C Compiler for the VERSAT Reconfigurable Processor

Gonçalo R. Santos
Técnico Lisboa
University of Lisbon
Email: goncalo.c.r.santos@tecnico.ulisboa.pt

***Abstract*—This paper describes de development of a C language compiler for *picoVersat*. *picoVersat* is the hardware controller for *Versat*, coarse grain reconfigurable array. The compiler is developed as a *back-end* for the lcc retargetable compiler. The *back-end* uses a code selection tool for most operations, while some where manually coded. Assembly inline support was added to the *front-end* since it was missing from lcc. The compiler was integrated into a compilation framework and extensive testing was performed. Finally, some limitations were highlighted, relating to the compiler and *picoVersat* usage, and some efficiency considerations were addressed.**

***Index Terms*—Versat, CGRA, picoVersat, C-compiler, lcc *back-end*.**

I. Introduction

RECONFIGURABLE COMPUTATION has had a great focus in the past decades. Due to the fact that these systems can change their architecture dynamically to be best suit to the task being executed, they can greatly accelerate the execution of applications mainly in fields reigned by embedded systems, like telecommunications, multimedia, etc [1], [2], [3]. Many of these embedded applications need to have low power consumption and reduced silicon area while still having a high number of computations done per second.

Versat is one of these architectures capable of changing dynamically [4]. This system uses a Data Engine in order to execute code loops. *Versat*, like architectures of this type, is not designed to run efficiently when the code has a great amount of different instructions but rather when the code revolves around loops and repeated instructions in general. To do the more eclectic code there is a controller that also manages *Versat*, making it run and taking care of the reconfigurations.

The purpose of this work is to improve the *Versat* compiler solving the issues it still has. For this, some different options will be discussed, keeping in mind that this is not a traditional CPU architecture. This means that there is need to study compilers in general as well as the architecture in question.

A. Motivation

High-level languages, like **C/C++** or Java, allow for program development without the knowledge of the inner workings of a given processor. Futhermore, the development is faster and programs are easier to debug or test than using an assembly language for a specific processor. *Versat* is programmable in assembly using the *picoVersat* instruction set. The assembler program available for program development is a cumbersome form to port existing algorithms to the *Versat* platform, most of them written in **C** or **C++**. A compiler is essential for the test and assessement of the *Versat* platform capabilities. A first prototype compiler, developed by Rui Santiago [5], exists and was used for the development of some examples. However, the compiler is not only not very practical still, but also quite primitive. It supports a limited subset of the **C** language with **C++** alike method invocation, with no variables, functions or strucutures. All programming uses the *picoVersat* register names to hold values. Furthermore, the only **C++** syntax used is the method invocation from object (*object.method*), but this being the only difference from **C** and since there is not the option to create multiple instances of these objects (they act as variables), this syntax could simply be replaced with a **C** compatible syntax. This would allow for the use of a simpler compiler, since in reality none of the extra functionalities of **C++** in relation to **C** are even supposed to be used in this architecture.

B. Objectives

The objective of this work is to implement the following improvements to the Versat compiler:

- Partial reconfiguration: prepare the next Versat configuration by changing only the configuration fields that differ. Currently all configuration bits are updated during reconfiguration, which costs time.
- Support a variable number of functional units: the current implementation assumes a fixed number of functional units, and if the hardware is changed then the compiler needs to change too. This improvement will lift this restriction.
- Support variable declarations: currently all variables and objects are predefined and user defined variables are not supported. This improvement will greatly improve programmability.
- Support function calls: currently all the code must be written in a single `main()` function and no other functions can be called from this one. This improvement will allow not only for better programming practices, but also to make the code more extensible and easy to understand.

Manuscript received October 19, 2019.

C. Challenges

Building a full **C** compiler on such a bare/simple instruction set requires that most instructions need to be decomposed. All **C** language operations that are not directly supported by the architecture have to be mapped into the few existing ones.

Having such reduced register support in *picoVersat* requires the constant memory access, and not having a *stack pointer*, or a *frame pointer*, makes function calls (in particular recursion) very impractical.

Detecting, along with regular **C** code, the portions that are meant to be computed into instructions for *Versat* instead of *picoVersat* is difficult not only because is there a need to identify these "packs" of instructions as well as to analyze and to compute these instructions in a different way.

D. Document structure

This document is divided into six sections. The next section introduces the *Versat* architecture in its present form. Section III surveys existing **C** compiler frameworks that can be adapted into a full *Versat* compiler. Section IV, Architecture, provides a first approach into the general architecture of the compiler for the *Versat* platform. The next section, Compiler development, highlights the *Versat* compiler *back-end* internals. Section VI, Results, exposes the resulting compiler capabilities and limitations. Finally, some conclusions from the developed work are drawn.

II. VERSAT

Versat is a hardware accelerator based on a reconfigurable architecture where each peace of the program being executed can use a different composition of functional units [6]. It is used as a hardware accelerator for embedded systems. It uses a controller to generate and update the reconfiguration. The controller is programmable and executes programs written in **C** and assembly (specific to the architecture). Contrary to most CGRAs, which can only be fully reconfigured , *Versat* allows for partial reconfiguration (where only the different configuration bits are changed), and optimizing configuration changes. Moreover, in this approach the configurations are self-generated. Consequently, the host does not have to manage the reconfiguration process, and is free to perform more useful tasks [7]. *Versat* uses an address generation scheme that is able to support groups expressed in a single CGRA configuration. Due to the better silicon area utilization and power efficiency of heterogeneous CGRAs architectures, when comparing to homogeneous ones, an heterogeneous structure was adopted.

