 99mW. The full crossbar silicon area is only 4% of the Versat total area, which is dominated by the area

of the embedded memories. Compared to an ARM Cortex A9 processor, Versat can execute a 1024-point complex FFT 17x faster, using 10x less silicon area and consuming 249x less energy.

Before this work, Versat was only programmable in assembly language. With the present compiler, Versat can be programmed with a higher level language, having a syntax similar to C++. By focusing in the sequences of program loops used in the target applications, we have dismissed most of the C++ features and come up with a concise dialect able to fulfill our needs. Initially, the use of a standard compiler framework such as GCC or LLVM was considered. However, this idea has been abandoned because the IR in these frameworks, albeit powerful, is too low-level, a sort of universal assembly language from which it is hard to extract the loop nest parameters needed to program Versat. Those IRs can easily be mapped to the assembly language of conventional processors, provide better parsing, high level optimizations, debugging support, error message handling, etc, but destroy the high-level information that we need to configure our CGRA. Therefore, we propose a simple yet high-level and effective IR for mapping loop nests. Since we only use a small number of C++ constructs, we have decided that implementing the parser ourselves was easier than using a standard parser and having to extract the Versat configurations from its unsuitable IR.

This paper has five more sections. In section II, the Versat architecture is summarized. In section III, the structure of the developed compiler is presented. In section IV, the main constructs supported by the compiler are outlined. In section V, experimental results are presented. Finally, in section VI, conclusions are outlined.

II. Versat Architecture

The Versat architecture is presented in detail in [7]. Here, only a brief description is presented for self containability reasons. The top level entity is shown in figure 1. The Controller executes the program stored in the Program Memory (PM). The communication between a host system and Versat is done through the Control Register File (CRF). The host uses the CRF to initiate the execution of programs, pass parameters and check if the execution is finished. The Data Engine (DE) is where heavy computations take place. The controller writes or selects the DE configurations in the Configuration Module (CM). The Direct Memory Access (DMA) module is used to access the external memory and transfer data/instructions/-configurations to/from the DE/PM/CM, respectively. The DE receives a command to start running from the controller and informs the controller when the execution is finished. This process normally repeats many times during the execution of a program, for many different DE configurations.

A. Controller

The Versat controller is used for the overall control of the system, especially the reconfiguration process. It performs loads and stores (from/to the Control Bus), simple calculations, and conditional instructions. The controller has not been designed for efficiency or to support complex operations. Its instruction set consists of just 16 instructions and its performance is just enough for carrying out reconfigurations

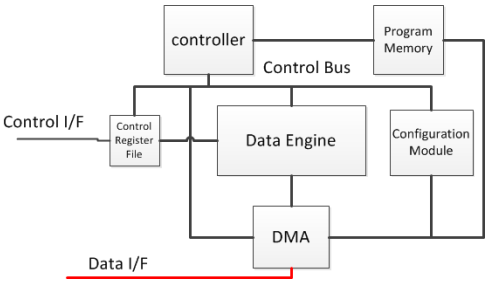

Fig. 1: The Versat top-level entity.

and simple control tasks; the intense computations are done in the DE. A simplified block diagram of the controller is shown in figure 2.

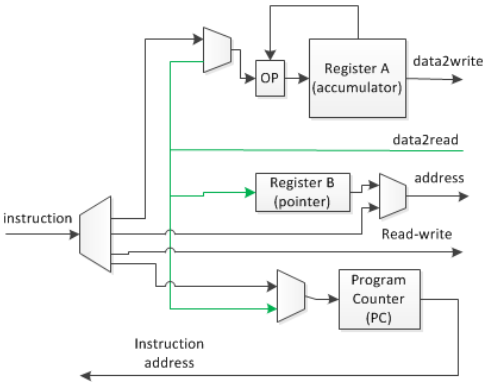

Fig. 2: Controller block diagram.

The Versat controller architecture has only three main registers: the program counter (PC), the accumulator (register A) and the pointer (register B). The program counter contains the next instruction address, as usual. The result of all operations is stored in the accumulator. Register B is used for indirect addressing or as general purpose.

