An optimizing C front-end for hardware synthesis

Alberto La Rosa, Luciano Lavagno, Mihai Lazarescu, Claudio Passerone

Politecnico di Torino
Corso Duca degli Abruzzi 24
10137 Torino, Italy
alarosa@gandalf.polito.it
{luciano.lavagno, mihai.lazarescu, claudio.passerone}@polito.it

Abstract- Modern embedded systems must execute a variety of high performance real-time tasks, such as audio and image compression and decompression, channel coding and encoding, etc. High hardware design and mask production costs dictate the need to re-use an architectural platform for as many applications as possible. Reconfigurable platforms can be very effective in these cases, because they allow one to re-use the architecture across a variety of applications.

The efficient use of a reconfigurable platform requires a methodology and tools supporting it in order to extensively explore the hardware/software design space, without requiring developers to have a deep knowledge of the underlying architecture, since they often have a software background and only limited hardware design skills.

This paper describes a tool that fits into a complete design flow for a reconfigurable processor and that allows one to efficiently transform a high level specification into a lower level one, more suitable for synthesis on the reconfigurable array. The effectiveness of the methodology is proved by a complete implementation of a turbo-decoder.

I. Introduction

Reconfigurable computing is emerging as a promising means to tackle the ever-rising cost of design and masks for Application-Specific Integrated Circuits (ASIC). Adding a reconfigurable portion to a Commercial Off The Shelf IC enables it to potentially support a much broader range of applications than a more traditional ASIC or microcontroller or DSP would. Moreover, *run-time reconfigurability* even allows one to adapt the hardware to changing needs and evolving standards, thus sharing the advantages of embedded software, with higher performance and lower power than a traditional processor, thus partially sharing the advantages of custom hardware.

A key problem with dynamically reconfigurable hardware is the inherent difficulty of programming it, since neither the traditional synthesis, placement and routing-based hardware design flow, nor the traditional compile, execute, debug software design flow directly support it. In a way, dynamic reconfiguration is a hybrid between software, where the CPU is "reconfigured" at every instruction execution and memory is abundant, and hardware, where reconfiguration occurs seldom and very partially, and memory is a scarce resource.

In developing a reconfigurable architecture, a key aspect that deeply impacts its usability and success is who will be using it. The designer must clearly identify the architecture's target users to envision a design methodology that optimally fits their needs. We consider two main user categories:

1. *Hardware-oriented* users have the hardware knowledge necessary to program a reconfigurable unit and extract maximum performance from it. They use software on a microprocessor for flexibility, but significant portions of the functionality are implemented as hardware co-processors. They seek complete control over the implementation, also having higher design and fabrication costs.

2. *Software-oriented* users seek a high level abstract programming model that hides architectural details through an optimizing compilation tool chain. They are typically not able to perform detailed hardware design and are mostly interested in speeding up their software in some computation-intensive kernels, by mapping part of them to the reconfigurable hardware unit. However, they would prefer an automated tool with limited user intervention, at the cost of lower performance gains.

The architectures that derive from the two use cases above are profoundly different in the way the processor and the reconfigurable unit are coupled. In the first one, they are separated and communicate using some standard or dedicated bus, much like co-processors (e.g. [10] and several commercial examples from Altera, Triscend and Xilinx). The overhead due to the communication is balanced by implementing long computations, corresponding to complete functional blocks, on the reconfigurable hardware, and by limiting their interaction. In the second case, the reconfigurable unit can be tightly coupled with the processor, becoming a new function unit in the datapath (e.g. [11–13]). Communication is performed through multi-port register files that are concurrently accessed by both the reconfigurable and the traditional functional units of the processor. Kernels that are implemented in hardware are typically smaller that in the hardware oriented case, and can be considered as dynamic extensions of the processor instruction set.

We will concentrate on the software-oriented use, where the reconfigurable unit is part of the processor data-path. In collaboration with a design group at the University of Bologna, we developed a complete design flow, including methodology and compilation tool chain, addressing the instruction set HW/SW codesign problem for the XiRisc architecture, that was designed by that group and is described in [3]. The processor is based on a 32-bit RISC (DLX) reference architecture. However, both the instructions set and the architecture have been extended to support:

- Dedicated hardware logic to perform multiply-accumulate calculation and end-of-loop condition verification, by using special DSP-like instructions added to the standard DLX ISA.