A. picoVersat controller

picoVersat is a minimal hardware controller with a reduced instruction set and few registers, with the purpose of doing simple calculations and giving instructions to its connected systems. Therefor, this controller is not designed for high performance computation, even if it is able to effectively implement simple algorithms. It is a programmable solution which mitigates the risk of hardware design mistakes, and helps the design and implementation of more complex control structures. This controller is simple and has a reduced silicon area which allows for a low power consumption.

picoVersat is composed of four main registers:

- Register **RA** is the accumulator and is the most relevant of the registers in the architecture. It can be loaded with a value read from the data interface or with an immediate value from an instruction. It gets the value from the result of the operations, and is used as an operand itself along with an immediate value or an addressed value.
- Register **RB**, the data pointer, is used to implement indirect loads/stores to/from the accumulator. This register is mapped in memory which means it is accessed by reading/writing as if accessing the data interface.
- Register **RC** is the flag register, storing three operation flags which are the negative, overflow, and carry flags. Much like register **RB**, this register is also memory mapped but this one is mapped to a *read only* section, being only set by the ALU.
- The Program Counter, **PC** register, contains the next instruction to be fetched from the Program Memory so that it can be executed. This register increments for every instruction, in order to fetch the next instruction except for branch, in which the **PC** register is loaded either with an immediate present in the branch instruction, or with the value present in **RB**. These two types of branches implement direct and indirect branches, respectively.

For increasing the frequency, by reducing the critical path, the controller takes two clock cycles to fetch one instruction, in pipeline. It executes every instruction that is fetched. In other words, it has 2 delay slots in the case of a branch instruction. These delay slots can be filled with a no-operation instruction (NOP), but the compiler/programmer can also use them to execute useful instructions. For instance, in the case of a for loop, the delay slots can be used to write the iteration count to some register [7].

The instruction set that is used to run *picoVersat* is minimal and has only one type. This type has 32 bits and is divided between a 4 bit *opcode* and a 28 bit immediate constant.

The operations that the controller can perform are listed in Table I. The notation used for the logic, that represents the operations, is written with **C** syntax in mind. It is also notable that *Imm* represents an immediate constant, and that $M[Imm]$ represents the contents in address *Imm* in memory.

`nop`: same as `addi 0`. It means No OPeration or do nothing. The instruction following a branch instruction is always executed due to the instruction pipeline latency (delayed branch or slot). Pad with `nop`s if no useful instructions can be executed.

This controller has the capability to work on its own, that is, the controller can run the functions and operations that do not depend on the *Versat* accelerator itself without it being connected, which allows the execution of simple applications that do not have tight time constrains.

III. COMPILERS SELECTION

The purpose of this work is to provide a compiler that should generate *picoVersat* assembly, identify and configure

TABLE I
INSTRUCTION SET.

Mnemonic	Opcode	Description
Arithmetic / Logic		
addi	0x0	RA = RA + Imm; PC++
add	0x1	RA = RA + M[Imm]; PC++
sub	0x2	RA = RA - M[Imm]; PC++
shft	0x3	RA = (Imm $<$ 0) ? RA $<<$ 1: RA $>>$ 1; PC++
and	0x4	RA = RA & M[Imm]; PC++
xor	0x5	RA = RA $\oplus$ M[Imm]; PC++
Load / Store		
ldi	0x6	RA = Imm; PC++
ldih	0x7	RA[31:16] = Imm; PC++
rdw	0x8	RA = M[Imm]; PC++
wrw	0x9	M[Imm] = RA; PC++
rdwb	0xA	RA = M[RB+Imm]; PC++
wrwb	0xB	M[RB+Imm] = RA; PC++
Branch		
beqi	0xC	RA == 0 ? PC = Imm: PC++; RA- -
beq	0xD	RA == 0 ? PC = M[Imm]: PC++; RA- -
bneqi	0xE	RA != 0 ? PC = Imm: PC++; RA- -
bneq	0xF	RA != 0 ? PC = M[Imm]: PC++; RA- -

the *Versat* CGRA architecture. Since using the current compiler as a starting point is not a good option, existing compilers were be investigated, analysed and compared in order to select a suitable candidate for *picoVersat* support.

A compiler is a tool that allows the programmer to write abstract high-level instructions and produces assembly code for the processor [8], [9]. The assembler tool can then be used to transform the assembly instructions into binary instructions of the processor. The compiler can also perform semantic error detection and data flow analysis, thus improving the required development cycle and the generated code quality. A good compiler can produce optimized code as good, or sometimes better, than the manually written assembly equivalent.

In order to transform a high-level programming language into assembly code, the compiler tool is composed of two major phases: an analysis phase, or *front-end*, and a synthesis phase, or *back-end*. During the analysis phase the input file is read, byte by byte, and structured into an abstract syntax tree (AST) containing all the relevant information for final code generation. The synthesis phase transforms an AST into machine code assembly instructions [9]. These phases correspond to, what is called in compiler terminology, the *front-end* and the *back-end*, respectively.

A. Standard Compilers

gcc is a **C** compiler [10] with thousands of generic and architecture specific optimizations. Very big, lots of people change it and maintain it, making the code very volatile, since the user must keep up to constant changes in the code. The purpose is not code optimization since *Versat* does not provide alternative instructions for its operations, only *picoVersat* could benefit of **gcc** features. It provides `asm` support and many other **gcc** specific **C** language extensions.

