Industrial Experience Using Rule-driven Retargetable Code Generation for Multimedia Applications

Clifford Liem[1,2], Pierre Paulin[2], Marco Cornero[2], Ahmed Jerraya[1]

(1) TIMA Laboratory
Inst. Nat. Polytechnique de Grenoble (INPG)
46, ave Félix Viallet, 38031 Grenoble, France
liem@verdon.imag.fr jerraya@verdon.imag.fr

(2) Central R&D
SGS-Thomson Microelectronics (ST)
850, rue Jean Monnet, 38921 Crolles, France
paulinp@stm.com cornerom@stm.com

Abstract

The increasing usage of Application Specific Instruction Set Processors (ASIPs) in audio and video telecommunications has made strong demands on the rapid availability of dedicated compilers. A rule-driven approach to code generation may have benefits over model-based approaches as the user is not confined to the capabilities supported by the model. However, the sole use of transformation rules may or may not be sufficient in optimization abilities depending on the target architecture. This paper outlines experiences with a rule-driven code generation approach for two applications in audio and video processing. The first is a controller for the VideoPhone Codec at SGS-Thomson Microelectronics [1][2]. The second is a VLIW (Very Large Instruction Word) for High Fidelity and MPEG Audio at Thomson Consumer Electronic Components [3]. The experience has shown that a rule-driven approach to compilation is applicable to both the controller and VLIW architectures; however, is limited in optimization abilities for the latter.

1 Introduction

Today's designer for multimedia applications is torn between the rush to meet market windows while balancing the support of ever changing standards and specifications. A key solution to these conflicting requirements is the ASIP, an architecture tuned to the application area while retaining the flexibility for late changes through embedded firmware. As assembly code is too cumbersome to maintain for instruction-set processors, users are making high demands for code generation utilities.

Examples of retargetable code generation systems based on a central model include the CodeSyn compiler of the FlexWare system [4], the Chess compiler [5], and the MIMOLA compiler [6]. Retargeting to new processors can be handled by changes to the instruction-set model at either the structural or behavioural level. Although many recent advances [7][8][9][10] show promise for this approach, the state-of-the-art has not matured to the point where these models can encompass all architecture types.

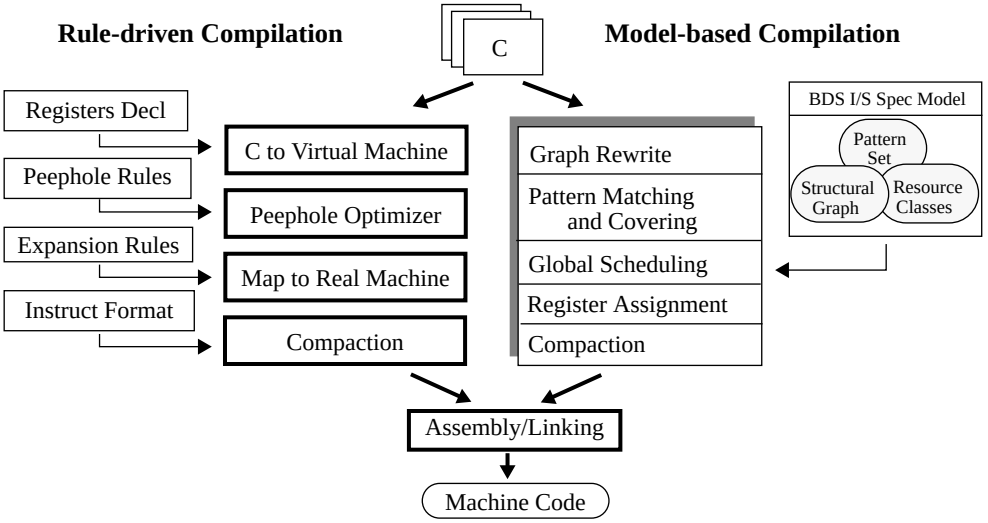

Figure 1 Rule-driven vs Model-based Retargetable Code Generation

On the other hand, a rule-driven approach [11] can be a useful means to arrive rapidly at a dedicated compiler. Rules can be written which do a step-by-step transformation of the source algorithm into the intended assembly and machine code. The quality of the resulting compiler is directly dependent on the skill of the user to write adequate transformation rules. In addition, since there is less need for a heavy intermediate representation and the maintenance of a central model, a rule-driven approach has the potential for very short compilation time.