- A double data-path, with concurrent VLIW execution of two instructions. The added data-path is limited to only arithmetic and logic instructions, and cannot directly access the external memory subsystem.

- An FPGA, called PiCoGA, to implement in hardware special kernels and instructions. Each cell may store up to 4 configurations (called layers) that can be switched in one clock cycle when needed. The FPGA can be dynamically reconfigured at run-time without stopping its operation, as long as the layer which is being reprogrammed is not used.

The flow, presented in [14], starts from a system level specification (usually as a software program) of the application, and partitions it into software and hardware domains to achieve the best performance in terms of speed and power, while satisfying constraints imposed by the limited resources available on the target platform architecture.

For simulation and profiling purposes, the entire system specification is considered as a software program that goes through a standard compilation flow, leveraging available and well established tool chains used in software development. However, the tool chain has been modified so that the hardware domains that have been identified during partitioning can be especially treated to take into account their performance. Different solutions can be quickly evaluated to explore the design space.

For synthesis purposes, while the software domain can be compiled with the very same tool chain used during design space exploration, the specification of those portions that are to be downloaded onto the PiCoGA must be transformed into a specification that is more suited for logic synthesis, placing and routing. The XiRisc processor already has a hardware synthesizer that, starting from a description given in a simplified subset of C, called Griffy C [1], generates a bit stream to program the PiCoGA. However, it lacks an automated and efficient tool to map a high level specification into a Griffy C program, taking into account the restrictions and the semantic rules of the language. The goal of this paper is to present one such tool.

The paper is organized as follows: in section II the Griffy C back-end compiler is briefly introduced. Section III presents the front-end tool. Experimental results are shown in section IV and finally section V concludes the paper.

II. Griffy back-end compiler

The Griffy back-end compiler [1] extracts and maps a pipelined DFG from the input program on the FPGA unit of the XiRisc, called PiCoGA [2–4]. The Data Flow synchronization is handled directly by the PiCoGA control unit, using explicit stage enable signals. It reduces the amount of hardware resources required to implement a fragment of high-level code, and reduces the complexity of the mapping phase.

Griffy C is the input language of the Griffy back-end compiler. It is a subset of ANSI C, which supports only the PiCoGA-synthesizable subset of the C operators. At the same time, it enhances C with new semantic rules aimed to simplify the mapping onto the PiCoGA.

Griffy C is based on three basic assumptions:

- *DFG-based description*. No control flow (branch or function calls) are supported, only conditional assignments (*?:*) implemented using standard multiplexers;

- *single assignment*. Each variable is assigned exactly once to easily resolve hardware connection;

- *operation dismantling*. Only single operator expressions are supported.

Variables (static and dynamic) can use any standard C integer type, while their bit size can be directly specified using specific *#pragma* directives. Operator width is derived automatically from operand sizes.

Griffy C can be manually written at different levels of abstraction or automatically generated by an optimizing compiler from a high level description of the program. The latter flow is detailed in section III.

III. The optimizing C front-end

The quality of the Griffy-C synthesis to the PiCoGA depends much on the quality of the Griffy C source. However, the conversion of the C source to good Griffy C is often tedious and error prone. Moreover, the conversion should be repeated whenever the C source changes, for example to investigate different HW/SW partitions.

The optimizing C front-end, *gcfe*, that is described in this paper aims to provide a path for an automatic, good quality conversion process from C to Griffy C. The front-end is based on *Impact*, an optimizing C compiler [5], whose scalar software optimizations are used virtually unchanged. The transformations and optimizations specific for the generation of Griffy C follow, because they transform the Intermediate Representation (IR) of the program up to the point that it is no longer compatible with most Impact software optimizations.

The following sections will present briefly the main processing steps for IR conversion to Griffy C, in the order they are applied: calling convention bypass, conversion to static single assignment (SSA), source- and destination-propagation, IF-to-MUX conversion, Look-Up Table (LUT) generation, and size calculation for variables.

A. Calling convention bypass

This step attempts to remove from the IR all artifacts related to target calling convention, such as the code related to function arguments passed through stack, etc. New local variables may be created and their usage propagated throughout the function code in place of stack references, etc.

B. Conversion to static single assignment format

Static Single Assignment (SSA) is a representation of the program, in which every variable is assigned exactly once [6–9]. This constraint is requested by the Griffy back-end.