B. Data Engine

The Data Engine (DE) is a 32-bit architecture and comprises 15 functional units (FUs) as shown in figure 3. There are 4 dual port embedded memories, 6 ALUs, 4 multipliers and 1 barrel shifter. The outputs of all FUs are concatenated to form a 19x32-bit Data Bus. (Since each memory contributes two outputs at most, there are **2***4+11=19 bus sections.) Each FU input can be configured to select any of the 19 sections of the bus – this implements the full mesh topology and avoids contention on the Data Bus.

The embedded memories have an address generator for each port. The address generator can support two-level `for` loop nests. It is possible to configure and run each address generator independently. The address generator parameters are described in table I. The ports can be configured to read or write the memories. The write configuration includes the selection of a data bus section to be written.

The ALU functions are given in table II. Two of the ALUs can execute all functions while four of the ALUs execute only

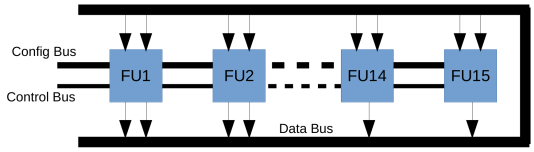

Fig. 3: Data engine block diagram.

TABLE I: Address generator parameters.

Parameter	Description
Start	Memory start address. Default value is 0.
Per	Number of iterations of the inner loop, aka Period. Default is 1 (no inner loop).
Duty	Number of cycles in a period (Per) that the memory is enabled. Default is 1.
Incr	Increment for the inner loop. Default is 0.
Iter	Number of iterations of the outer loop. Default is 1.
Shift	Additional increment in the end of each period. Note that Per+Shift is the increment of the outer loop. Default is 0.
Delay	Number of clock cycles that the address generator must wait before starting to work. Used to compensate different latencies in the converging branches of the configured datapaths. Default is 0.
Reverse	Bit-wise reversion of the generated address. Certain applications like the FFT work with mirrored addresses. Default is 0.

the first 6 functions (ALULite). The multipliers take two 32-bit operands and produce a 64-bit result, of which only 32 bits are used. It is possible to choose the upper or lower 32-bit part of the result. It is also possible multiply the output by two, which is useful when working in the Q1.31 fixed-point format. This format is common in some DSP algorithms. For the barrel shifter, one operand is the word to be shifted and the other operand is the size of the displacement. The barrel shifter is configured with the shift direction (left or right) and the right shift type (arithmetic or logic).

TABLE II: ALU functions.

Operation	Description
OR	Logical OR.
AND	Logical AND.
NAND	Logical NAND.
XOR	Logical XOR.
ADD	Arithmetic addition.
SUB	Arithmetic subtraction.
SEXT8	Sign extend from 8 to 32 bits.
SEXT16	Sign extend from 16 to 32 bits.
SHR	Arithmetic shift right.
SLR	Logic right shift.
SCMP	Signed compare.
UCMP	Unsigned compare.
CLZ	Count leading zeros.
MAX	Maximum.
MIN	Minimum.
ABS	Absolute value.

Each FU has a latency due to pipelining, which is implemented by adding registers to the output of the FU and using the retiming feature of the synthesis tool to move the registers backwards, balancing combinational paths. The number of registers added to each FU (latency) has been chosen in order to achieve timing closure. The obtained latencies are the following: 2 cycles for the ALU, 3 cycles for the multiplier and 1 cycle for the barrel shifter. Thus, when configuring a datapath in the DE, it is necessary to take into account the latency of each branch, and compensate for any mismatches when branches with different latencies converge. To do this, the address generators have the delay parameter explained in table I. The branch latency is the sum of its FU latencies.

To control the DE, it is necessary to write to its control register. It is possible to initialize/run the FUs by setting bit 0/1 of the control register. Bits 2-20 select the FUs to initialize or run. (Since each memory port can be controlled independently, there are **2***4+11=19 bits for selecting the FUs.) To check the DE, its status register is used. The status register indicates which address generators have ended execution (bits 0-7).