Independent of the approach used for compilation, rule-driven and model-based approaches may contain common parts. For example, the tasks of assembly and linking can be formulated generally for both approaches. As well, compaction may be shared among the approaches. Figure 1 shows examples of the two approaches to code generation starting from the C source language.

This paper describes experience through the development of rule-driven compilers for two industrial projects. The first is the MicroSeQuencer (MSQ) controller for the Videophone Codec from SGS-Thomson Microelectronics (ST) [1]. The second is an evaluation VLIW (Very Large Instruction Word) chip for High Fidelity and MPEG Audio at Thomson Consumer Electronic Components (TCEC is an enterprise owned jointly beween SGS-Thomson Microelectronics and Thomson Multimedia). This processor is an improvement on a previous chip [3]. While both compilers had a degree of success, the experience has clearly shown the applicability and limitations of a rule-driven approach.

The paper is organized as follows. Section 2 briefly describes the major steps in the rule-driven code generation approach. Section 3 describes the MSQ architecture, a description of the compiler, and presents experimental results. Section 4 has similarily these three sub-sections for the TCEC architecture. A conclusion and summary is provided in Section 5 .

2 Rule-driven Retargetable Code Generation

The steps in the rule-driven approach used in these projects are shown in the left side of Figure 1 and are based directly on concepts in [11]. These steps are explained here for clarity.

After the usual preprocessing step, the C source algorithm is mapped onto a virtual machine for a generic architecture [12]. The virtual machine contains a set of predefined *assembly*-level operations for a non-existing machine; however, it does contain register sets indicated by the user. The user is able to guide the register assignment by indicating which C data-types a register may hold, whether a register is to be used as a source and/or destination for operations, and other various restrictions on the usage of each register. For example, some registers must be set aside as places for intermediate calculations. In other cases, it is sometimes useful to supply non-existent registers for use in the following stages.

Generic operations for the virtual machine are passed to a peephole optimizer. The optimizer transforms sequential occurences of operations into more efficient operations through simple replacements. The user indicates a source and target sequence of code using keywords and wildcards. In addition to optimizations, the peephole optimizer may be used to transform sequences of virtual machine operations not available on the target machine into feasible implementations.

The operations that remain are then expanded into operations for the real machine. Each expansion follows a rule provided by the user. Each rule indicates a source piece of code and a target implementation in the form of micro-operations representing bit fields of the instruction-set. The target implementation may be complex; that is, it can contain conditional statements based on the operands of the source.

Micro-operations are subsequently compacted into instructions. The compaction procedure follows rules of read/write/occupy resources which are indicated

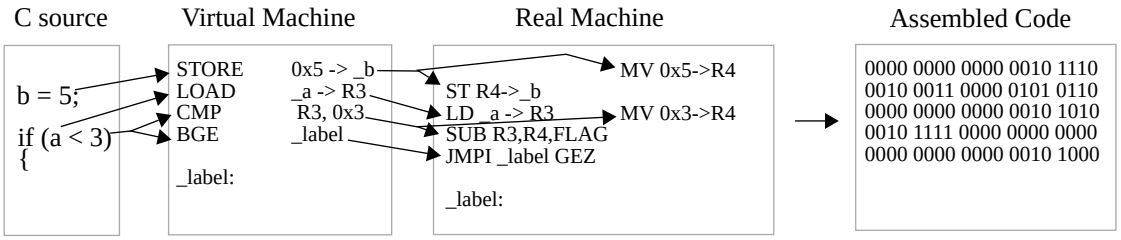

Figure 2 Steps in Rule-driven Compilation

by the user. The compactor attempts to push the maximum number of micro-operations to the earliest possible positions. The straight-forward tasks of assembly and linking immediately follow compaction.

Figure 2 shows an example of the steps in this approach, excluding peephole optimizations. The C source is refined step-by-step into the required machine code. The virtual machine contains generic sequential instructions and is later transformed and compacted into operations on the real machine.