SSA conversion is performed by scanning in topological order all instructions looking for duplicate assignments to the same variable. Whenever a duplicate assignment is found, the assignment target is given a new name, which is then used in place of the old name for all subsequent instructions.

```
                                v = 0;
v = 0;                          if (a > 0)
if (a > 0)                          v1 = 5;
    v = 5;                      else
else                    ⟹          if (b > 0)
    if (b > 0)                          v1 = 7;
        v = 7;                      else
                                        v1 = v;
```

Figure 1. SSA conversion example: `v` is renamed to `v1`

Since the renaming on the *then* and *else* branches of conditional statements is performed independently, particular care should be taken to enforce the same exit name for both branches. This is done by setting the target of the last assignment to a given variable on each branch to the same name, which becomes the new name of that variable after the branch (figure 1). This mechanism emulates the functionality of the classical ϕ node used for such a merging by other SSA implementations. The ϕ node must be avoided here in order to preserve the IR compatibility of the modified IR with other Impact algorithms.

C. Source and destination propagation

Source propagation attempts to find out what is the real value of a variable, by traversing backward the data flow looking for an instruction that changes the value of the variable of interest. The value found replaces all subsequent uses of the variable and the variable itself gets removed later on. Source propagation will go across memory references.

Typically, Impact's optimized IR will not have such intermediate variables, unless they are required by the target. For example, this mechanism is used to remove part of the calling convention code (see figure 2).

```
r1 = 5;
...                     ⟹   f(r7, 5);
f(r7, r1);
```

Figure 2. Source propagation: `r1` replaced with its value

Note that *f()* above denotes a Griffy C intrinsic function (Griffy C does not support user function calls).

Destination propagation attempts to do a similar task by removing intermediate variables in the data flow between the place where a variable is defined and where it is used.

D. IF-to-MUX conversion

Somewhat similar to SSA, this is a complex IR transformation. It operates on SSA-compliant code and attempts to convert all branch instructions that set the same variable into a series of speculative executions followed by a MUX to select the proper value based on the truth value of the branch boolean condition (figure 3).

```
v = 0;
if (a > 0)                  r1 = a > 0;
    v1 = 5;                 r3 = 5;
else                        r2 = b > 0;
    if (b > 0)        ⟹    r4 = 7;
        v1 = 7;             r5 = v;
    else                    r6 = r2 ? r4 : r5;
        v1 = v;             v1 = r1 ? r3 : r6;
```

Figure 3. IF-to-MUX conversion example

Griffy C requires MUX inputs be variables. The innermost branches are converted first and their result(s) are used in the higher level branches, based on the scope of the variables. For clarity, the C-level names were preserved in the example code.

This transformation converts the whole program to one single basic block (BB), with speculative executions for all paths and result selection using MUXes.

E. Look-up table generation

Using Griffy C extensions many hardware-specific elements can be expressed in an efficient way. Look-up tables (LUTs) represent an effective way to store constants in the hardware.

The *gcfe* LUT generation algorithm extends the IR with specific constructs to represent the numeric constants as required by the Griffy back-end. Figure 4 depicts a simple example, showing how a vector of constants is converted to its Griffy C model, using the LUT definition (@) and concatenation (#) extensions.

```
                            char
char                        f(unsigned char i)
f(unsigned char i)          {
{                             int r1;
  static char                 int r2;
  v1[5] = {           ⟹      int P15;
      1,2,3,4,5               r1 = i @ 0x1b1;
  };                          r2 = i @ 0x5;
                              P15 = r2 # r1;
  return v1[i];               return P15;
}                           }
```

Figure 4. LUT generation example

The input variable `i` is used to select the output bits from the two LUTs, which get stored in the variables `r1` and `r2`. Next, the bits are concatenated to generate the proper output result. This example uses LUTs with two bits output, but *gcfe* is able to optimally select the type of the LUT to best map the constants in hardware.

F. Automatic size calculation for variables

gcfe provides support for freely interleaving automatic datapath size calculation with manual definition of operand sizes. To this purpose the designer can specify the size of the input and intermediate operands declared in the C source using a special `#pragma` construct. The Griffy C model

```
char f(unsigned char ia)
{
  static char v1[5] = { 1, 2, 3, 4, 5 };
  return v1[ia] + 5;
}

char f(unsigned char p0)
{
    int r4;
#pragma attrib r4 SIZE=2
    int r5;
#pragma attrib r5 SIZE=2
    int r3;
#pragma attrib r3 SIZE=4
    int P15;
#pragma attrib P15 SIZE=5

  r4 = p0 @ 0x1b1;
  r5 = p0 @ 0x5;
  r3 = r5 # r4;
  P15 = r3 + 0x5;
  return P15;
}
```

Figure 5. Datapath size definition

produced defines the size for every variable using the same notation (figure 5).

