# Developing a Retargetable Compiler: Some Preliminary Results

Elena Panainte, Koen Bertels, and Stamatis Vassiliadis
Computer Engineering Laboratory, Electrical Engineering Department, Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands
Phone: +31 (0)15 27 83644, Fax: +31 (0)15 27 84898
E-mail: elena@ce.et.tudelft.nl

*Abstract*—**The current paper reports on the first results of building a retargetable compiler for reconfigurable computing. The discussed research is part of a larger project whose main objective is to develop a semi-automatic tool platform for reconfigurable computing supporting a fully integrated design environment. It constitutes the first attempt to provide a workbench that will cover the entire design trajectory of general-purpose augmented processors with reconfigurable computing parts. The scope of the proposed platform regards processors that intend to speed-up single program execution using reconfigurable hardware. Consequently, the Delft Workbench platform targets architectures with single program counters (uni-processors) incorporating reconfigurable co-processing. Based on a particular architectural paradigm we use a SET / EXECUTE function scheduling platform and we provide appropriate modifications to the compiler back-end so that the SET / EXECUTE can be incorporated.**

*Keywords*— **retargetable compiler, reconfigurable system, IR extension.**

## I. Introduction

Computing devices become increasingly present in different areas of the human world, having as an immediate consequence that a wide array of functional behavior has to be supported by such devices. Designers are furthermore confronted with the need to make their device not only small but also more powerful as the requirements increase. During the design process, they have to identify what application or part of an existing application can be hardwired in order to obtain maximal performance. There is no fundamental trade-off between hardware and software but only a choice to determine what functions are implemented at what level.

Reconfigurable computing workbenches comparable to the workbenches for well-established technologies are rather primitive and they support specific parts of the design process. While the vendors provide workbenches and software tools to perform some intermediate phases of the mapping process, there is a lack of tools supporting other important stages of the design process as well as the integration of all the phases into a single workbench tool.

The current paper reports on the first results of building a retargetable compiler for reconfigurable computing. The discussed research is part of a larger project - Delft WorkBench - whose main objective is to develop a semi-automatic tool platform for reconfigurable computing supporting a fully integrated design environment. It constitutes the first attempt to provide a workbench that will cover the entire design trajectory of general-purpose processors augmented with reconfigurable computing parts. The scope of the proposed platform regards processors that intend to speed-up single program execution using reconfigurable hardware. Consequently, the Delft Workbench platform targets architectures with single program counters (uni-processors) incorporating reconfigurable co-processing. Based on a particular architectural paradigm we use a SET / EXECUTE function scheduling platform and we provide appropriate modifications to the compiler so that the SET / EXECUTE can be incorporated.

The paper is organized as follows. In Section II we explain the issues concerning appropriate code scheduling that accommodates reconfigurable functions to co-exist with general-purpose code. Specific attention is given to the architectural changes and the machine description information that are required for an efficient use of reconfigurable components. We then discuss the underlying infrastructure of our retargetable compiler that is based on the SUIF (Stanford University Intermediate Format) compiler and the Machine SUIF framework. This combination allows us to easily add new passes and transformations and also to extend the intermediate representation of the compiler. For the purpose of integrating SET/ EXECUTION functionality, we first extend the compiler front-end with a pass that recognizes the context for such instructions. The next step is to extend the intermediate representation with these functions. To this purpose, we modify the pass of lowering the high-level SUIF representation to a medium-level representation for a virtual machine. We also determine the back-end modifications in order to support the new functionality. We illustrate in detail a number of typ-

ical situations such as function calls and multiple parameters passing and show how they are currently addressed. We conclude by offering an overview of the remaining problems and by presenting the next steps in extending the complexity of the compiler back-end.