3 MSQ Controller for the ST VideoPhone Codec

3.1 MSQ Architecture Description

The MSQ (MicroSeQuencer) is the top-level control unit for the SGS-Thomson VideoPhone Codec [1][2] and controls the functions of all the other blocks found on the chip. The main functionality of the unit is depicted in Figure 3.

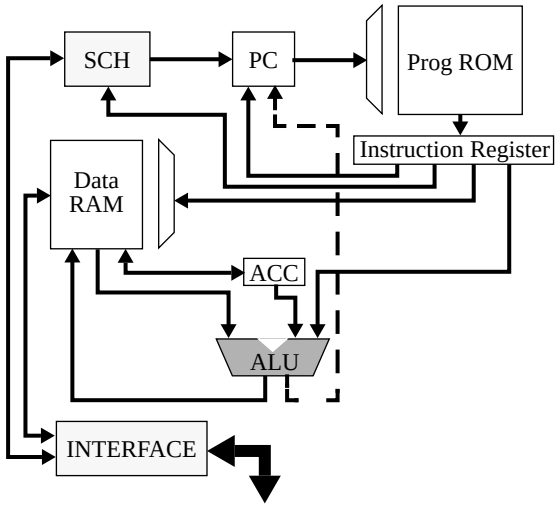

Figure 3 MSQ Block Diagram for the ST VideoPhone Codec

The architecture is a single execution stream controller providing standard ALU operations (ADD, SUB, AND, OR, CMP, SHIFT, etc); as well as standard control operations (BR conditional/unconditional, BR indirect). A Data RAM connects to one input of the ALU and a dedicated register (ACC) connects to the other input. Only one location in the Data RAM may be referenced in each instruction; therefore, values are read and written to the same location. Moves can occur by means of the ACC register.

A unique property of the MSQ is the existence of a register interface which communicates with the rest of the chip. This interface contains a set of registers which can be accessed by both the interior and exterior. A second unique property is a unit known as the scheduler (SCH) which can affect the position of the program counter independent of the natural order of the program. The scheduler can access the interface directly and make decisions depending on values from the exterior. This block works in a polling fashion checking the status of other blocks in the chip and reacting upon this information.

3.2 MSQ C Compiler Description

A rule base for a C compiler was developed for the MSQ in approximately two person weeks. The compiler supports a subset of C; however, it does support the entire functionality of the architecture. The Data RAM is treated as a large register file, thus the support of a traditional memory is unnecessary. The interface and scheduler are treated as special purpose registers which can be accessed directly by the user. Restricted subroutine calling is also supported.

Main issues arisen in the development of the C compiler include the restrictions on source and destination of ALU operations. Since the MSQ allows only one location in the Data RAM to be referenced at one time, ALU operations with more than one source and destination must be transformed to a series of moves by means of the ACC register. In some cases an intermediate location is needed for temporarily storing a value. One location in Data RAM is reserved for this purpose. These manipulations are all handled in the expansion to micro-operations.

Other issues include the manipulation of the C *int* data-type into the single data-type supported by the architecture, which differs from the ANSI standard; the formulation of the interface and scheduler units as special-purpose register structures; and supporting the indirect branching instruction which provides jump table capability for case statements. The latter had to be carefully treated for alignment upon specific bits. As well, we introduced some nice features to the compiler by providing the dynamic expansion of a constant number of shifts into a series of single shifts, as the processor provides only a *shift by one* operation.

3.3 MSQ C Compiler Results

Code was written for approximately 50% of the total expected embedded firmware and was compiled using the MSQ C compiler. The examples contain a cross section of the different types of tasks the MSQ performs. The results are shown in Table 1. The av-

erage code size overhead is roughly 1% smaller when compared with hand code. This indicates that the compiler performs on average as well as an assembly-level programmer.

	Hand Code	C Compiler
codec_gr	189	203 : (7% ov)
codec_mo	318	311 : (-2% ov)
codec_io	592	587 : (-1% ov)
codec_hi	710	676 : (-5% ov)
Average Overhead		**-1 %**

Table 1 Number of Assembly lines: C vs Hand Code for ST MSQ controller

One may argue that the hand assembly could have been improved in some cases. However, we can counter-argue that it is the nature of working on the assembly level that made it difficult to write very compact code. Thus, we conclude that Table 1 is a fair comparison of the two coding levels.