C. Configuration module

The configurations of the DE are stored in the Configuration Module (CM). It comprises the configuration register file, the configuration shadow register and the configuration memory. The configuration register file is used by the Versat controller to write a configuration for the DE. The shadow register allows maintaining the currently active DE configuration while a future configuration is being created in the configuration register. The configuration memory can be used to save 64 frequently used DE configurations. A configuration in the configuration register file can be saved in the configuration memory and reloaded later for use.

D. DMA

The Versat controller uses the DMA to access an external memory and transfer programs, data and configurations. The maximum block transfer size is 256 words of 32 bits. The DMA can work in parallel with the controller and the DE.

III. Versat compiler structure

The most distinctive feature of our compiler is its Intermediate Representation (IR). Driven by the need to map sequences of loop nests, and forgoing optimizations steps, a simple yet high-level IR is proposed. This data structure is a list that represents a linear sequence of commands, where some of the commands are high-level, such as the commands named *register expression* and *memory expressions*, which are discussed below. The commands are scheduled in the order by which they are read from the program text.

The register expression command is a high-level representation of an expression involving arithmetic and logic operations, to be executed by the controller. Besides grouping operations in a high-level command, there is nothing special about the register expression command, and we could have used a standard framework such as GCC or LLVM to obtain equivalent functionality. This is because the controller is a conventional machine.

On the other hand, the memory expressions command, which configures a complete loop nest in the DE, is difficult to implement with standard compiler frameworks. It is necessary to partially reverse the compilation process to recover the high-level parameters that describe the loop nest (the number of levels, number of iterations in each level, address start values and increment patterns), as well as the compute graphs in

its body. The memory expressions command captures all this information effortlessly (section IV-C).

The syntax of the C++ language has been adopted for the compiler language, because C++ is a widely used language. The C++ dialect used in this work will be called VC++ from now on.

The Versat compiler is divided in two parts as usual: the front end and the back end. In the front end the text description of the program is read and the IR is built. In the back end the IR is traversed and assembly code is generated for each command in the list. For machine code production, it is necessary to use the Versat assembler, a separate program which already existed when this project started.

A. Front end

The front end is divided in two phases: lexing followed by parsing. In the lexing phase the program file is read, keywords are recognized and tokens are returned to the parser. A particular token is returned when a particular sequence of symbols is detected during lexing. During parsing, the sequence of tokens received from the lexer is analyzed and compatible grammar rules are detected. If no matching rules are detected, errors are reported. The tools used for the front end are the well known Flex (lexer) and Bison (parser) tools. Consider the following expression in VC++:

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

The expression parse tree is shown in figure 4, where `Reg_target` is the register where the result will be saved and `expressions` are the nodes responsible for storing the expression terms. The whole expression is stored as a register expression command in the IR.

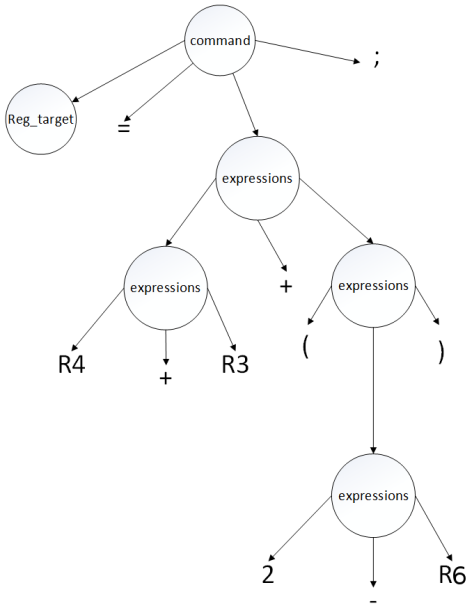

Fig. 4: Parse tree of a register expression in VC++.

B. Back end

The back end is responsible for scanning the commands in the IR and generating the respective assembly code. Each command produces a chunk of assembly instructions. No optimization steps are run before or after the sequence of instructions is produced. For some commands, generating the assembly code is straightforward. Other commands, which typically involve expression trees, require more elaborate processing.