LLVM (*Low Level Virtual Machine*) was born as an infrastructure for optimization and it evolved into a full compiler with many *front-ends*. A new target description is provided by a declarative domain-specific language processed by the **tblgen** tool, similar to **burg** grammar descriptions. It features a stable internal instruction set implementation and documentation. Nowadays, the code is extensive, very large and difficult to manipulate. The interface is complex, written in **C++**, but stable. Requires a big effort for a single person for a few months.

The **lcc** is a **C** compiler developed after the publication of the **BURG** papers by Proebstring, Hanson and Fraser in 1992, as a demonstration of the concept [11], [12], [13]. Up to that time *back-end* compiler writing was an art. The **burg** concept allowed to describe the target architectures in a single file as well as maintaining all of them in a single executable, a retargetable compiler. The target architecture is selectable by a command line option. Only the **C** *front-end* is supported with a custom made analyzer. The input program is described in an AST with a well documented 32 instruction set. Optimizations are performed in the AST, and through common sub-expression analysis, the tree is converted into a **DAG** (*Directed-Acyclic Graph*), although the original tree is still accessible. A single **C** *front-end* reduces the complexity and line count. Simple to introduce new *back-end* since it was designed to be retargetable. A detailed and extensive book, and well defined *back-end* interface, provides a good work basis [14]. It provides no `asm` directive and it's introduction will require changes to core of the *front-end* code. However, the compiler core is composed of 260K lines of code and any changes should be accessible.

Other compilers, like **tcc** (*Tiny C Compiler*), **pcc**, and Amsterdam Compiler Kit (ACK) or **sdcc**, are small and simple but not as well documented. Also, the *back-end* code generation must be coded without the help of a code selection tool.

B. CGRA Compilers

Unlike the previous general purpose **C** language compilers, CGRA compilers directly address the problem of configuration and reconfiguration of a CGRA platform.

Some CGRAs, like ADRES, Silicon Hive, and MorphoSys are fully dynamically reconfigurable: exactly one full reconfiguration takes place for every execution cycle [15]. Other CGRAs like the KressArray are fully statically reconfigurable, meaning that the CGRA is configured before a loop is entered, and no reconfiguration takes place during the loop at all [15]. Still other architectures feature a hybrid reconfigurability. The RaPiD architecture features partial dynamic reconfigurability, in which part of the bits are statically reconfigurable and another part is dynamically reconfigurable and controlled by a small sequencer [15]. Most compiler research has been done to generate static schedules for CGRAs [15]. Not enough details are available in the descriptions of the compiling techniques, and few techniques have been tried on a wide range of CGRA architectures [15]. For that reason, it is very difficult to compare the efficiency, effectiveness and user interface of the different techniques, this is, compilation time, quality of the generated code, and how easy it is to use and to port code to the language used, respectively [16].

There are frameworks that try to compile code in usual languages to CGRAs. Most of these frameworks have however

some limitations that do not allow them to adapt the code fully to CGRAs' purposes. For example, the IMPACT compiler framework is used in order to parse **C** source code and to get the Intermediate Representation (IR) [17]. We can then change what the *back-end* does to this IR, to get the desired result. However, because of the IMPACT *front-end*, most algorithms can only handle inner loops of loop nests [17]. And since changing the *front-end*, as well as the *back-end*, is the same as creating a new compiler, most of these frameworks have problems when there is a need to really adapt the code not made in the specific assembly language of the CGRA, which is supposed to run the code.

IV. Architecture

The development of a high-level language compiler, with variable and function support, is the main objective of this work. In order to achieve this goal, two approaches can be taken: the existing compiler can be extended, or a new compiler can be developed.

The existing compiler, although it provides valuable information, adopted many design decisions that make the required extensions very tedious and complex. Furthermore, there is no documentation and the lone developer is no longer available to explain any doubts that might arise, since only the source code is accessible. The alternative, the development of a new compiler, can be start from scratch or use an existing compiler.

The development of a new compiler for a rather large and complex language like **C** is a daunting task, and would require human resources beyond the scope of this thesis. Since several retargetable **C** compilers exist, where the *back-end* can be reconfigured to a new processor, a selection must be carried out. This options also implies that the *front-end* remains unchanged and that the full **C** language is implicitly support. Consequently, all existing **C** programs that, when compiled, will fit the *Versat* memory can be used. Also, the development of new software can be made without any specific language or architecture common knowledge other than the **C** programming language.

A. Compiler selection

After analyzing the previous options it was decided that the best approach was to take an existing compiler and add the *picoVersat back-end* to it, making it a *Versat* CGRA compiler. Even using another compiler as a starting point some code from the existing compiler that interacts with *Versat* can be adapted to fit this model.

The work begins with the development of a simple compiler which is able to parse **C** code, and generate the assembly necessary to run the **C** program in the controller (*picoVersat*). Since the controller can technically work on its own without the *Versat* accelerator, the compiler should also be able to reflect this. So even with a reduced instruction set, and limited memory, simple applications have the capability to be run in this component.