Clearly the MSQ C compiler is a necessary and sufficient compiler for today's MSQ architecture. The rule-driven approach had a small retargeting set-up time and produced high quality results. The main reason for this success may be the narrow range of implementation possibilities for each C construct.

Modifications are planned for the MSQ architecture to include more functionality. The C compiler will be modified to meet these changes. It is also hoped that the compiler be used as a method to examine hardware trade-offs. In this case, new rules for the compiler will have to be added and trialed. While a rule-driven compiler has rapid set-up time, it may not be the best suited for architecture exploration, as rules can be implemented only after a careful study of hardware capabilities.

4 TCEC VLIW Audio Architecture

4.1 TCEC Architecture Description

A top-level diagram for the evaluation Thomson Consumer Electronic Components VLIW architecture for a High Fidelity and MPEG Audio application is depicted in Figure 4. The architecture has a 68 bit wide instruction allowing several tasks to done in parallel. It has two memories each with a dedicated data bus and addressing unit. The ALU has separate register files for the left input and right input. The multiplier (MAC) has a dedicated register at its output. Each ACU unit has address and increment registers including dedicated registers for modulo addressing.

In one cycle, the TCEC architecture can perform the following parallel operations: an access to RAM (X) memory, an access to ROM (Y) memory, an ALU operation, a MAC operation, an increment on addressing unit X, and an increment on addressing unit Y. These can take place within the context of hardware loops, of which 3 nests are possible. Other control features are the standard call/return and branch instructions. On the RAM (X) memory, a stack is available for software use. Each data memory supports three data-types.

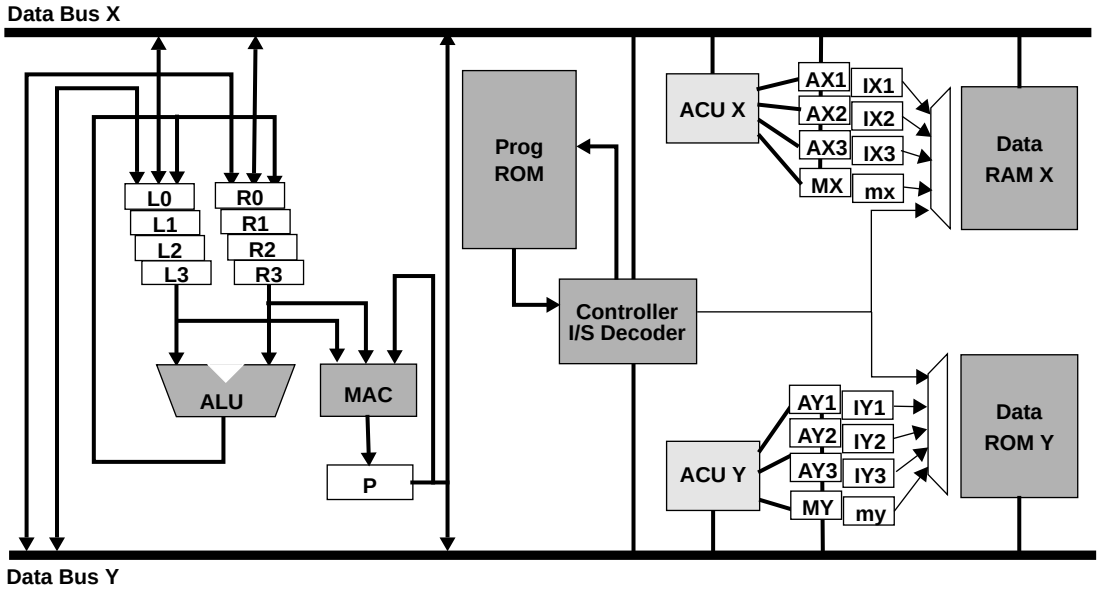

Figure 4 TCEC VLIW Audio Block Diagram

4.2 TCEC C Compiler Description

A rule-driven C compiler prototype was successfully built for the TCEC architecture in approximately 3 person-weeks. It supports a much larger subset of C than the MSQ compiler. This includes the support of data in the two different memories, denoted through small extensions to the C syntax. Three C data-types (char, short int, long int) are supported, with a non-ANSI interpretation of the bit-widths. The permitted arithmetic operations match the capabilities of the ALU. Standard control-flow operations are mapped to conditional and unconditional branches in the standard manner. Subroutines and parameters are supported in the traditional way as a stack is available on the RAM (X) data memory. Hardware do-loops are supported through built-in functions.

