A Flexible Code Generation Framework for the Design of Application Specific Programmable Processors

François Charot, Vincent Messé
Irisa/Inria
Campus de Beaulieu
35042 Rennes Cedex, France
{charot,messe}@irisa.fr

Abstract

This paper introduces a flexible code generation framework dedicated to the design of application specific programmable processors. This tool allows the user to build specific compilation flows, using a library of modules, implementing flexible compilation passes such as code generation, resource allocation, scheduling, etc. Retargeting is performed at two levels: minor changes in the target processor architecture are handled by a retargeting of the modules of the defined compilation flow, while major modifications require a structural modification of the flow. To build a compiler for a target processor, the user selects modules from the library, and links them together. While the global compiler structure is user-defined, the retargeting of modules is automatically performed by the framework. Target processors are described using Armor, a programmable processor modeling language especially defined for design space exploration. The proposed tool is then suitable for a large range of instruction set architectures.

1 Introduction

In a hardware-software co-design methodology, designers have to decide which programmable processors should be used to run the software components of the system. Among existing possibilities, in house application specific programmable processors (ASIPs) are valuable components, as they provide a tradeoff between efficiency and flexibility. Retargetable code generation tools play an important role in ASIP design process, since many instruction set architectures have to be evaluated. The target architecture may be changing in order to minimize cost, speed, code size, power consumption and/or to increase performance of the whole system. This is achieved by designing its architecture and instruction set, and then evaluating the code for desired performance. Interesting results on design space exploration can be found in [12, 4, 3].

Three levels of retargetability are commonly considered, differing in the amount of user intervention required during the retargetting process: automatic, user and developer retargetability. Most of the retargetable compilation frameworks [6, 1, 8, 5, 7] operate at a user level of retargetability. A description of the target processor is provided to a compiler-compiler, which determines those optimizations that are applicable to the processor and then automatically constructs an optimizing compiler for it. This approach has led to high-quality compilers for application-specific processor especially in the area of DSP. In such an approach, the target processor has to be close to the specific model supported by the compilation tool, which is only suited for a restricted category of processors and consequentlty suffers from a lack of flexibility.

To allow an efficient architectural exploration process to be performed, code generation tools may deal with a large variety of processors, while achieving a good performance according to code quality. Depending upon the architectural features of the target processor (register and memory structure, data-path organization, etc.), different compilation techniques may be used to produce efficient code satisfying the requirements of the application. In order to be efficiently exploited, these different techniques may require different compilation flows, which is not usually allowed by classical retargetable compilers and is of major importance in the context of an architectural exploration process.

This paper focuses on a code generation framework with the goal of allowing the user to build specific compilation flows, using a library of flexible modules. This framework, operating at a developer level of retargetability, permits the compiler to be tailored to the target processor. The interest of this work is twofold. On the one hand, the framework allows different target processors and variants of these processors to be evaluated and compared, since rapid prototyping of specific compiler flows can be performed. On the other hand, it can be viewed as an experimental framework for experiments in compiler design.

This paper is organized as follows. Section 2 introduces the framework, and the way compilers are built for candidate processors. Section 3 illustrates, by the way of examples, the use of the tool.

2 Flexible code generation framework

2.1 Library structure

The framework is built on a library of modules, as illustrated in figure 1. Each module implements a compilation pass such as code selection, resource allocation or scheduling and is individually flexible. This framework thus allows flexibility at two levels: minor changes in the target processor architecture are handled by a retargeting of the modules of the defined flow, while major changes require a structural modification of the compilation flow, as explained below.

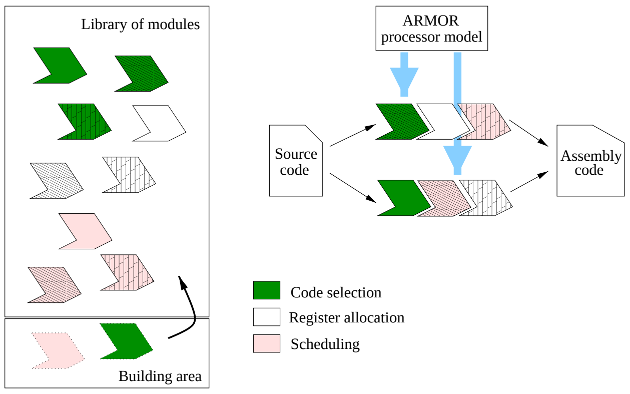


Figure 1: Building code generation flows using the library

To build a compiler for a target processor, the user selects modules from the library, and links them together. Figure 1 illustrates the evaluation and comparison of two compilation flows. The first one achieves sequentially code selection and register allocation followed by compaction. The second one uses three other modules and a different scheduling. Although these two examples illustrate the use of standard compilation flows, more sophisticated modules may exist and implement complex algorithms, mixing, for instance, compaction and resource allocation. Additionally, the user may build iterative flows, where some modules execute several times, until result is satisfied.