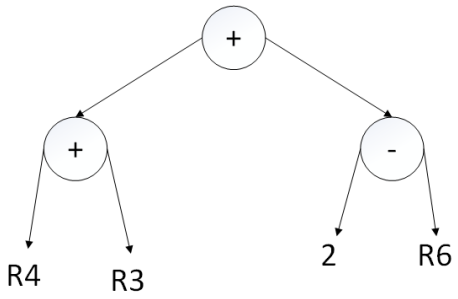

Fig. 5: Register expression tree.

The register expression command used in section III-A stores the tree shown in figure 5. The front end builds the tree taking into account the precedence of the operations, and the back end processes it from right to left. It first issues instructions to compute the expression 2-R6 and to store the result in register B. When processing larger trees, register B may be already occupied with the result of another subtree. In this case the back end returns an error indicating lack of resources to process the expression. To support larger expressions in future versions of the compiler, more registers for storing temporary results are needed. Finally, instructions are generated to compute R4+R3+RB in the accumulator. The assembly code generated by this example is the following:

```
ldi 2    //loads constant 2 in register A
sub R6   //subtracts R6 from register A
wrw RB   //stores register A to RB
rdw R4   //loads R4 to register A
add R3   //adds R3 to register A
add RB   //adds RB to register A
wrw R5   //stores register A to R5
```

IV. Versat compiler constructs

The VC++ constructs that reflect the main research proposal in this work are the *memory expressions* constructs, introduced in the beginning of section III. VC++ has many other lower-level and less original constructs, which are nevertheless useful. In this section, we describe the compiler constructs in ascending order of abstraction, ending with the description of memory expressions.

A. Predefined objects, variables and intrinsic methods

In the VC++ dialect, the programmer mostly works with predefined objects and variables, since resources are limited. The exception is for variables of type `node`, which are explained in section IV-E. The predefined objects and variables represent parts of the Versat hardware, which are exposed to the programmer. Hence, it is possible to work close to the

machine level, using a more convenient syntax compared to assembly code. The VC++ predefined objects and variables are outlined in table III. The FUs in the DE can be configured using only the predefined objects and variables. We call this *manual configuration*.

TABLE III: Predefined objects and variables.

Obj/Var	Type	Suffix	Description
mem	object	[0-3][A-B]	Allows data access to memories 0-3 via ports A or B. Used in memory expressions.
alu	object	[0-1]	ALUs.
alulite	object	[0-3]	ALULites.
mult	object	[0-3]	Multipliers.
bs	object	0	Barrel shifter.
de	object	-	Data Engine.
dma	object	-	DMA Engine.
R	variable	[1-15]	Read/write registers to use as variables. The R0 register is reserved for boot ROM use.
Ralu	variable	[0-1]	ALU output registers (read-/write).
RaluLite	variable	[0-3]	ALULite output registers (read/write).
Rmult	variable	[0-3]	Multiplier output registers (read/write).
Rbs	variable	0	Barrel shifter output register (read/write).
i, j	variable	-	Memory expression iterators (read/write).

The intrinsic methods associated with the data engine object are shown in table IV. The `run` method writes to the DE control register and the `wait` method reads DE status register. The intrinsic methods for configuring the FUs are given in table V and the intrinsic methods for connecting the input ports of an FU are explained in table VI.

TABLE IV: Data Engine intrinsic methods.

Method	Description
de.run(FU, FU, ...)	Initializes and runs the data engine. Receives as arguments the memory ports to run and the FUs to initialize. Adds automatically to the argument list all FUs of the last memory expression.
de.wait(memPort, memPort, ...)	Waits for the data engine to finish running. Receives as arguments the memory ports to wait for. Adds automatically to the argument list all memory ports of the last memory expression.
de.autoDuty(bit)	Duty parameter is calculated automatically (bit=1, default) or not (bit=0).
clearConfig()	Clears the configuration register and deallocates all FUs.
de.loadConfig(address)	Loads a configuration from a given configuration memory address to the configuration register.
de.saveConfig(int)	Saves a configuration from the configuration register to a given configuration memory address.