## II. Reconfigurable Computing and the MOLEN Architecture

The Delft WorkBench Project addresses one of the new technologies gaining increasingly wider acceptance in both the academic and industrial world, namely reconfigurable processing. It is considered to be a viable alternative to pure hardwired systems. Reconfigurable hardware co-existing with a core processor has been proposed as a good candidate for speeding up processor performance, see for example [4], [5]. Such an approach can be very promising; however, as indicated in [4], the organization of such a hybrid processor can be viewed mostly as an open topic. In most cases the hybrid organization assumes the general-purpose paradigm. In such organization, it is assumed that the processor operates in "ordinary processor environments" and is extended by reconfigurable unit(s) that speed-up the processing when possible. The execution and the reconfiguration are under the control of the "core" processor. Furthermore, due to the potential reprogrammability of the reconfigurable processor, a high flexibility is assumed in terms of programming resulting in tuning the reconfiguration for specific algorithms [6] or for the general-purpose paradigm [4]. An "intermediate" approach is proposed by considering specialized multimedia processing and by assuming that complex and expensive functions are implemented in hardware that have common computational blocks. This potentially allows fast, coarse and flexible reconfiguration for specialized multimedia processing.

In our project we assume the machine organization as proposed in the MOLEN project[1], where the configuration of the hardware and the execution on the reconfigured hardware is performed by $\rho$-code, an extension of the classical microcode for reconfiguration and execution of the resident and not-resident microcode. This organization is illustrated in Fig. 1, where the arbiter decides which instructions are for core processing (CP) unit and which for reconfigurable unit. The reconfigurable unit contains a $\rho\mu$ -code unit and a custom computing unit (CCU) which may be implemented on a Field-Programmable Gate Array (FPGA). The instructions for the reconfigurable unit are of two types: SET and EXECUTE. Loading a new configuration into a reconfigurable unit is performed under the command of a SET instruction, while EXECUTE instructions launch the operations performed by the com-

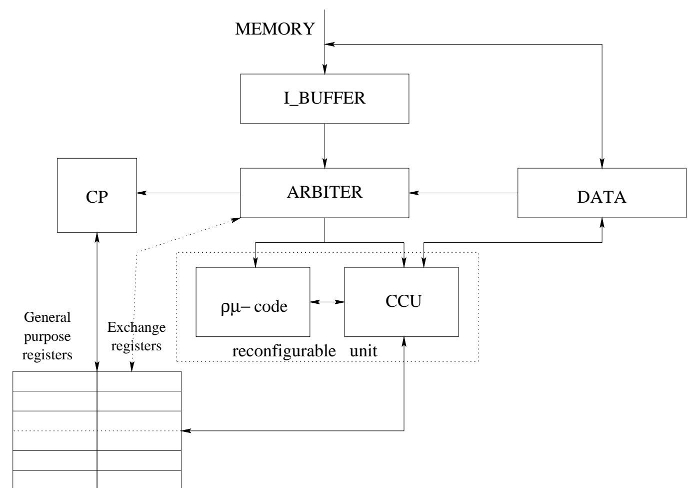


Fig. 1. The MOLEN machine organization

puting resources configured on the raw hardware. In this way, the execution of a reconfigurable unit-mapped operation requires two basic stages: SET and EXECUTE. In the MOLEN approach, the SET instructions are divided in p-SET(partial SET for common functions) and c-SET(complete SET for less common functions). For the moment, in our project we consider a SET instruction as a c-SET instruction, for which the whole configuration is performed at the program execution time. The instruction format for SET and EXECUTE contains an opcode and one bit that indicates if the location of the microcode is resident/pageable, while the last field is reserved for the address of the first instruction of the microcode. An important element in this approach is the use of the exchange registers. It is evident that data will pass between the CP and the FPGA. In order to deal with the issues of parameter passing, the exchange registers are used as communication gates. Therefore, the FPGA does nor directly store data in the main memory or other store locations.

As an example of the MOLEN approach, we consider paper [2] describing a mixed hardware and reconfigurable TriMedia processor. TriMedia is a 5 issue-slot VLIW processor optimized with respect to media processing. The TriMedia processor will be augmented with a Reconfigurable Functional Unit (RFU) that consists mainly of an FPGA core. In order to use the RFU, the new instructions are provided: SET and EXECUTE. With these new instructions, the programmer is given the freedom to define and use any computing facility subject to the FPGA size and TriMedia organizations. The potential of the FPGA-augmented TriMedia has to be evaluated within the multimedia-processing domain. Several functions related to MPEG decoding task have been considered so far and the improvements of the FPGA-augmented TriMedia versus standard TriMedia ranges from 25 to 43%.