While the global compiler structure is defined by the user, the retargeting of modules is automatically performed by the framework. Target processors are described using the ARMOR language [2] which is a programmable processor modeling language especially designed for design space exploration. This language describes instruction set behavior, including semantic, timing information, resource usage, and a detailed instruction-level parallelism specification.

2.2 Flexible modules

In order to achieve an easy retargeting in many different situations, the library should contain many pre-defined modules, implementing various algorithms. Moreover, modules have to share compatible program representations.

Nevertheless, even if many algorithms are available in the default library, the framework must provide a way of easily building new ones. This feature may be useful in implementing architecture specific algorithms when necessary, and in integrating new efficient code generation or optimization techniques.

To this end, the framework has a *building area*, as illustrated in figure 1. A part of this area, used to build optimization modules such as compaction and resource assignment, is illustrated in figure 2 which shows the structure of low-level code treatment tools. Three levels are identified. The *program representation* level consists of a library of C++ classes implementing several program abstractions, such as procedure, basic block, instruction, etc: it concerns the code representation level. The intermediate one, called the *toolkit* level, implements functionalities which are not binded to a particular generation or optimization algorithm: data-flow analysis, timing and resource usage control, generic graph coloring are typical examples. These functions correspond to building blocks, making the writing of new modules easier. The third level, called *user module*, corresponds to allocation, scheduling and optimization algorithms, which are built on the top of the two lower levels.

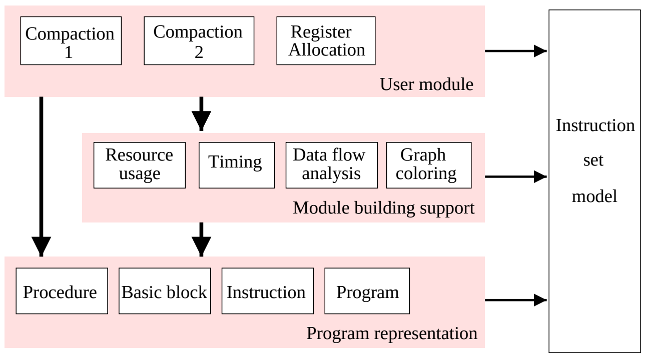


Figure 2: Library structure

2.3 Framework prototyping

In order to experiment with such a structure, we focused on the use of existing tools as components of the framework. Concerning instruction selection, although many accurate algorithms appeared recently, available tools are mainly restricted to standard algorithms based on tree matching. As a consequence, the *Olive* code generator generator [11] was chosen. Concerning code optimizations, we did experiments with the SALTO framework [9] and the SPAM library [10]. SALTO is a tool for assembly code restructuring, and SPAM is dedicated to code optimization for DSPs. Both are built following the structure described in figure 2, with a code representation, some toolkit equivalent components and an algorithmic level. Experiments have underlined qualities and limitations of both tools in the domain of design space exploration.

SALTO benefits from an interesting way of retargeting. The user provides a specification file containing the syntax of instructions, and their resource usage. The model is cycle accurate, and is exploited using a resource and delay checking C++ class, which is very useful in writing scheduling algorithms. However, SALTO is oriented towards general purpose microprocessors, and many architectural specific properties of ASIPs do not fit the model. Such properties have then to be defined manually in specific algorithms. Finally, although SALTO benefits from interesting code representation and toolkit levels, no architecture independent powerful algorithm is distributed with it.

SPAM is oriented towards DSP processors. Unlike SALTO, retargeting is quite difficult, since it does not use a modeling language. Instruction set description is spread out in many different C++ classes. As a consequence, SPAM cannot be rapidly and frequently retargeted. In contrast, SPAM benefits from a powerful toolkit level, including useful tasks such as data-flow analysis or graph coloring. Moreover, the library is distributed with many powerful optimization algorithms.

If none of these tools is suitable for a large design space exploration, experiments have rapidly shown a real complementarity between SALTO and SPAM. As a result, the prototype tool can be seen as composed of SALTO-like components built on the top of a slightly modified SPAM library.

Although the resulting framework is still incomplete, it is close to the scheme described in figure 2.