B. Register expressions

The Versat language supports expressions involving the predefined register variables and arithmetic and logic operators. As explained in section III-B, it is not guaranteed that long expressions can be processed. The supported operations in register expressions are listed in table VII. The use of parentheses is supported, the operations are evaluated in the order established by the precedence levels given, (lower is first,) or left to right by default.

TABLE V: FU configuration intrinsic methods.

Method	Description
mem[0-3][A-B].setStart(expression)	Sets the Start parameter of a memory port.
mem[0-3][A-B].setDuty(expression)	Sets the Duty parameter.
mem[0-3][A-B].setIncr(expression)	Sets the Incr parameter.
mem[0-3][A-B].setIter(expression)	Sets the Iter parameter.
mem[0-3][A-B].setPer(expression)	Sets the Per parameter.
mem[0-3][A-B].setShift(expression)	Sets the Shift parameter.
mem[0-3][A-B].setDelay(expression)	Sets the Delay parameter.
mem[0-3][A-B].setReverse(expression)	Sets the Reverse parameter.
alu[0-1].setOper(operation)	Defines the operation of the ALU.
alulite[0-3].setOper(operation)	Defines the operation of the ALULite.
mult[0-3].setLonhi(bit)	Selects the lower 32-bit part (bit=1) or the higher 32-bit part (bit=0, default) of the 64-bit multiplier result.
mult[0-3].setDiv2(bit)	Shifts the multiplier result 1 bit to the left (bit=0, default), or not (bit=1).
bs.setLNR(bit)	Shift is to the left (bit=1) or to the right (bit=0, default).
bs.setLNA(bit)	Right shift is arithmetic (bit=0, default) or logic (bit=1).

TABLE VI: Connect intrinsic methods.

Method	Description
x.connectPortA(y)	Connects input port A of FU x to the output of FU y.
x.connectPortB(y)	Connects input port B of FU x to the output of FU y.

TABLE VII: Supported operations in register expressions.

Operation	Description	Precedence
+	Addition	3
-	Subtraction	3
&	Logic AND	2
$>>$	Shift right	1
$<<$	Shift left	1

C. Memory expressions

In this section, the most important construct of the developed compiler is discussed: *memory expressions*. These represent `for` loop nests where only the DE memories need be explicitly instantiated by the programmer – this is why they have been called *memory expressions*. The DE memories are used as data arrays in the loop body expressions. Operators are automatically mapped to FUs. It was this feature alone that forced us to deviate from a standard IR such as GCC's or LLVM's and to develop our own IR, an ordered list of commands where memory expressions is one of the command types.

The memory expressions command stores the parameters of a loop nest. When processed by the back end, this command produces DE configuration instructions only. To run them, one needs to call the `de.run()` method (table IV). An example of a simple memory expression for vector addition is the following:

```
for(j=0;j<iter;j++)
   mem1A[start1A+incr1A*j] =
      mem0A[start0A+incr0A*j] +
      mem2B[start2B+incr2B*j];
```

Memory ports `mem1A`, `mem0A` and `mem2B` are instantiated explicitly and the addition is mapped automatically. The configuration parameters of the address generators in the memory ports (table I) can be extracted by direct inspection of the memory expression as shown in the example. Note that the parameter `iter` is common to all memory ports. The parameters can be extracted from constants or register values.

An example of a complete instantiation of a memory port in a memory expression is given below:

```
for(j=0;j<iter;j++)
 for(i=0;i<per;i++)
   mem3B[start3B+j*perPLUSshift3B+incr3B*i]=...
```

For memory port `mem3B` the extracted parameters are the following: `iter=iter, per=per, start = start3B, incr=incr3B`, and `shift = perPLUSshift - per`. This shows that the extraction of nested loop configuration parameters from the memory expressions command is trivial.

The `delay` parameter is calculated automatically by the compiler. This is useful automation as the manual calculation of this parameter is the most difficult aspect of programming Versat in assembly language. The algorithm for calculating the delay uses the memory expression tree and the FU latencies: the delay of a branch is simply the difference between the latency of the longest branch and the latency of the branch in question. (Hence, the delay for the longest branch is always 0.)