From the compiler candidates studied in the previous section, the **lcc** compiler seems to be the most promising approach. It offers an instruction selection tool that can simplify the conversion process, but no `asm` directive (not a **C** standard feature) is available. The first stage of the development process is to describe all standard **C** operations and functions using *picoVersat's* assembly. This will allow the code to be parsed from **C** code to the language native to the controller which will then go through the assembler to generate the machine code needed to run the desired program. The assembler is already working with the instructions described in Section II-A so only minimal changes will need to be done to this one, in order to generate the correct machine code.

B. picoVersat back-end

The first stage in the development process consists in constructing a *back-end* that produces *picoVersat* assembly code for the selected compiler. Since we are choosing a retargetable compiler, the introduction of a new *back-end* is not a complex task. At this stage, all *back-end* low-level operations must be mapped into *picoVersat* assembly. The *picoVersat* instruction set is composed of a small number of instructions (see section II-A) when compared with general purpose processors, like the I386 or the ARM processors. Therefor, the mapping of the **C** language constructs into such a small instruction set is a complex task. Most constructs will surely need long sequences of *picoVersat* instructions. Furthermore, some of the *picoVersat* registers must be permanently assigned to compiler management tasks, such as a `stack-pointer` and a `frame-pointer`, while other register may require temporary allocation for tasks such as indirect load or save operations. On general purpose processors, the register set includes such registers (`esp` and `ebp` in I386, for instance) and complex load and save instructions that can combine up to 3 registers (`lea` in I386) allowing the addressing of field in vectors of structures.

A compiler like **lcc** must also be extended to support the `asm` directive, since it does not support it yet. Please note that such instruction, although useful for the tasks required of this compiler, is not a part of any **C** language standard. It started as **gcc** extension and has since been adopted by other compilers. This instruction is also useful to provide direct control over the *Versat* data engine.

Figure 1 highlights the **lcc** main blocks that need to be modified in order to obtain a workable **lcc** compiler for *picoVersat*.

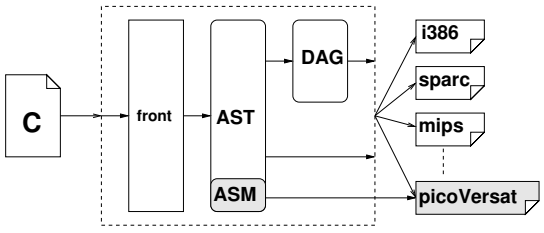

Fig. 1. picoVersat *back-end* integration into **lcc**.

C. Data Engine Incorporation

Even though *picoVersat* (Controller) can work on it's own for very simple problems or auxiliary calculations, it's main

purpose is to manage *Versat*, that is, all of its components. In particular, the controller is meant to manage the Data Engine in order to take full advantage of the accelerator. This is the biggest challenge when developing a compiler for this type of architectures.

No matter what compiler is chosen, one thing that must hold is that all code written needs to be compatible with the syntax used for the controller, that is, the code used to incorporate *Versat* support to the compiler needs to be compatible with the code used to compile *picoVersat*. Also, in an ideal scenario, the compiler should not need to be different for compiling code with, or without the accelerator's functions, since the controller can operate separately.

V. Compiler development

The **lcc** compiler is a retargetable compiler with a single *front-end* for the **C** programming language (ANSI-C). As a retargetable compiler, multiple *back-ends* are available and new ones can be easily added. Currently, **lcc** produces **mips**, **sparc** and **intel-x86** assembly, amongst other output formats. Most formats, including the three referred, use an instruction selector to generate and optimize the output assembly code. The **lcc** compiler uses a specific instruction selection tool (**lburg**), included in the compiler distribution. Ideally, the creation of a new *back-end* corresponds to an **lburg** grammar description, some auxiliary functions, and a controlling structure.

A. Compiler interface

As referred, the *back-end* is registered in the compiler by a single data structure, `Interface versatIR` in this case. This structure defines the processor metrics. In *Versat* all data types have a size metric of 1 and the align metric is also 1, so `sizeof(char)==1` as required by the **C** language.

The *back-end* data structure also defines some architecture requirements. These include the endian number format representation, whether multiplication and division are executed by software libraries, whether the *back-end* can handle a **DAG** (directed acyclic graph), or if it can handle passing structures to and from functions.

The final part of the *back-end* data structure defines a set of procedures to handle specific parts of the code generation.

B. Register assignment

In order to implement the **C** language constructs, the processor must provide an accumulator to handle return values from functions, a stack pointer to save arguments, locals and spills, and a frame pointer to access arguments and locals by a fixed amount. All of these registers can be set in fixed memory positions, but the execution degradation is significant.

The *picoVersat* has 16 registers, but only registers `R1` through `R12` are available as program parameters. The register assignment set `R12` as a stack pointer, and `R11` as a frame pointer. The stack pointer is initially set to the highest memory position `0x1FFF`, and the stack grows downward to lower memory addresses. Also, the stack pointer points to the last used position. Hence, the address `0x1FFF` is used to store the return address `end` before the `main` is called.

The accumulator is not a fixed register and requires spilling only when non-void functions are called, in order to hold the return value. The first register `R1` (ACC=0) is assigned as accumulator. The remaining registers are of free use by the compiler. However, to make code generation more efficient, about half of the available 10 registers (from `R1` to `R10`) can be assigned to temporary values, and the others to store program variables. Program variables stored in register speed up significantly the program execution, specially if they require indexing, such as function arguments, locals and vector indices. As `R1` and `R2` are already used as temporaries by some instructions, registers `R1` through `R5` are defined as temporaries. Registers `R6` through `R10` are defined as variables registers. Registers `R12=SP` and `R11=FP` are permanently assigned.