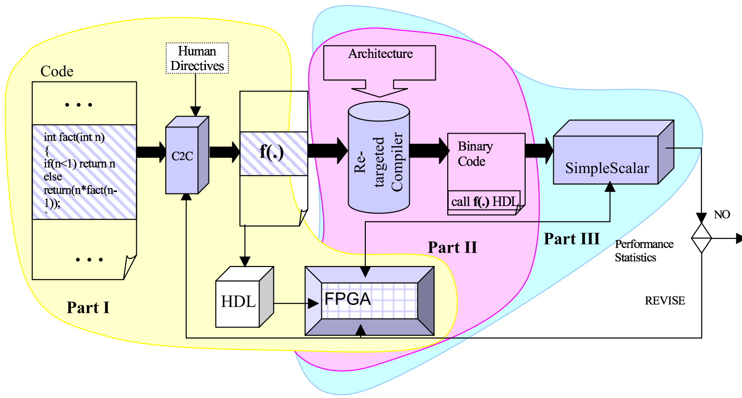


Fig. 2. The overall workflow of the Delft WorkBench

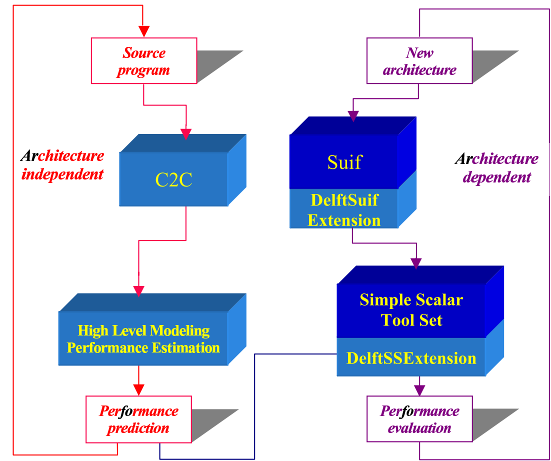


Fig. 3. Delft WorkBench software tools scheme

## III. THE RETARGETABLE COMPILER

The development of architectural improvements or innovations is a very complex process as it deals with a large number of highly interconnected factors. An improvement in one component does not necessarily result in an improved system performance. This complexity increases considerably as heterogeneous architectures (e.g. ASIC, FPGA's) are included. Such an approach is becoming increasingly popular as it allows developers to better partition and manage their projects. Facing such complexity, designers need tools to make the challenges manageable. Currently no such tools, known as workbenches, exist. There are only workbenches for fragments of the complete design trajectory and certain parts of the design process completely lack support.

On the basis of existing workbenches, it is clear that the overall time and development cost can be substantially decreased as testing and simulating a "system in a workbench" is much cheaper and faster than testing a "system on a chip". The ideal workbench supports the complete design cycle comprising, among other things, the following : it should assist the developer to identify functions the target architecture will support. There are always multiple candidates for hardwired implementation and a choice has to be made. Consequently, candidate functions need to be evaluated in a detailed way. This can be done through simulation where, in a first phase, the function will be considered separately. After this, especially in the case of heterogeneous architectures, a more complete simulation can be performed. Such a large scale simulation requires the execution of benchmarked programs.

In Fig. 2, we present the Delft WorkBench workflow and the main design phases it aims to support. The goal of the first stage (Part I) is to assess to what extent hardwired functions would increase the overall performance and what the cost is to build them. One way to support this process is to offer C2C functionality where execution traces of the program can be collected and analyzed. This gives the designer a handle to make a more than educated guess in identifying pieces of code. Factors such as number of times a function is called, total execution time etc. are taken into consideration. The end result of this phase is an overview of different pieces of code, which can be implemented at the hardware level; the tool that performs this task is called the *profiler*.