The compiler verifies the memory port parameters. If there are two different configurations for the same memory port in the loop nest body, the compiler issues an error. If the same port is in two different trees in the loop nest body, the compiler verifies if the calculated `delay` parameter has the same value in the two trees, returning an error otherwise.

While processing memory expressions, the compiler knows which FUs (other than the explicitly instantiated memories) have been used by means of a resource allocation table. An error message is issued if there are not enough resources to implement a memory expressions command. The release of resources occurs only when the method `de.clearConfig()` is called.

Anytime an FU is configured it is marked as an used resource by the compiler. Therefore, when creating datapaths using a mix of memory expressions and manual configurations, one should proceed as follows: (1) do the necessary manual configurations to allocate the resources; (2) do the memory expressions, which are guaranteed not to use the resources manually configured in (1); (3) do the connections between the manually configured resources and the resources allocated with memory expressions. After this, manually configuring more resources is not recommended before releasing the current ones, because those resources may have been used by the memory expressions.

After writing memory expressions, the `de.run()` and `de.wait()` methods will have as implicit arguments the list of FUs allocated by the memory expression. The user should pass only the manually configured FUs as arguments of these methods.

Memory expressions configure the `duty` parameter automatically with a value equal to the `per` parameter. The exception is when the programmer needs to build datapaths with feedback, which cannot be done with memory expressions alone. In this case, the automatic calculation of the `duty` parameter should be disabled using the `de.autoDuty()` method (table IV), and the duty parameter should be configured using the `setDuty()` method (table V).

D. DMA

The intrinsic methods for the DMA Engine (section II-D) are given in table VIII.

TABLE VIII: DMA intrinsic methods.

Method	Description
config(ExtAddr, IntAddr, Size, Direction)	Configures the DMA. ExtAddr and IntAddr are the external and internal memory addresses, respectively, which can be given by register expressions. Size is the number of 32-bit data words to transfer; the maximum is 256. Direction is the direction of the transfer: 1 if from external memory to Versat and 2 otherwise.
run()	Executes the DMA.
wait()	Waits for the DMA to end execution. In case of error in the data transfer, this method terminates with an error reported in register R1.
setExtAddr(expression)	Sets the external memory address.
setIntAddr(expression)	Sets the internal memory address.
setSize(Size)	Sets the transfer size only.
setDirection(Direction)	Sets the transfer direction.

Note that the external memory is normally in a separate chip, while the internal memory comprises the instruction, configuration and data memories of Versat.

E. Variables

The Versat compiler supports only one type of variables: the `node` type. These variables are used to represent the output of FUs and can be assigned in memory expressions. The following example, a dot product of two complex vectors, illustrates the use of nodes:

```
node X;
node Y;
alu0.setOper('+');
alu0.connectPortB(alu0);
alu1.setOper('+');
alu1.connectPortB(alu1);
for(j=0;j<255;j++) {
   X=(mem0B[1+j]*'mem1B[1+j])-
     (mem0A[j]*'mem1A[j]);
   Y=(mem0A[j]*'mem1B[1+j])+
     (mem0B[1+j]*'mem1A[j]);  }
alu0.connectPortA(X);
alu1.connectPortA(Y);
```

In this example, variables X and Y are used to capture the output of the two memory expressions inside the `for` loop, as shown in figure 6. X and Y are the real and imaginary parts of the complex multiplication of vector elements stored

in `mem0` and `mem1`. X and Y are connected to the input of the accumulators implemented by `alu0` and `alu1`, which cannot be instantiated in the memory expressions due to feedback. The *' operator instantiates a multiplier with the non-default parameters *lonhi = 1* and *div2 = 1* (table V).