Many issues had arisen during the compiler development of this complex architecture. The two data memories were treated in the traditional load/store model with local register files for the ALU. Priorities were placed on the registers in order that the left and right restrictions were not violated. For the wide instruction word, compaction legality rules had to be carefully implemented. The three C data-types were successfully mapped into the data-types supported by the architecture by masking unused bits. Encoding overlaps of fields in the micro-instruction sometimes limited the compaction capabilities. These were overcome with special-purpose peephole optimizations. Sequential occurences of compactable virtual operations were mapped directly into machine operations. Operations which are not possible on the address calculation units because of register restrictions were converted into legal operations through the insertion of moves to legal registers.

4.3 TCEC C Compiler Results

Code was written for an example of the tasks for the TCEC VLIW architecture. The results are shown in Table 2. The example is given in two parts, the overall function and the inner-most loop of the function which is indicative of the time-criticality of the algorithm. The C code was written in three particular styles as to improve compilation results (Note: This was done not to condone the practice of writing low-level code; but rather, to evaluate the level of usability of this compiler). The first ((1) array style) is the designer's original code, closest to the intended algorithm. It uses traditional looping constructs (e.g. for, while) and array structures (e.g. x[i]). The second ((2) pointer style) is a direct transformation of the algorithm using: hardware loops (e.g. for, while => hardware loop), loop-folding techniques, references of the arrays by pointers (e.g. x[1] => x++; *x), and liberal use of the *register* storage class to encourage the efficient use of the local registers. The third ((3) pointer style, assigned registers) is an extension of style (2) with all the registers hand assigned by the programmer. With minor modifications (pre-processor replacements), all styles of the code can also be made to compile on a standard workstation compiler.

It is clear from the inspection of Table 2 that the compiler is unacceptable for a high-level C coding style; the style closest to the designer's original concept. For high performance code, a low-level coding style is required. As this chip is expected to use an expensive on-chip ROM, any code size overhead cannot be afforded. Therefore, coding of style (3) is expected to be predominant in all of the C code.

The main drawbacks with the rule-driven approach for style (1) type of code are:

- Inability to take advantage of dedicated register structures.
- Inability to efficiently use dedicated addressing units for memories.
- The lack of data-flow analysis capabilities for optimizations such as loop-folding.
- The disjoint effects of code generation tasks such as register allocation and compaction.

	Hand Code	**C Compiler (1) array style**	**C Compiler (2) pointer style**	**C Compiler (3) pointer style, assigned registers**
tcec_in	33	160 : (385% ov)	61 : (85 % ov)	33 : (0% ov)
tcec_in (inner loop)	11	47 : (327 % ov)	14 : (27 % ov)	11 : (0% ov)
Average Overhead		**356 %**	**56 %**	**0 %**

Table 2 Number of Assembly lines: C vs Hand Code for TCEC VLIW

5 Conclusion

The two projects cited have shown clearly two extremes in the area of retargetable code generation. The MSQ is a simple dedicated architecture for which a compiler using a rule-driven approach performs as well as an assembly-level programmer. The set-up time was relatively short (2 person weeks) and the compiler is fast and efficient. To date, the users have been quite satisfied with its performance and ease of use. The main drawback for this compiler is the difficulty of being able to do hardware tradeoffs for architecture exploration. To evaluate instruction-set design choices, the retargeting time is longer than the ideal, making interactivity difficult.