C. Code selection

The grammar for the code selection is defined in the second area of the **lburg** description file. Each grammar rule defines a non-terminal target, a tree pattern, an output string, and a selection cost. The selection cost value represents the latency of the full instruction sequence. In *picoVersat* all instructions take a single clock cycle, so the cost is the number of instructions. For dynamic selection of instructions, the cost value literal is replaced by a function call. This function evaluates the node and returns a cost value.

D. Code emitting

The sequence too complex to emit code based only on a string template must be handled by the `emit2` routine. This routine is selected by starting the rule output string with the `#` symbol. In *Versat* some instructions require temporary labels like `CALL`, `RSH` or `LSH` and must be handled by `emit2`.

The `CALL` instruction does not exist in *picoVersat*. Its emulation must store the return address before jumping to the address of the routine. Since calls from different places require a different return address, the routine creates a new label name for each call.

In *picoVersat* the shift operations only shift one bit (left or right). In order to support multiple shifts in a single instruction a loop must be implemented. Shift operations are performed by a cycle that decrements the counter and shifts the destination register by one.

The code generation requires that a comparison is coded as a branch when the codition hold *true*. If the condition hold *false*, the instruction should not branch to the given label. The comparison *greater-than-unsigned* (`GTU1`) requires that the result is *not-zero* an *no-carry*, a label was required to implement the *logical-or* using branches.

picoVersat does not have instructions for multiplication or division, like many low budget processors. For these processors, the operations are performed by library functions. The `mulops_calls=1` flag in the *back-end* configuration data structure makes the compiler generate regular functions calls for these operations. However, the coding of these operations

can be optimized, since SP=R12 manipulation is simpler because 3 pushes are performed in sequence, with register saves.

For large integer literals two instruction must be emitted, one for the low bits and another for the high bits. Since the ldi instruction can handle upto 28 bit constants a 0xFFFFFFF mask is used, while the ldih instruction sets the literal high nibble by shifting right 28 bits the constant.

E. Function handling

The function() routine must handle all the specifics of defining a function, including its activation frame. The activation frame is the organization of arguments, return pointer, saved registers, frame pointer, and local variables of a routine. Its memory mapping depends on whether the arguments are passed in registers or on the stack, *etc.*

F. Code optimization

A code selection tool like **lburg** allows the insertion of alternative tree selection patterns that can be used in specific trees with an inferior cost than the generic rules required for all instructions.

As referred above, the add instruction (ADD+IUP) can be performed with a register add or with an immediate value addi. Although the cost is 3 in both cases, the use of an immediate value saves a register and the respective store instruction that was previously required. Note that the operation between two registers is required to perform sums, while the sum with a constant is an optimization when one of the values is not already stored in a register.

The instructions BAND+IU, BXOR+IU, ASGN+IUP, and ARG+IUP can not handle immediates. However, by replacing a rdw by a ldi instruction a register is also saved.

The shift instructions are very costly since they must be implement through loop. In order to unroll the loop, optimizations can be defined for each shift value, as long as it is a literal. When both values reside in registers, no optimizations are possible.

The *Versat* processor is controlled by *picoVersat* by writing values to specific memory positions. Consequently, vector addressing instructions are common and should be optimized. To optimize such instructions, the compiler was run in debug mode, where it emitted the trees it was selecting. Two such cases were identified, where global vectors where indexed by literal and assigned literals: int vec[10], *ptr; vec[6] = 3; ptr[6] = 3; Local vectors require frame pointer indexing and are implicitly slower.

G. ASM support

The use of an asm call is important to have a low level control over the *Versat*. Since the compiler did not support asm calls a generic support was added. Any *back-end* can activate the doasm flag in its progbeg routine and a CALLASM instruction is generated for selection.

Note that only literals at compile time can be used because the values must be outputted to the assembly file. Since register assignment is performed by the compiler, there is no way of knowing which register will be assigned to a given variable. Exceptions are global variables, always referred by name, and locals that have a fixed offset to the frame pointer. However, in the later case the user must write simple routines, with no register spilling, so that the offsets are known.

H. Program bootstrapping

The program bootstrapping consists on setting up the processor before calling the main() routine, the actual calling of the main() routine, and the cleaning up after calling the main() routine.

The setup of the main routine sets the stack pointer to the top of the stack and sets the frame pointer to zero. The top of the stack must be determined from ADDR_W and MEM_BASE, MEM_BASE+2**(ADDR_W-1)-1, and stored in R12.

No arguments (argc, argv, envp) are passed, so the main routine is directly invoked. The return address end is saved at the top of the stack, the frame pointer R11 is set to zero, the main() routine is called, and the return address is defined.

Finally, to end the program, a trap must be generated. The trap address is MEM_BASE+2**ADDR_W-1 and is determined in the way as the top of the stack:

I. Runtime support

Since the assembler does not support multiple files and there is no support for explicit file liking, the runtime support must be added by include files. These include files have the usual declarations and defines, but also routines fully coded. There is no problem of multiple definitions since there is no linking.

The putchar routine uses the *picoVersat* debug capability to print a single ASCII character to the simulation terminal. It allows the debug and testing of the program examples used in this work.

```
void putchar(int ch) {
    asm("\trdw R11\n\taddi 2\n\twrw RB\n"
        "\trdwb\n\trdwb\n\twrw CPRT_BASE\n");
}
```

The function is called using the usual **C** language convention, and then the function argument is fetched from the *stack-frame*. The offset (2) accounts for the saved *frame-pointer* and the function return address, both pushed to the stack after the argument was saved on the stack.