Once the choice has been made and a function f(.) is identified, the code containing the f(.) logic needs to be eliminated from the original source code and replaced by an appropriate FPGA call (Part II). The instruction set needs to be extended with the appropriate instructions for setting up the FPGA and for starting its computation. To this purpose an interface needs to be developed between the software program and the FPGA implementation of f(.). This boils down to modifying the compiler as to include the appropriate function calls that set up the FPGA, transfer the required parameters, and receive the result(s) of the execution. The retargeted compiler can then generate the appropriate machine code of the original program containing the f(.) function call. The main issue in this stage is the automatic rather than the time consuming and difficult manual modification of the target compiler that takes into account the new system properties. The current paper is focused on these issues.

The obtained machine code can then run on a simulator such as the SimpleScalar tool (Part III). The simulation will provide detailed statistical information on the overall performance of the proposed architecture. This evaluation will have to take into account all factors determining the performance such as the set-up time of the FPGA and make for instance recommendations on where to place the set-up functions.

The tools involved in the Delft WorkBench are presented in Fig. 3. The compiler system relies on the Stanford SUIF 2 Compiler Infrastructure for the front-end component, while the back-end component and the machine-dependent optimization passes are built over the framework offered by the Harvard Machine SUIF Compiler Infrastructure.

The SUIF system is a compiler infrastructure designed to support collaborative research and development of compilation techniques, based upon a program representation also called SUIF (Stanford University Intermediate Format) [7]. This system provides the necessary framework for developing new compiler passes, also allowing flexible interaction between the already existing modules. Another important feature is the extensible program representation that allows the users to capture and implement new program construct semantics. The extensions are specified in a high-level language that insulates the users from the implementation details.

Machine SUIF is a flexible and easy to extend infrastructure for developing compiler back-ends [8]. This system works in conjunction with the SUIF2 compiler infrastructure and follows the same modular design, allowing the users to integrate and combine existing and new optimization and analysis passes. The current Machine SUIF distribution includes a working compiler that is capable of producing code for machines based on the Alpha architecture, x86 and IA64.

Machine SUIF is designed for a wide range of users, including those interested in adding support for a new or augmented target architecture, developing new optimizations that are parameterizable with respect to the compilation target or developing a new intermediate representation while preserving all the existing optimization passes.

Both SUIF and Machine SUIF are used in numerous research projects (see [7]). They are well documented, reliable and continuously improving.

## IV. Compiler Extensions for SET/EXEC Functionalities

In this section we describe the interface between the profiler, which selects the C code sections that are candidates for mapping in hardware and the extended compiler supporting these new functionalities. We also discuss the special cases we encounter and present the manner we treat them.

### A. *Profiler-Compiler Interface*

After the profiler identifies the C code segments to be mapped in hardware, the compiler has to produce special machine code that includes the target SET/EXECUTE instructions; the generated code is passed to the simulator which measures the actual improvement gained by this mapping. To this purpose, the profiler has to delimit and stamp the selected C code with information revealing the FPGA configuration associated to the current functionality; the required information will be discussed later in this section. There are several different methods to introduce these stamps into the C language. One of them is to extend the C language with new keywords or to overload the existing syntax in C. Another possibility is to provide dedicated libraries, which are not portable and have to be manually extended for any new configuration.

```
#pragma call_fpga abs_int
int f(int a, int b)
{
  int c;
  c=a-b;
  if(c>0) return c;
  else    return -c;
}
```

Fig. 4. Pragma annotation

We use the ”pragma” directives to inform the compiler about the C code that has to be mapped in hardware. This approach has the advantage that the program remains in standard C and it may be tested with standard C compilers. The compilers that don't recognize the pragmas will simply ignore them and generate machine code for the associated standard C code, while our compiler will ignore the C code and generate SET/EXECUTE instructions for the associated FPGA configuration.