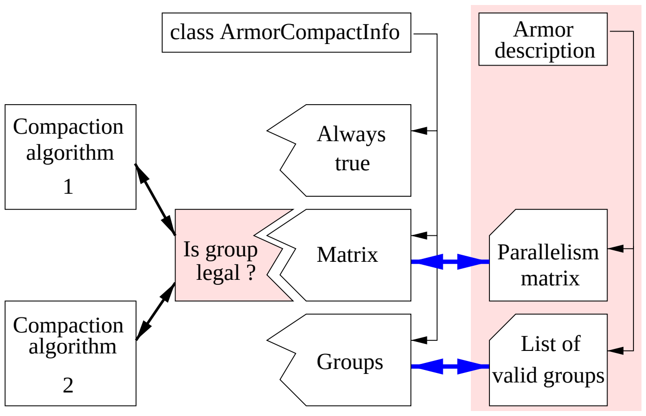


Figure 3: Iterative programmable processor design methodology

3 Use of the framework

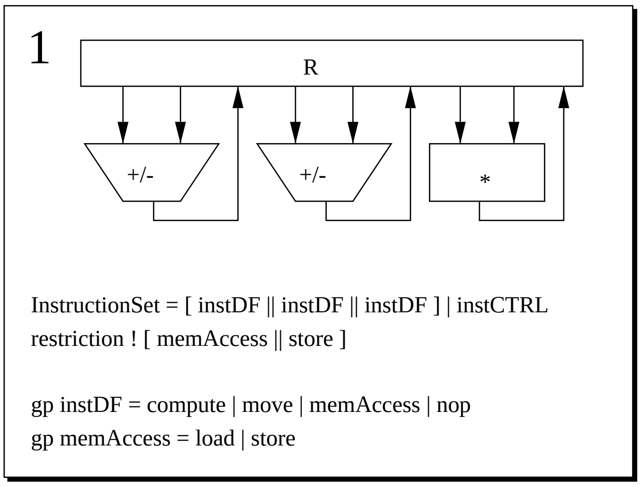
To illustrate the use of the framework, this section focuses on compaction modules. As shown in figure 3, compaction algorithms are written using the *ArmorCompactInfo* class. This class provides a method whose goal is to check if groups of instructions are valid according to the instruction set architecture.

Three different implementations of the checking method are considered. The first one is based on a parallelism matrix. The *(i,j)*th entry in the matrix is *true* if instruction i can be scheduled in parallel with instruction *j*. A group is valid if all combinations of two components are valid and the group size is lower than the global parallelism limit. This representation is accurate in case of architectures having an homogeneous instruction level parallelism. The second implementation is based on an enumeration of valid groups. It may be used in case of processors having an occasional and heterogeneous parallelism. A third implementation (`always true`) considers that groups are always valid. This is for instance useful in measuring the effects of parallelism restrictions on the quality of the compacted code.

3.1 Target architecture examples

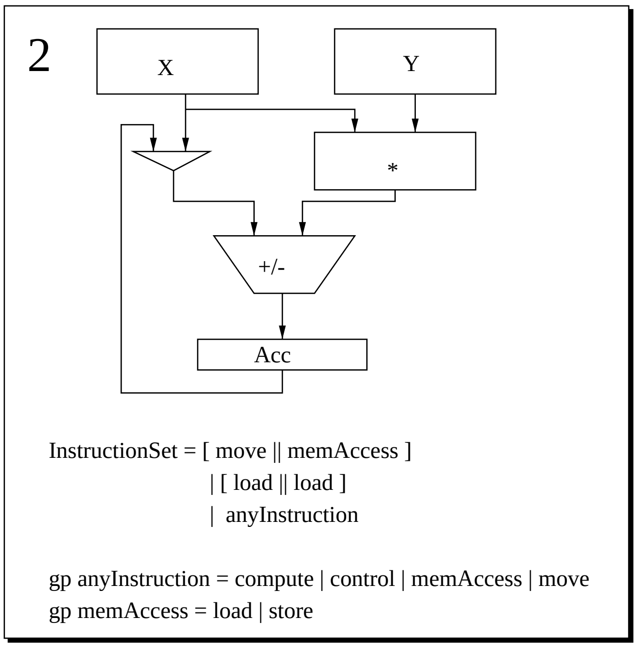
Target processors are described in the ARMOR language [2], especially designed to be used in an application-architecture-compiler codesign framework based on a retargetable compiler technology.