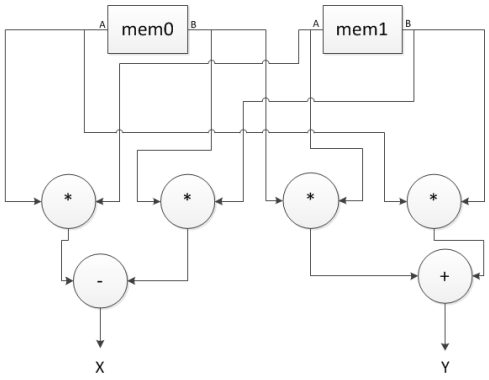

Fig. 6: Datapath that shows the use of variables.

V. Results

Experimental results using a set of programs, which had been previously written in assembly, are presented. The test programs are: vector addition (vec_add), complex dot product (cdp), Fast Fourier Transform without partial reconfigurations (fft), Fast Fourier Transform with partial reconfigurations (fft_pr), first and second order low pass filters (lpf1 and lpf2). All test programs operate on 1024-point vectors. The experimental results characterize the compiler in terms of compilation time, generated assembly code size and execution time.

A. Compile time

The compile time for the test programs is provided in table IX. The results have been obtained on a PC (@2.13GHz, 4GB RAM) running the Ubuntu 12.04 OS.

TABLE IX: Compile time.

Program	Compile time [ms]
vec_add	3.458
cdp	5.202
fft	7.507
fft_pr	11.408
lpf1	5.041
lpf2	5.346

The results show that the VC++ programs compile quickly. The most complex part of the compiler algorithm is the analysis of memory expressions. In a memory expression, the tree that represents a hardware datapath is traversed twice: once to generate the data engine circuit and another to calculate the `delay` parameter for each memory port. The maximum size of a tree is 15 nodes, which represents the entire data engine. Since there are no optimization steps, the complexity of the compiler algorithm is proportional to the number of nodes in the analyzed trees.

B. Code size

The code size of the VC++ and assembly programs, both compiled and handwritten, is studied in this section. The results are shown in table X.

TABLE X: Number of lines of the test programs

Program	VC++	Compiled	Handwritten	Overhead [%]
vec_add	7	47	38	23.7
cdp	31	145	102	42.2
fft	98	959	830	15.6
fft_pr	258	1236	870	42.1
lpf1	23	61	54	13.0
lpf2	33	111	85	30.1

The difference between the number of lines in the VC++ programs and the number of lines in the handwritten assembly programs shows that the developed compiler can significantly reduce the programmer's effort, which is what is expected from a compiler.

As one would expect, the number of lines of the compiled assembly code is higher than the number of lines of the handwritten assembly code (about 42% in the worst cases). Though this is true for most compilers, in ours the main reason for this is the fact that the compiler does not memorize the accumulator and configuration register contents, and frequently reloads them with the same values they already have. This rarely happens when the programs are written in assembly, and the problem worsens if *manual configurations* (IV-A) are extensively used. In the fft_pr example, the difference between the compiled assembly and the manual assembly is largest because there are more manual configurations in this example than in the other programs.

Even so, the generated assembly code size is close to the manual assembly code size. This happens because the programs follow a similar structure, configuring and running the data engine several times, which is basically how Versat works.

C. Execution time

An unfabricated UMC 130nm VLSI design exists for Versat [7], which can be emulated in FPGA. The execution time of each test program has been measured using the SP605 evaluation board featuring the Spartan-6 XC6SLX45T-FGG484 -3C FPGA. A Microblaze processor has been used as the host system. It is used for loading the external memory with data, loading the Versat programs, starting Versat and measuring execution times. For each example, we compare the run time of the assembly code produced by the present compiler with the run time of the assembly code handwritten by a human programmer. The results are provided in table XI.

The results show that the execution time of the compiled programs is close to the execution time of the handwritten assembly programs, which means the compiler produces efficient assembly code. The worst result occurs for the `fft` program (9.3% time overhead), which is written very compactly in VC++ using memory expressions, but generates less

TABLE XI: Execution time of test programs in clock cycles.