Memory allocation on the heap uses the memory between static data (functions and global variables) and the top of the stack. The top of the stack is maintained by register R12. To signal the end of the static data, the compiler inserts an integer variable, called _end, and initialized to zero (0). The value of this variable is then used as the head of an allocation block list, entirely written in **C** (malloc.h). The allocator checks whether the required memory would overlap with the stack, but since stack allocations (function calls, arguments and local variables) do not perform the inverse check, stack overruns are possible.

Implementation of memory allocation on the stack (see the `alloca` **C** library routine) requires the effective movement of the stack pointer. Special care must taken during expression evaluation, since temporaries may clobber the stack. The implementation provided is very simple and decreases the stack by the given amount `sp -= size` (in 32-bit words). The `sp()` routine then returns the new stack top value, which is the allocated block lowest address, since the stack runs from high addresses to low addresses.

J. Application integration

The compiler **lcc** is composed of several applications that, when used in sequence, produce an executable from a given source file. The `lcc/etc` directory contains the code for the generation of the top level `lcc` application that controls the prepprocessor (**cpp**), the compiler (**rcc**), the assembler (**va**) and the loader. The `lcc/etc/versat.c` file controls arguments and invocation of these sub-applications. The preprocessor is invoqued with `-Dversat` so that programs can use `$ifdef` directives to select specific code. The compiler must be invoqued with `-target=versat` to select the *back-end*. The assembler is invoqued with an optional third argument that points to `xdict.json` file. The *verilog* compiler (**iverilog**) works as a loader since it allows the generation of a final executable from the assembly file generated.

K. Data engine incorporation

The *Versat* CGRA configuration is memory mapped, thus each functional unit can be configured by writing to specific memory addresses. The writing can be performed by ordinary **C** language assignment instruction of the form `*addr = value;` where the `addr` variable was previously set to the specific memory address.

Alternatively, the structure containing all parameters, defined in `versat.h`, can be used to assign only the desired parameters, or all of the parameters in sequence.

```
#define VERSAT_MEM0A 5120
#define VERSAT_FU 6176

typedef struct versatFU {
    struct mem { int iter, per, duty, sela, start,
                     shift, incr, delay, rvrs; }
        mem0A, mem0B, mem1A, mem1B,
mem2A, mem2B, mem3A, mem3B;
    struct alu { int sela, selb, fns; }
        alu0, alu1,
alulite0, alulite1, alulite2, alulite3;
    struct mult { int sela, selb, lonhi, div2; }
        mult0, mult1, mult2, mult3;
    struct bs { int sela, selb, lna, lnr; } bs;
} Versat;

typedef struct versatDE {
    int mem0A, mem0B, mem1A, mem1B,
        mem2A, mem2B, mem3A, mem3B;
    int alu0, alu1,
        alulite0, alulite1, alulite2, alulite3;
    int mult0, mult1, mult2, mult3, bs0;
} VersatDE;
```

Now, by including `versat.h`, all parameters are accessible through a single memory mapped structure initialized to the base address of each parameter base.

```
#include "versat.h"
static int base = VERSAT_MEM0A, versatFU = VERSAT_FU;
int main() {
    Versat *versatFu = (Versat*)base;
    VersatDE *versatDE = (VersatDE*)versatFU;
    /* ... */
    return 0;
}
```

Also, a structure can be defined with all required parameters and then copied into the required memory position simply by **C** structurs assignment.

```
#include "versat.h"
static int base = VERSAT_MEM0A;
Versat config1 = { { 2000, 1, 1, 0, 0, 1, 1, 0, 0, }
                   /* ... */ };
int main() {
    Versat *versatFu = (Versat*)base;
    /* ... */
    *versatFu = config1;
    /* ... */
    return 0;
}
```

VI. Results

The aim of this work is to produce a workable **C** language compiler for the *Versat* architecture using the *picoVersat* instruction set. The success can be measured by the number of **C** language constructs that are working properly. Consequently, testing is of primordial importance, as are the range of tests used to exercise the compiler.

A. Testing

In the test of a compiler, where a small change can affect the generation of multiple instructions, a good set of regressive tests is very important. In order to automate the process, a `test/` directory was setup. This directory includes a set of **.c** test files and the expected output **.out**.

The `Makefile` compiles, executes and compares the new result with the previously stored result. All differences in output are printed and can then be analyzed.

Since the output from **iverilog** includes the number of clocks spent, it is easy to compare whether the changes in the compiler result in improvements, or in performance degradation.

A set of 86 regression tests is currently being used, ranging from specific operator testing to complex recursive and iterative examples.

B. Limitations

The **C** language imposes that `sizeof(char)==1` as does the **lcc** compiler (see [14, p.79]). This works fine as far as `sizeof(char)` can be 32-bits. However, additionally, the **lcc** assumes through out the code that 8 is the number of *bit-per-byte*. If it was a variable, one could set it to 32. As it is hardcoded, all address literals will be truncated to 8-bits (`8^ty->size`).

```
int *addr = (int*)0x123456;
```

This can be avoided by setting an integer to the required value and then assigning it to a pointer. This works since

integer literals are 32-bit wide and the conversion to pointer, controlled by the *back-end*, does not truncate the value.

```
int *addr, value = 0x123456;
addr = (int*)value;
```

Nevertheless, defining literal pointer is never a good predictive in virtual memory machines. In *Versat* it is useful to map variables to specific addresses.