The C code that is mapped in hardware can be a whole function or a part of a function (e.g. loops). We currently restrict ourselves to the functions; the other case is similar to this one and its implementation is straightforward, due to the powerful mechanism provided by the SUIF annotations.

For example, if the profiler designates a function **f** that computes the absolute difference of two integers to be mapped on the FPGA with the associated context name **abs_int**, then it introduces the pragma as presented in Fig. 4. The **call_fpga** pragma parameter informs the compiler that the current function has to be mapped in hardware and the last parameter represents the name for the FPGA configuration that implements the current function; the body of the function remains untouched.

For the case where only a piece of code has to be mapped in hardware, it has to meet the intuitive and natural condition that it can be included in a new scope statement. For example, for the piece of code presented in Fig. 5 the

```
a=b+c;
for(int i=0;i<10;i++){
  a=a+i;
  b=b*i;
  c=c-i;
}
```

Fig. 5. Piece of code

```
void main()
{
  int x,z;
  x=5;
  z=f(x,7);
}
```

Fig. 6. Call of the f function, presented in Fig. 4

first tree lines - which are *a piece of code* - cannot be annotated with pragma since the rest of the code is detached from the inner scope statement contained in the *for* loop. Instead, all lines or just the inner lines of the *for* loop can be annotated. The chosen code is separated into a new scope statement and the pragma annotation is added in front of it. The annotations are similar to the annotations for functions, with some additional parameters - these are beyond the scope of this paper.

We assume that a function with an FPGA annotation has an associated representation in the FPGA description file, where at least the following characteristics are specified:

- the name of the FPGA context: this has to match the pragma's second parameter.
- the FPGA input and output registers.
- the address of the microcode for SET.
- the address of the microcode for EXECUTE.

Returning to the example presented in Fig. 4, the FPGA context description may look as in the following lines:

*abs_int*: XR2 ← XR3, XR7
SET: xxx...xxx
EXECUTE: xxx...xxx

We assume as well that the function and the associated FPGA context have the same number of input and output parameters and the order of the parameters is the same. These assumptions do not limit the generality of the functions mapped in hardware, as the process of manually mapping on reconfigurable architectural components starts from the C code of the functions identified by the profiler. For other projects based on our compiler, where the FPGA context is predefined, a manual intervention on the C code is required. Its complexity is determined by the similarity between the new C code and the FPGA functionality.

### B. Compiler Extension

After the profiler introduces the above presented pragma directives into the original C code, the modified C code is passed to the compiler. In SUIF, the pragma directives are expressed as annotations to the associated scope statements. It is important to interpret the annotations as soon as possible in the compiler flow, in order to prevent modifications in the annotated piece of code or in the code that interacts with it. Therefore, we introduce a special pass in the SUIF front-end that detects the functions that have *call_fpga* pragmas and introduces annotations with the pragma parameters into the code that calls these functions. For pragmas associated to pieces of code, this pass should remove the code and add annotations with the pragma parameters to the associated scope statements. Special attention should be given for preserving the program data dependencies; a method to solve a similar problem is presented later in this section (see Fig. 8).

The pass name is *call_fpga_pass* and it is applied immediately after the program transformation in the SUIF representation. The rest of machine independent optimizations and transformations are applied after its execution.

As explained above, we extend the instruction set with two instructions SET/EXEC. We also assume that the FPGA is associated with a set of registers that are also under the control of the processor. These "exchange registers" are the only registers the FPGA has access to and the processor uses them only for communicating with the FPGA. Such registers facilitate the process of replacing one FPGA with another one and they also facilitate the instruction scheduling algorithms. Therefore, we introduce two additional instructions for inter-register communication, namely:

- MOVEGP reg_private, reg_gen - moves the value of a general register into a FPGA register
- MOVEPG reg_gen, reg_private - for the opposite direction.