In ARMOR, an instruction set is described using rules, which form a grammar from which each possible derivation represents one legal instruction. A description consists of a top-level rule (`InstructionSet`) which describes all the alternatives in the instruction set and `gp` (group) rules. The combination of such rules models the set of instructions and the available parallelism. The behavior of each instruction is defined using `df` rules (data-flow instructions) which correspond to traversals of a data-path unit and `ctr` rules for control flow instructions. These rules define register transfers using operators (defined with `op` and `class` rules), or lists of operands (`mode` rules). Processor resources are defined using `reg` (register), `regFile` (register file), `mem` (memory), `fu` (functional unit), etc. rules. Resources are stamped with access and timing information. The `restriction` rule models explicit parallelism restrictions. These rules use the parallel (`||`) and alternative (`|`) operators.



InstructionSet = [instDF || instDF || instDF] | instCTRL
restriction ! [memAccess || store]

gp instDF = compute | move | memAccess | nop
gp memAccess = load | store



InstructionSet = [move || memAccess]
| [load || load]
| anyInstruction

gp anyInstruction = compute | control | memAccess | move
gp memAccess = load | store

Figure 4: Two target processor examples

Figure 4 shows two different target processors, and parts of the corresponding ARMOR description (timing and resource usage are not detailed in these examples).The first one is an homogeneous architecture, with a general purpose register file and three computation units. Its instruction set allows either three parallel data flow instructions (named *instDF*), or a control instruction (named *instCTRL*) to be simultaneously executed, but `store` instructions cannot be executed in parallel with any other memory access. The second architecture is an heterogeneous one, with two specific register files (`X` and `Y`), and an accumulator. In addition to conventional computations, the processor has a multiply-accumulate instruction ($Acc = Acc + X \times Y$). In this example, the instruction encoding is supposed to be restrictive. As a consequence only a few parallel combinations are allowed: a `move` instruction with a memory access, or two `load`

	ADD	SUB	MOVE	LOAD	STORE	...
ADD	true	true	true	true	true	...
SUB	true	true	true	true	true	...
MOVE	true	true	true	true	true	...
LOAD	true	true	true	true	false	...
STORE	true	true	true	false	false	...
...	...	...	...	...	...	...

Table 1: Partial compaction matrix for processor #1

Group 1	LOAD - LOAD
Group 2	LOAD - MOVE
Group 3	MOVE - STORE

Table 2: List of valid groups for processor #2

instructions.

3.2 Code generation

Compaction parameters For processor #1, the scheduling algorithm is retargeted using the compaction matrix as illustrated in table 1. Since processor #2 cannot be modeled using this data structure, a list a valid groups of instructions is used instead (table 2). In both cases, explicit parallelism is completed by resource usage checking.

Compacted code Figure 5 illustrates the result of a partial code generation pass. In this example, the same (partial) code generation flow was applied: code selection (using *Olive*), followed by a simple list scheduling algorithm.

Code quality is not the topic here, since optimizations such as loop unrolling or software pipelining should be used. The interesting point is that code generators for two completely different instruction set architectures were automatically generated, using common library modules, built on the top of two different models.

4 Conclusion and future work

The code generation framework introduced in this paper is dedicated to the design of application specific programmable processors. It allows the user to build for a given target, specific compilation flows, using a library of code generation and optimization modules. Different target architectures and variant of these architectures can then be rapidly evaluated and compared.

The prototype is built on the basis of existing tools. Although the library is still incomplete, it achieves a satisfying flexibility. Current work involves the design of powerful optimization modules such as loop optimization and address generation unit exploitation.

However, retargetable code generation is only a part of a design space exploration environment. Target processors evaluation requires compilers to be combined with many other tools, such as system prototyping frameworks, synthesis tools, instruction set simulators, etc. Among these connections, synthesis and system specification are of particular interest.

Processor synthesis In a context of design space exploration, candidate architectures have to be evaluated. Although compiling significant parts of the application is nec-

C source code

```
#define LENGTH 16
int x[LENGTH];
int h[LENGTH];

cv()
{
  int y;
  int i;
  int *px = x;
  int *ph = &h[LENGTH - 1];

  y = 0;

  for (i = 0; i < LENGTH; ++i)
    y += *px++ * *ph--;

}
```

Processor #1 - parallelism matrix

```
     MVC  r10,60  ||  MVC  r3,<.x,0> ||  MVC  r1,0
     ADDu  r4,r10,<.h,0> ||  MVC   r2,0
<L:cv.L3,0>:
     LOAD  r8,(r3) ||  LOAD  r9,(r4) ||  ADDu  r3,r3,4
     MULi  r7,r8,r9 ||  SUBu  r4,r4,4
     ADDi  r1,r1,r7
<L:cv.L1,0>:
     ADD  r2,r2,1
     LESS  r11,r2,16
     BTRUE  r11,<L:cv.L3,0>
```