Due to the same reason, a warning message is issued (`shifting an 'int' by 12 bits is undefined`) but the code is correctly generated.

Finally, the compiler does not support floating point data types, since every operation must be supported by library routines. This is the case for many android devices, namely smartphones. However, the *Versat* purpose is to perform integer arithmetic operations fast and is not aimed at scientific programming. The error message `compiler error in _label--Bad terminal` is issued by the compiler when it cannot handle a given operation, namely floating point operations.

C. Register assignment

Register assignment in compiler design considers two types of registers: global registers that hold variable values and scratch registers that hold temporary values. The **lcc** compiler defines these registers by setting a mask for each type of register. It is up to each *back-end* to define the mask values according to processor capabilities. For instance, the **sparc** processor defines 4 sets of 8 registers: global, temporary, input and output; where the later two sets replace the stack for argument passing. In **i386** all 7 registers are temporary, while **mips** uses half for each purpose $(16 + 16)$.

Since the *picoVersat* has no specific register assignment, a study was carried out in order to assess the best balance between global and scratch registers. Registers `R0` and `R13` to `R15` are used to communicate with *Versat* and are invisible to the compiler. The stack is controlled by a *stack pointer* (`R12`) and a *frame pointer* (`R11`). The remaining registers (`R1` to `R10`) compose a mask `7FE`, where the lowest bit (`R0`) is omitted for register assignment and the highest used bit `400` is `R10`. The register `R1` is used to return function values and all arguments are passed on the stack. At least two registers must be used as scratch for binary operations temporaries. The compiler allows the definition of a `tmask` for temporaries and a `vmask` for variables.

Initially, in run 1, the experiment uses all registers for temporaries. Each run adds a variable register at the expense of a temporary, until only two temporaries remain (run 9). Three examples where used: `assign`, `repeating locals` and `bubble sort`. The first two represent opposite extremes of register usage, while the last is a more balanced and realistic example.

The first example uses the **C** language right associative *assign* operator where each new assignment to the variable `a` requires a new temporary register.

```
int f() { return 1; }
int main()
{
  int a = f() + (a = f() + (a = f() + (a =
      f() + (a = f() + (a = f() + (a =
      f() + (a = f() + (a = f() + (a =
      f() + (a = f() + (a = f() + (a =
      f() + (a = 1))))))))))));
  return a;
}
```

The register usage shows that each assign uses a register 4 times at the expense of the return register `R1`. The best solution, represented by the lowest clock count, is to use only two temporaries, since more variables imply more stack (`R12`) saves and restores between each call to the function **f**.

run	vars	vmask	tmask	clks
1	0	000	7FE	677
2	1	400	3FE	638
3	2	600	1FE	633
4	3	700	0FE	628
5	4	780	07E	623
6	5	7C0	03E	618
7	6	7E0	01E	613
8	7	7F0	00E	608
9	8	7F8	006	603

The second example uses lots of repeating local variables so that each one is assigned a register, for its uses from the first to last line, if one is available.

```
int func(int a, int b, int c, int d, int e, int f,
    int g, int h, int i, int j, int k) {
    a = a + b - c - d - e + f - g + h + i + j + k;
    b = a - b + c + d - e - f + g - h + i - j - k;
    c = a + b - c - d + e + f + g - h + i + j - k;
    d = a - b - c + d - e + f - g + h - i - j - k;
    e = a + b + c - d - e - f - g + h - i + j - k;
    f = a - b - c + d - e + f + g - h - i - j + k;
    g = a + b - c - d + e + f + g - h + i + j - k;
    h = a - b + c + d - e - f - g + h + i - j - k;
    i = a + b - c - d - e + f + g + h + i + j - k;
    j = a - b - c + d - e + f + g - h - i - j - k;
    k = a + b + c - d + e - f - g - h - i + j + k;
    return a + b + c + d + e + f - g + h - i + j - k;
}

int main() {
    return func(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0);
}
```

As expected, the best solution is to use highest of temporaries in order to reduce frame pointer (`R11`) accesses to stack saved values.

run	vars	vmask	tmask	clks
1	0	000	7FE	956
2	1	400	3FE	1001
3	2	600	1FE	1051
4	3	700	0FE	1106
5	4	780	07E	1161
6	5	7C0	03E	1211
7	6	7E0	01E	1278
8	7	7F0	00E	1356
9	8	7F8	006	1447

The last example, the *bubblesort*, uses a mixture temporaries and variable reuses.

```
#include "printi.h"

int bubble(int list[], int n) {
    int c, d, t, swap, cnt = 0;

    for (c = 0; c < n - 1; c++) {
```

```
        for (swap = 0, d = n - 1; d > c; d--)
            if (list[d - 1] > list[d]) {
                swap++;
                t = list[d];
                list[d] = list[d - 1];
                list[d - 1] = t;
            }
        if (!swap)
            break;
        cnt++;
    }
    return cnt;
}

int v[] = { 7, 4, 9, 6, 2, 1, 3, 5, 8, 0 };

int main() {
    int i, size = sizeof(v) / sizeof(v[0]);
    int cnt = bubble(v, size);
    for (i = 0; i < size; i++) {
        putchar(v[i] + '0');
        putchar(' ');
    }
    printi(cnt, 10);
    putchar('\n');
    return 0;
}
```