After the transformation pass *call_fpga_pass*, where the functions with FPGA annotations have been identified and their calls have been marked, we have to introduce the SET/ EXECUTE instructions. The goal is to allow the compiler to apply specific optimizations for them, as they are expensive instructions that consume from hundreds to thousands of machine cycles. We choose to introduce them as soon as possible, namely in the medium intermediate representation (MIR), as these instructions work with registers. The high intermediate representation (HIR) does not at that stage take into account the registers of the target machine. Also, the SET/EXECUTE instructions are machine-independent and the low intermediate representation (LIR) is already too close to the target machine. In the Delft WorkBench compiler, the MIR is SUIFvm

```
main:
  // x=5
  mrk 2, 21
  ldc $vr1.s32 <- 5
  mov main.x <- $vr1.s32
  // f(x,7)
  mrk 2, 22
  lda $vr2.p32 <- f
  ldc $vr4.s32 <- 7
  cal $vr3.s32 <- main.x, $vr4.s32,
                     ($vr2.p32)
  //z=f(x,7)
  mov main.z <- $vr3.s32
  // return 0 - implicit
  mrk 2, 23
  ldc $vr5.s32 <- 0
  ret $vr5.s32
  .text_end main
```

Fig. 7. Original SUIFvm code generated for the C code from Fig. 6

(SUIF for a virtual machine) and it is generated by the pass from Machine SUIF *do_s2m*. Due to the power of the SUIF system that allows passes with similar functionality to coexist in the compiler structure and to choose the best one for a specific architecture in the compiler flow, we do not modify the *do_s2m* pass. Rather we create a similar pass *do_s2m_fpga* that is identical to the previous one, with modifications only for the FPGA instructions generation.

The modifications we made are related to the calls of the functions that have to be mapped in hardware. Another alternative is to operate directly on the body of the function, namely to remove it and introduce only SET/EXECUTE and their associated instructions for parameter passing. However, this alternative has the disadvantage that it insulates the SET/EXECUTE instructions in the function and it limits some important optimizations, such as moving these instructions outside the function in order to gain a better scheduling. As our approach replaces the function call with the FPGA call instructions, it also has the advantage that it eliminates the instructions for invoking a function, such as passing parameters, returning results or saving registers.

The mechanism to ask the FPGA to execute a specific task contains the next steps:

- transfer parameters, with two sub-phases:
  - move parameters in general-purpose (GP) virtual registers (if they are not already there), since the only communication between the general processor and the FPGA is through registers.
  - move GP virtual registers to FPGA virtual registers that will be mapped on the FPGA hard registers.
- SET the FPGA: the associated microcode address has to be determined from the FPGA description file.
- EXECUTE the FPGA context: the microcode address is again obtained from the FPGA description file.
- return the result: if the function type is not *void*, then the FPGA register that contains the result has to be sent to the CP and is therefore stored in a general register.

```
main:
  // x=5
  mrk 2, 21
  ldc $vr1.s32 <- 5
  mov main.x <- $vr1.s32
  // begin FPGA call
  // abs_int : XR2 <- XR3, XR7
  //           z   = f(x,  7)
  mrk 2, 22
  // XR3 <- x
  mov $vr2.s32 <- main.x
  movgp   $vr3.s32($fpgaXR3 ) <- $vr2.s32
  // XR7 <- 7
  ldc $vr4.s32 <- 7
  movgp   $vr5.s32($fpgaXR7 ) <- $vr4.s32
  set ( abs_int ) xxx...xxx
  exec ( abs_int ) xxx...xxx
        $vr6.s32($fpgaXR2 ) <-
           $vr3.s32($fpgaXR3 ),
           $vr5.s32($fpgaXR7 )
  movpg   $vr7.s32 <- $vr6.s32($fpgaXR2 )
  //end FPGA call
  mov main.z <- $vr7.s32
  // return 0 - implicit
  mrk 2, 23
  ldc $vr5.s32 <- 0
  ret $vr5.s32
  .text_end main
```

Fig. 8. SUIFvm code extended with SET/EXECUTE instructions for the C code from Fig. 6

For the function presented in Fig. 4, and the call presented in Fig. 6, we present in Fig. 7 the original SUIFvm code for the *main* procedure - without taking into account the pragma *call_fpga*, in order to emphasize the modification we made. The generated code for the proper integration of reconfigurable components is presented in Fig. 8.