The algorithm used by *gcfe* sets the size of output variable of operations wide enough to accommodate any possible output value. This conservative approach may be occasionally too large an estimate. The size can be easily constrained to tighter packing by annotations in the C source whenever the value range is known and fits into a smaller size.

IV. Experimental results

We considered the specification of a turbo-decoder that follows the 3GPP recommendations for UMTS cellular phone systems [16]. We assume to receive a bit stream from a Recursive Systematic Convolutional encoder, with an 8 state trellis and rate equal to $1/3$ (i.e. three bits are transmitted for each bit in the message, two of them being parity bits).

Turbo-decoder high computational demands restricted its application until recently, when enough computational power has become widely available. For this same reason several source code optimizations with respect to its textbook representation are necessary to obtain an efficient and cost-effective implementation.

The flow described in this paper aims to enable a software developer to obtain an optimized implementation by profiling, optimizing the code, and finally selecting which code fragments benefit most by being implemented as PiCoGA operations.

The decoder implements an iterative algorithm, repeated until convergence is achieved. The most computationally intensive components of an iteration are: 1) trellis metrics computation (γ); 2) forward probabilities computation (α); 3) backward probabilities computation (β); 4) maximum likelihood ratio computation (LLR).

Each of the above computation kernels was manually mapped on PiCoGA. The manual mapping is described in [15] and extensively exploits the word-level parallelism, the logic optimization of combinational operators, carefully optimizes data memory allocation to minimize data fetches, and packs multiple operand in a single register. Operand and operator size was also minimized, by analyzing the required precision on output values and the available precision of input values.

The goal of this experiment was to evaluate how *gcfe* and the Griffy compiler compare to manual optimizations when implementing the turbo decoding algorithm on the PiCoGA.

For the sake of simplicity, we chose to analyze only the butterfly kernel, which is the main computational bottleneck of the turbo decoding algorithm and is mainly used in forward and backward probability computation.

The manually extracted best implementation of the butterfly kernel is able to process six 16-bit inputs packed in three 32-bit registers, producing four 16-bit outputs packed in two 32-bit registers. Operand packing uses 16 bits even when single operand precision could be smaller (e.g. 12 or 10 bits).

There is no inter-dependency among the two threads of computations, since output values are computed using two parallel data paths that share only input values.

```
alpha_in_0 = geth(in0);          //S0
alpha_in_1 = getl(in0);          //S1

a = sat_sum(alpha_in_0, gamma0); //S0 + S0/0
b = sat_sum(alpha_in_1, gamma1); //S1 + S1/1
if (abs(a - b) <= T)
  corr = ((T - abs(a - b)) >> S);
else
  corr = 0x0;
//S0 = max(S0+S0/0, S4+S4/1)
alpha_out_0 = max(a, b);
alpha_out_0 = sat_sum(alpha_out_0, corr);

a = sat_sum(alpha_in_0, gamma1); //S0 + S0/1
b = sat_sum(alpha_in_1, gamma0); //S1 + S1/0
if (abs(a - b) <= T)
  corr = ((T - abs(a - b)) >> S);
else
  corr = 0x0;
//S0 = max(S0+S0/0, S4+S4/1)
alpha_out_1 = max(a, b);
alpha_out_1 = sat_sum(alpha_out_1, corr);
```

Figure 6. Butterfly kernel of turbo-decoding.

Each thread of computation (figure 6) contains multiple instances of the saturated sum operator described later, and of the `abs` and `max` computations, specified with the ternary ?: C operator.

The exact bit size of each input value was annotated with a *#pragma* directive to allow the *gcfe* to accurately compute the required sizes of intermediate operand and operators.

The functional model obtained from *gcfe* processing was successfully compiled and tested for correct functionality. Then it was processed by the Griffy synthesizer, which produced the size and cost estimation of the automated implementation of the butterfly operator on the PiCoGA.

Table 1 presents the results of the automatic (*gcfe* and Griffy) and the manual implementation of the butterfly op-

TABLE 1. Comparison between manual and automated implementation

	Implementation	
	automatic	manual
issue delay:	2	2
input to output latency:	14	10
PiCoGA rows:	58	20

erator. Of particular importance are the close results for the issue delay and the latency, given that the manual design is very labour-intensive (several man-weeks were required for optimizations.)