This example exploits the tradeoff between global and temporary register usage. In the first runs the compiler is unable to use all temporaries. In the last runs some variable registers are left unassigned and the number of required execution clocks rises again.

run	vars	vmask	tmask	clks
1	0	000	7FE	9855
2	1	400	3FE	8193
3	2	600	1FE	7714
4	3	700	0FE	7399
5	4	780	07E	7022
6	5	7C0	03E	6949
7	6	7E0	01E	6949
8	7	7F0	00E	6921
9	8	7F8	006	7492

Based on experience with the examples above, a balanced approach should work best in most cases. Therefor, the first five registers, `R1` to `R5`, are used as temporaries and the remaining five, `R6` to `R10`, are used as variables.

D. Efficiency considerations

Calls are very expensive operations for any processor. *Intel* has made a significant effort over the year to address this problem. In the last years, its high end processors provide faster calls then jumps at the expense of higher transistor count. In a processor like *picoVersat*, the problem is magnified since no stack specific registers or opcodes are available.

The present compiler detects when a routine accesses no arguments or locals and does not emit frame pointer code. So, if a routine only uses global variables, the call becomes a bit more efficient. Some of the tests used become upto 5% faster by removing the frame pointer in routines where it not needed.

As any routine can be called many times, even recursively, the compiler must save, at the beginning, and restore, at the end, all the registers the routine uses. This means that, at the start of the program, the `main` routine will spill all registers it will use, although they have no defined value. Such procedure is required since the routine may be recursively called. However, in most cases, the `main` routine is only invoked once, at the start of the program. The NOSAV environment variable can be set if the `main` routine is not used recursively and no registers will saved by the compiler. This special hack can be dangerous to use, but it makes `main` based programs more efficient.

The *picoVersat* controls *Versat* by setting specific values to predefined memory positions. The use of a routine to perform such a task is a very expensive way to change memory positions, either through `asm` directives or standard **C** code, as the tests `set.c` and `setvar.c` show, respectively. Memory values can be efficiently changed by assigning to a pointer `*addr=val` (see Limitations, above).

During this work, the *picoVersat* evolved. The use of a single memory, for program code and data, removed the need for a `addi MEM_BASE` instruction for each variable load and store, resulting in a 5% improvement over all the regression test in use, at the time.

Finally, the compiler some times generates a register read after the same register was written by another instruction selection. At least, the read can be suppressed, but **lcc** provides no peephole optimizer for final code cleanup.

VII. Conclusion

In this paper we have described the development of a complete **C** language compiler for the *Versat* architecture. The development work first addressed the support for *picoVersat* as an **lcc** compiler *back-end*. The `asm` extension was added to **lcc** to provide direct control over *Versat* and *picoVersat*. Integration of the compiler with **lcc**'s own preprocessor (**cpp**), the *Versat* assembler (**va**) and the *Verilog* simulator (**iverilog**) was added for a smooth compilation of examples from source to execution. A large set of regression tests ensure that future changes do not compromise existing functionality. Then mechanisms were added for the control of the *Versat* CGRA, using **C** language structures and assembler macros.

The resulting compiler provides all the requirements initially set out for this work as refered in section I-B.

Future work should essentially address some of the limitations refered in section VI-B, namely *bits-per-byte*, wide integers and floting point numbers. To allow `sizeof(char)` to be 32-bits wide significant changes had to be made to the core of the **lcc** compiler. Support for 64-bit wide integers, known as `long long` in **C**, as well as floating point, `float` and `double` data types, would require all operations to be performed by software routines, since *picoVersat* only handles 32-bit integers.

References

[1] Salvatore M. Carta, Danilo Pani, and Luigi Raffo. Reconfigurable coprocessor for multimedia application domain. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 44(1-2):135–152, 2006.

[2] 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.

[3] Hochan Lee, Mansureh S. Moghaddam, Dongkwan Suh, and Bernhard Egger. Improving energy efficiency of coarse-grain reconfigurable arrays through modulo schedule compression/decompression. *ACM Trans. Archit. Code Optim.*, 15(1):1:1–1:26, March 2018.

[4] 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.

[5] Rui Santiago. Compilador para a arquitectura reconfigurável versat. Master's thesis, Instituto Superior Técnico, 2016.

[6] 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.

[7] João Dias Lopes. Versat, a compiler-friendly reconfigurable processor-architecture. Master's thesis, Instituto Superior Técnico, 2017.

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 2 edition, 2006.

[9] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1 edition, 2003.

[10] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

[11] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg: Fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, April 1992.

[12] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, September 1992.

[13] Todd Proebsting. Burg, iburg, wburg, gburg: So many trees to rewrite, so little time. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-based Programming*, RULE '02, pages 53–54, New York, NY, USA, 2002. ACM.

[14] David R. Hanson and Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation*. John Wiley and Sons, New York, NY, USA, 1 edition, 1995.

[15] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. *Coarse-Grained Reconfigurable Array Architectures*, pages 449–484. Springer US, Boston, MA, 2010.

[16] M. A. A. Tuhin and T. S. Norvell. Compiling parallel applications to coarse-grained reconfigurable architectures. In *2008 Canadian Conference on Electrical and Computer Engineering*, pages 001723–001728, May 2008.

[17] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10296–, Washington, DC, USA, 2003. IEEE Computer Society.