A SUIFvm virtual register is marked with the number, the type and the length, such as $vr7.s32, and one with an associated specification, for example $vr6.s32($fpgaXR2) contains an annotation to inform the register allocator to map it on the corresponding FPGA register (its number is included in an annotation and it is obtained from the FPGA description file).

An illustration of the above presented flow is presented in Fig. 9, where x is a variable stored in memory. In order to send its value to the FPGA, it is first moved in a

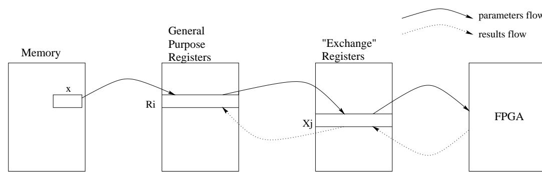


Fig. 9. Data flow to the FPGA

general purpose register (Ri), then it is sent into an ”exchange register” Xj that is accessed by the FPGA. In the reverse direction, one result computed by the FPGA is first moved into one of the exchange registers and sent to the core processor via the general purpose register. The described mechanism defines a clear interface between CP and FPGA, which allows easily replacing each of them and maintaining unmodified the other component. It also provides the compiler with the possibility of a good control for both of them.

In order to preserve the program data dependencies for the following compiler passes, we also introduce sources and destinations in the EXECUTE instructions. For a general approach, we consider all the function parameters as its sources and if the function type is not *void* we consider that the EXECUTE instruction also has one destination. A more detailed presentation of possible cases is presented in the next subsection.

### C. Particular Cases

**Multiple Results**

In an usual function call, the function is required to return more than only one value. As the code that has to be mapped in hardware, must first be expressed in the C language, where parameters are sent by values and the function syntax admits only one return value, we adopt a solution similar to the one available in the C language. As passing a pointer to a function allows that function to modify the outside object, we send as function parameters pointers to variables that will contain after the function execution the return values. Thus, the EXECUTE instructions have to incorporate these dependencies as well and mark the pointer parameters used to modify their associated memory locations. For this purpose, the EXECUTE instruction wraps these pointer parameters into its destinations. This is possible as SUIF and Machine SUIF support instructions with multiple destinations. The best approach - which we will adopt further, is to explicitly specify the pointer parameters that are used to return the desired values in the FPGA description file, as those that implement the FPGA know the correct significations of each parameter. We also assume that if a function uses this modality to return many results, than its type is *void*.

```
#pragma call_fpga add_dif
void f(int a, int b, int *c, int *d){
  *c=a+b;
  *d=a-b;
}

int main()
{
  int z,m,n;
  z=5;
  f(z, 21, &m, &n);
}
```

Fig. 10. Function that computes two values

```
main:
  // z=5
  mrk 2, 10
  ldc $vr1.s32 <- 5
  mov main.z <- $vr1.s32
  mrk 2, 11
  // begin FPGA call
  // add_dif: XR2, XR5 <- XR3, XR7, XR2, XR5
  //                     f( z,   21,  &m, $n)
  mov $vr2.s32 <- main.z
  movgp   $vr3.s32($fpgaXR3) <- $vr2.s32
  ldc $vr4.s32 <- 21
  movgp   $vr5.s32($fpgaXR7) <- $vr4.s32
  lda $vr6.p32 <- main.m
  movgp   $vr7.s32($fpgaXR2) <- $vr6.s32
  lda $vr8.p32 <- main.n
  movgp   $vr9.s32($fpgaXR5) <- $vr8.s32
  set ( add_dif ) xxx...xxx
  // mark the modifications for m and n
  exec ( add_dif ) xxx...xxx
     $vr7.s32($fpgaXR2),$vr9.s32($fpgaXR5) <-
       $vr3.s32($fpgaXR3), $vr5.s32($fpgaXR7),
       $vr7.s32($fpgaXR2),$vr9.s32($fpgaXR5)
  //end FPGA call
  mrk 2, 12
  ldc $vr6.s32 <- 0
  ret $vr6.s32
  .text_end main
```

Fig. 11. SUIFvm extended with SET/EXECUTE for the C code presented in Fig. 10

For an illustration of this case, consider the C program annotated by the profiler, from Fig. 10. The generated MIR should be as in Fig. 11.