For a more complex programmable architecture such as that of TCEC, it was shown that a rule-driven compiler is possible, however with shortcomings. Some of these shortcomings may be improved upon with enhancements to the compiler through the addition of new rules, but it is clear that the approach is limited. A low-level coding style is certainly necessary to achieve both acceptable code size and run-time performance. For an assembly level programmer, the compiler offers some benefits, but clearly this is only a first step towards an acceptable full C compiler. Moreover, architecture exploration tasks for the TCEC architecture using the compiler would become much too cumbersome. This is due to the vast number and complexity of inter-dependent rules. A model-based approach for this type of architecture clearly has benefits.

The experience with these two architectures has shown that the rule-driven compilation approach has benefits in the following areas:

- Simplicity of data-structures: this allows rapid set-up time and compilation time.
- Ease of implementation: the user is able to quickly write and adapt rules for architecture peculiarities.

However, the experience has also shown that rule-driven compilation has drawbacks in the following areas:

- Disjoint steps in compilation flow: this may lead to inefficient code.
- Limited data-flow representation: this restricts the optimization abilities.
- Not automatically retargetable: this makes architecture exploration difficult.

In our experience, by far the greatest advantage to using a rule-driven approach is the retargeting flexibility to control the compilation process for architecture idiosynchrocies that could not be anticipated (the very reason that makes compilation to today's embedded processors difficult). This aspect makes rule-driven compilation an attractive industrial route.

Acknowledgements

The authors would like to thank Michel Harrand of SGS-Thomson for his valuable assistance with issues of the VideoPhone Codec. As well, we would like to thank Laurent Bergher, Jean Marc Gentit, and Xavier Figari of Thomson Consumer Electronics Components for their assistance and discussions of the High Fidelity and MPEG Audio VLIW architecture.

References

[1] SGS-Thomson MicroElec., "STi1100 VideoPhone CODEC Preliminary Data Specification", Aug 1993.

[2] M. Harrand, et. al., "A Single Chip Videophone Video Encoder/Decoder", *Int. Solid-State Circuits Conf.*, Feb 1995, pp. 292-293.

[3] L. Bergher, X. Figari, F. Frederiksen, M. Froidevaux, J.M. Gentit, O. Queinnec, "MPEG Audio Decoder for Consumer Applications", submitted to CICC'95.

[4] P. Paulin, C. Liem, T. May, S. Sutarwala, "FlexWare: A Flexible FirmWare Development Environment", in Code Generation for Embedded Processors, ed. by P. Marwedel, G. Goossens, Kluwer Acad. Pub., 1995.

[5] J. VanPraet, G. Goossens, D. Lanneer, and H. DeMan, "Instruction Set Definition and Instruction Selection for ASIPs", *Int. Symp. on High-Level Synthesis*, May 1994, pp. 11-16.

[6] P. Marwedel, "Tree-Based Mapping of Algorithms to Predefined Structures", *Int. Conf. on Computer Aided Design*, Nov 1993, pp. 586-593.

[7] C. Liem, T. May, P. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation", *European Design & Test Conf.*, Feb 1994, pp. 31-37.

[8] C. Liem, T. May, P. Paulin, "Register Assignment through Resource Classification for ASIP Microcode Generation'', *Int. Conference on CAD*, Nov 1994.

[9] D. Lanneer, M. Cornero, G. Goossens, H. DeMan, "Data Routing: a Paradigm for Efficient Data-path Synthesis and Code Generation", *Int. Symp. on High-Level Synthesis*, May 1994, pp. 17-22.

[10] R. Leupers, W. Schenk, P. Marwedel, "Retargetable Assembly Code Generation by Bootstrapping", *Int. Symp. on High-Level Synthesis*, May 1994, pp. 88-93.

[11] R.P. Gurd, "Experience Developing Microcode Using a High-Level Language", *Proc. of the 16th Annual Microprogramming Workshop*, Oct 1983, pp. 179-184.

[12] S. Antoniazzi et. al., "A Methodology for Control-Dominated Systems Codesign", *Int. Workshop on Hardware/Software Codesign*, Sept 1994, pp. 2-9.

[13] P. Paulin, C. Liem, T. May, S. Sutarwala, "DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective", *Journal of VLSI Signal Processing*, 9., K.A.P., 1995, pp. 23-47.