Processor #2 - valid groups

```
     MVC    Y14,0         ||  MVC    Y15,0
     ST     (FP+cv.y),Y14 ||  MVCu   Y16,60
     ADDu   Acc,Y16,<.h,0>
     ST     (FP+cv.i),Y15 ||  MVCu  Y17,<.x,0>
     ST     (FP+cv.ph),Acc
     ST     (FP+cv.px),Y17
<L:cv.L3,0>:
     LD     AR9,(FP+cv.px) ||  LD    AR11,(FP+cv.ph)
     LD     Acc,(FP+cv.y)  ||  LD    Y10,(AR11)
     LD     X8,(AR9)
     MULACC Acc,Acc,X8,Y10
     ST     (FP+cv.y),Acc
     LD     Y12,(FP+cv.px)
     ADDu   Acc,Y12,4
     ST     (FP+cv.px),Acc
     LD     Y13,(FP+cv.ph)
     SUBi   Acc,Y13,4
     ST     (FP+cv.ph),Acc
<L:cv.L1,0>:
     LD     Y7,(FP+cv.i)
     ADDi   Acc,Y7,1
     ST     (FP+cv.i),Acc
     LD     Acc,(FP+cv.i)
     BRANCH_INF  Acc,16,<L:cv.L3,0>
```

Figure 5: Examples of compacted code for the two processors

essary, this is not sufficient. Resulting code and profiling information have to be used to evaluate the code size and the number of execution cycles. However, in order to evaluate execution time, the machine cycle is essential. A first approach is to build an estimation model, using many high-level information: number of registers, data-path structure, etc. The second approach consists of applying a partial synthesis, until this information is available.

Furthermore, processor synthesis is a necessary step as soon as the final processor is defined. An efficient specification in a hardware description language has then to be produced, starting from a high-level, behavioral instruction set model.

System prototyping The second interesting topic is directly related to hardware/software co-design. In such a context, choosing a target processor implementing software parts of the design depends upon the global system. Therefore, it seems particularly interesting to study the combination of a retargetable code generation framework and rapid system prototyping tools like Ptolemy, COSSAP, SPW, etc. With a tool like Ptolemy, the generation of assembly code for a few DSP processors is supported. To this end, a library of predefined components is available. Therefore, taking advantage of a flexible compilation framework in order to define such libraries for many different target processors could be considered.

References

[1] Jean Claude Bauer, Étienne Closse, Éric Flamand, Michel Poize, Jacques Pulou, and Patrick Penier. SAXO: A Retargetable Optimized Compiler for DSPs. In *Proc. of ICSPAT*, 1997.

[2] François Charot, Gwendal Le Fol, and Vincent Messé. Programmable Processor Modelling for Retargetable Compiler Design and Architecture Exploration. Technical Report 1167, IRISA, January 1998.

[3] Joseph A. Fisher, Paolo Taraboschi, and Giuseppe Desoli. Custom-Fit Processors : Letting Applications Define Architectures. Technical report, HP Laboratories, 1996.

[4] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of DATE conference*, 1999.

[5] Silvina Hanono and Srinivas Devadas. Instruction Selection, Resource Allocation and Scheduling in the AVIV Retargetable Code Generator. In *Proc. of the Design Automation Conference*, 1998.

[6] Dirk Lanneer, Johan Van Praet, Augusli Kifli, Koen Schoofs, Werner Geurts, Filip Thoen, and Gert Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. In *Code Generation for Embedded Processors.* Kluwer Academic Publishers, 1995.

[7] Rainer Leupers and Peter Marwedel. Retargetable Code Generation based on Structural Processor Descriptions. *Design Automation for Embedded Systems*, 3(1):1–36, January 1998.

[8] Clifford Liem, Pierre Paulin, Marco Cornero, and Ahmed Jerraya. Industrial Experience Using Rule-Driven Retargetable Code Generation for Multimedia Applications. In *Proc. International Symposium on System Synthesis*, September 1995.

[9] Erven Rohou, François Bodin, André Seznec, Gwendal Le Fol, François Charot, and Frédéric Raimbault. SALTO: System for Assembly-Language Transformation and Optimization. Technical Report 2980, INRIA, September 1996.

[10] SPAM Research Group, http://www.ee.princeton.edu/spam/. *SPAM Compiler User's Manual*, September 1997.

[11] S. Tjiang. An Olive Twig. Technical report, Synopsys Inc., 1993.

[12] J. Wilberg, A. Kuth, H.-T. Vierhaus, R. Camposano, and W. Rosentiel. A Design Exploration Environment. In *Proc. of the 6th Great Lakes Symposium on VLSI*, pages 77–80, 1996.