**Pointer Invoked Functions**

Another problem we encounter is the situation where a function is invoked through a pointer to it. We consider that for this case, the SET/EXECUTION instructions should not replace the classical mechanism for function call, as a pointer analysis is required and there are cases when only at the execution time, it is determined if a

```
int (*p)(...);
if(C)
  p=f
else
  p=g;
(*p)(...)
```

Fig. 12. Pointer to functions with or without FPGA annotations

pointer is related to a normal function or to a function that has to be mapped in hardware. Such a case is presented in Fig. 12, where f is a function that is implemented on the FPGA, g is a function executed on the CP and having the same prototype as f, and C is a condition whose value is determined at the execution time. We cannot determine at the compilation time if the call of p is directed to f - which should be replaced by SET/EXECUTE instructions, or to g - which is a normal call. In consequence, the call of p is treated in the classical way.

Special attention has to be paid for the situation in which a function parameter is the result of a function that is also mapped on the FPGA, as the result may reside in one FPGA register and the parameter is expected in another FPGA register. Consider the function presented in Fig. 4 and the call *f( f(a,b),c)*, with the FPGA context description previously assumed. The first parameter of *f* has to be in the FPGA register XR3, while the result of *f* is put in the FPGA register XR2. In order to solve the problem, the result of the inner call of *f* is moved to a general register and then moved again to the proper FPGA register.

## V. Conclusions and Further Research

In this paper we presented the tools and components involved in Delft WorkBench project that aims to support the design space exploration for reconfigurable computing. We focused on compiler extensions and we first defined an interface between the profiler and the compiler. We then described the compiler modifications based on pragma annotations and detailed the IR extension with two instructions - SET/EXECUTE , that control the reconfigurable unit. We also proposed and implemented the communication mechanism between CP and RC unit.

One of the open issues is to allow the mapped C code to be just a segment of code and not a whole function. To this purpose, the FPGA description file has to be extended with information revealing the use of each expected parameter. Furthermore, the pragma annotation has to include the mapping between the actual parameters and the generic parameters, and finally, the mechanism to call the FPGA also has to be extended in order to save each modified value at the end.

Finally, additional research is required to complete the compiler back-end focusing on register allocation and code optimization issues.

## References


[1] Vassiliadis, S., Wong, S., Cotofana, S., "The MOLEN $\rho\mu$-coded Processor," Tech. Rep. 1-68340-44(2001)-01, Computer Engineering Laboratory, TUDelft, Netherlands, 2001.

[2] Sima, M., Cotofana, s., Vassiliadis, s., Eijndhoven, Jos T.J., Vissers, Kees A., "MPEG Macroblock Parsing and Pel Reconstruction on an FPGA-augmented TriMedia Processor," Proceedings of the International Conference on Computer Design (ICCD 2001), Austin, Texas, September 2001.

[3] Muchnick, S., " Advanced Compiler Design and Implementation," Morgan Kaufmann Publishers, 1997.

[4] Hauser, J.R., Wawrzynek, J., "Garp: a MIPS Processor with Reconfigurable Coprocessor," IEEE Symposium on FPGAs for Custom Computing Machines, NAPA Valley, California, pp.92-100. 1997.

[5] Razdan, R., Smith, M. D., "A High Performance Microarchitecture with Hardware-Programmable Functional Units," 27th Annual International Symposium on Microarchitecture, San Jose, California, pp.172-180, 1994.

[6] Bitter, R.A., Athanas, P.M., "Wormhole Run-time Reconfiguration," Proc. 5th International Symposium on Field Programmable Gate Array, Monterey, California, pp.79-85, 1998.

[7] http://suif.stanford.edu/

[8] http://www.eecs.harvard.edu/hube/research/machsuif.html