Moreover, analyzing the Griffy output we identified the main issues that can improve the automatic results.

The ternary operator (?:) used in `abs` took 1 PiCoGA row for manual and 3 for automatic, resulting for the latter in an additional 6 PiCoGA rows and 3 latency clocks.

The saturated sums defined as:

```
#define sat_sum(a,b) \
  ((((int)(((a)<<16)>>16)+\
    (int)(((b)<<16)>>16))>\
    ((int)0x00007FFF))?(0x7FFF)\
  :\
    ((((int)(a)+(int)(b))<\
      (int)0xFFFF8000)?(0x8000)\
    :\
      ((a)+(b))))
```

required 1 row for manual and 3 rows for automatic (using shift and adders). Such operators are better implemented using library functions, since some C constructs are very difficult to automatically implement efficiently as combinational logic.

V. Conclusion

The results presented in this paper were obtained using the *gcfe* and Griffy compilers almost without effort, starting from a manually optimized C model. No extensive hardware design experience was required, as it was needed to obtain the manual implementation. Adding just a few *#pragma* directives to the C model lead to a very good implementation on the PiCoGA of the selected computational kernel. We consider the proposed approach as very effective, well suitable for exploring the design space that is required in the early design phases, before investing more resource on manual optimization.

The level of optimization achieved by the synthesis flow is comparable with that offered by software compilers for digital signal processors (DSP). In both cases, manual optimization can reduce cost and improve performance by factors ranging from 10 to 50%, depending on the application, the compiler, and especially the processor architecture. We find our results very comforting in this respect, firstly because the gap is not excessive and it can be further reduced. Secondly, it is quite likely that the proposed flow will find good acceptance for design space exploration and, subsequently, for complete synthesis of implementations of hardware kernel acceleration on reconfigurable processors.

Acknowledgment

The authors would like to thank Claudio Mucci, and the entire XiRisc team, for the support during this work.

References

[1] C. Mucci, C. Chiesa, A. Lodi, M. Toma, and F. Campi. *A C-based algorithm development flow for a reconfigurable processor architecture.* In *Proc. of the International Symposium on SoC*, pages 69–73, November 2003.

[2] F. Campi et al. *A Reconfigurable Processor and Software Software Environment.* International Symposium on SoC 2002, Tampere, Finland.

[3] F. Campi et al. *A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications.* ISSC 2003.

[4] A. Lodi, M. Toma, F. Campi. *A Pipelined Configurable Gate Array for Embedded Processors.* In Proc. on FPGA 2003.

[5] The IMPACT Research Group http://www.crhc.uiuc.edu/Impact/

[6] Aho, Alfred V. Sethi, Ravi & Ullman, Jeffrey D. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1988.

[7] Appel, Andrew W. *Modern Compiler Implementation in ML.* Cambridge University Press, 1999.

[8] Cooper, Keith D. & Torczon, Linda. *Engineering a Compiler.* Morgan Kaufmann, 2005.

[9] Muchnick, Steven S. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[10] T. Callahan, J. Hauser, and J. Wawrzynek. *The Garp architecture and C compiler.* IEEE Computer, 33(4):62–69, April 2000.

[11] R. Razdan and M. Smith. *A high-performance microarchitecture with hardware-programmable functional units.* In Proc. of the 27th Annual International Symposium on Microarchitecture, November 1994.

[12] S. Hauck, T. Fry, M. Hosler, and J. Kao. *The Chimaera reconfigurable functional unit.* In Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1997.

[13] B. Kastrup, A. Bink, and J. Hoogerbrugge. *ConCISe: A compiler-driven CPLD-based instruction set accelerator.* In Proc. of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, April 1999.

[14] A. La Rosa, L. Lavagno, C. Passerone. *Software Development for High-Performance, Reconfigurable, Embedded Multimedia Systems.* IEEE Design & Test of Computers, 22(1):28–38, Jan-Feb 2005.

[15] L. Lavagno, A. La Rosa, and C. Passerone. *Hardware/software design space exploration for a reconfigurable processor.* In Proc. of DATE, March 2003.

[16] 3GPP. *Technical specification group radio access network; multiplexing and channel coding.* Technical Specification TS 25.212 v5.1.0, 2002.