Program	Compiled	Handwritten	Overhead [%]
vec_add	329	318	3.5
cdp	681	642	6.1
fft	14246	13038	9.3
fft_pr	12259	12112	1.2
lpf1	2647	2645	0.1
lpf2	4227	4211	0.4

efficient assembly code. The reason for this is the fact that the configuration time is not completely hidden by the execution of the DE. When using manual configurations to perform more aggressive partial reconfigurations (fft_pr) the overhead drops to 1.2%. Also note that for lpf1 and lpf2, which do not use memory expressions, the overhead is close to zero.

Versat should be used for programs that benefit from acceleration in the DE. The controller plays an important role in configuring the DE and sequencing operations, but it should do so preferably while the DE is running – hidden control. When this is not possible, the controller becomes the only element active in the system – unhidden control. The number of clock cycles used by the DE and the number of unhidden control cycles are shown in table XII. These results show that most of the control clock cycles are effectively hidden.

TABLE XII: DE Cycles vs. unhidden control cycles.

Program	DE cycles	Unhidden cycles	Total cycles
vec_add	279	50	329
cdp	540	141	681
fft	11474	2812	14246
fft_pr	11474	785	12259
lpf1	2586	61	2647
lpf2	4124	103	4227

VI. Conclusion

In this work a first version of a compiler for the Versat CGRA architecture is presented. Central to this work is the proposal of an Intermediate Representation (IR) that conveniently captures the high-level parameters and compute graphs in loop nests, used for creating hardware datapaths in Versat. If standard compiler frameworks such as GCC or LLVM had been used, such high-level information would be crushed and it would be necessary to re-extract it from their more powerful but lower-level IR.

The proposed IR is simply a list of commands, but some of them are high-level. They are translated to assembly code by the order in which they appear in the list. The simplicity of this IR stems from the application domains considered: basic vector operations, digital filters, transforms and big data algorithms such as deep learning and k-means clustering.

The resulting Versat language (VC++) is a small subset of the C++ language. At a low level, VC++ has a number of predefined variables, objects and intrinsic methods, which expose the hardware modules to the programmer. With such constructs, structural description of hardware datapaths is always possible. At a high level, the compiler converts entire `for` loop nests into configurations of the data engine, using a single command in the newly developed IR.

A set of test programs previously written in assembly have been rewritten in VC++, compiled, and the resulting assembly code compared in terms of size to the original assembly code. The results show that the compiler produces code that is less than 42% larger than handwritten assembly code.

The compilation time for the benchmarks was always lower than 12ms, which can be considered fast. This is due, not only to the small size of the kernels used, but also to the linear complexity of the compiler algorithm, which does not have any optimization or place and route (P&R) steps. The full mesh topology of Versat dispenses with P&R.

The execution time of the compiled programs is very close to the time taken by the same programs handwritten in assembly language. Moreover, the amount of time when only the controller is running is very small compared to the total execution time. This means that, using the compiler, the control and reconfiguration times can be effectively hidden.

As future work we propose to raise the level of abstraction even more, supporting more behavioral descriptions and reducing the need for structural descriptions. Additionally, since the compiler is tailored to the architecture of Versat, it should be updated for use with a similar architecture having different numbers of units in the DE.

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] 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.

[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] J.T. de Sousa, V.M.G. Martins, N.C.C. Lourenco, A.M.D. Santos, and N.G. do Rosario Ribeiro. Reconfigurable coprocessor architecture template for nested loops and programming tool, September 25 2012. US Patent 8,276,120.

[4] L. Liu, D. Wang, M. Zhu, Y. Wang, S. Yin, P. Cao, J. Yang, and S. Wei. An energy-efficient coarse-grained reconfigurable processing unit for multiple-standard video decoding. *IEEE Transactions on Multimedia*, 17(10):1706–1720, Oct 2015.

[5] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. *Handbook of Signal Processing Systems*, chapter Coarse-Grained Reconfigurable Array Architectures, pages 553–592. Springer, 2 edition, 2013. ISBN: 978-1-4614-6858-5.

[6] Reiner 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.

[7] J.D. Lopes and J.T. de Sousa. Versat, a minimal coarse-grain reconfigurable array. In *Proc. of the 12th Int. Meeting on High Performance Computing for Computational Science*, VECPAR, Porto, Portugal, June 2016.

[8] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.