MOUSE: A Shortcut From Matlab Source to SIMD DSP Assembly Code

Gordon Cichon and Gerhard Fettweis

Mobile Communcations Chair, TU-Dresden
D-01062 Dresden, Germany
cichon@ifn.et.tu-dresden.de

Abstract. This article presents a novel design flow called MOUSE for the effective development of digital signal processing systems in terms of development time, performance and power consumption. It uses a model in high-level language like Matlab[1] as a starting point. Utilizing techniques originating from supercomputing and dynamical compilation, these models can be translated to assembly code for specialized DSP processors of the CATS family. An implementation of terrestial digital video broadcast (DVB-T) serves as an example.

1 Introduction

The traditional design flow for the development of signal processing systems requires the implementation of two models of the system: First, conceptual development and prototyping is done in high-level languages like Matlab. The actual implementation then takes place in a second model, which is developed in a modeling language close to the target hardware platform.

According to [8], Matlab is the most popular language for the development of the first kind of model. It has a community of over 500,000 users.

The development of the second model is usually the major effort in the development process. It is written either in VHDL for FPGA and ASIC targets, or in low-level C and assembly language for DSP targets. Fig. 1 shows such a design flow on the left hand side.

Experience with the development of complex signal processing systems at the Vodafone Chair of Mobile Communication Systems suggests that a great amount of productivity enhancement can be achieved by cutting down the development effort of a target specific system model. For this reason, we chose to avoid reimplementing the second model from scratch, but instead to reuse the high-level model for this purpose (see Fig. 1, right hand side).

This paper presents a high-level language frontend of the MOUSE compiler system. Additionally, we provide an implementation of a transmitter and receiver for terrestial digital video broadcast (DVB-T), as a benchmark application.

[1] Matlab is a registered trademark of MathWorks, Inc.

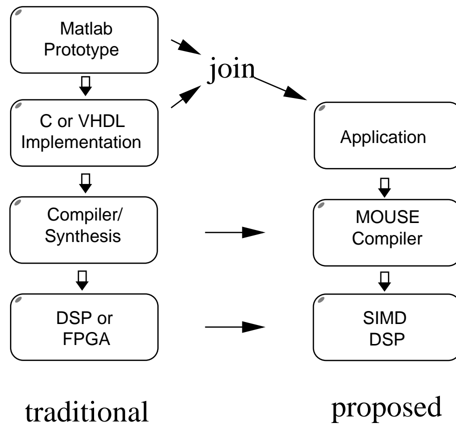

Fig. 1. Digital Signal Processing, Design Flow

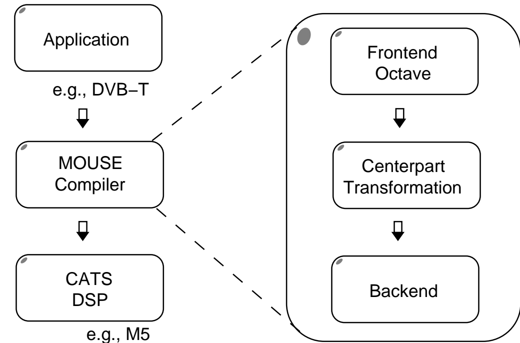

Fig. 2. MOUSE development environment

2 Related Work

Matlab compiler [11] translates code from Matlab into generic C code. The type switching and memory management of this approach is not optimized for an environment with tight constraints on memory consumption and performance.

[3] offers a dynamic compilation technique that utilizes runtime type inference. This approach is not well-suited to embedded systems with tight memory consumption constraints, either.

[8] presents a system to compile Matlab to FPGA platforms. This kind of translation implies a data-flow driven architecture.

[9] presents a system for translating nested loop programs written in the Matlab programming language into a process network description. This kind of process network description is suitable for implementations using generic functional units constituting a systolic array.

It has been shown at the M3 processor design [15] that for many typical signal-processing applications, like wireless transmission systems, DSL, and others, an implementation using a SIMD approach can improve resource utilization and is able to reduce the demand for buffer memory between different processing elements. This is particularly the case with systems that involve data interleaving [16].

3 The MOUSE Design Flow

Fig. 2 shows a novel design flow for the development of a digital signal processing system based on the experience at the Vodafone chair. The Matrix Optimized Universal Signal-processing Environment (MOUSE) leverages the inherent parallelism inside vector- or matrix-based high-level development languages, such as Matlab [11], for the compilation of programs for SIMD processors.

It utilizes techniques originating from supercomputing, especially vectorizing compilation [2], [19], and the efficient translation of dynamically typed programming languages, such as Java [4].

Using this approach, high-level language models can serve directly as a source code to implementations in a SIMD architecture, like CATS DSP family [15], without the need for rewriting the algorithm in a language of lower abstraction level, such as C or VHDL.

The MOUSE compiler consists of three pieces: First, a frontend that reads the program source code and transforms it into a suitable intermediate representation. Second, the centerpart that performs dynamic type analysis, floating point to fixed point mapping, vectorization, and unique program transformations. And third, a traditional compiler backend [1, 12] that maps an intermediate representation at the level of C code to machine language.

Together, these three pieces form a tool-chain that enables rapid system development for signal processing systems. In contrast to other approaches, the proposed design flow allows the compilation of assembly code for a high-performance, low-power DSP-implementation from existing Matlab code.

A carefully written code can be translated directly without further modification. However, in larger projects, it often becomes necessary to gradually adapt the code to facilitate recognition of data types and to exploit parallelism. In contrast to dual-implementation approaches, the transition can be done within a running environment, and without breaking existing code. It can be done in a way that is integrated in the environment.

In addition to that, modifications are small and merely affect the arrangement in which statements and loops are written. However, the abstraction level and the expressiveness of the code still remain high. The necessary transformations are the ones described for Fortran in high-performance super-computing environments [2].

4 Octave Frontend

The frontend part of the MOUSE compiler is based on the code of the Octave interpreter. Octave accepts a substantial subset of the Matlab programming language. It is freely available under the general public license (GPL), and the Octave Frontend can be downloaded from the MOUSE homepage [5].

The MOUSE adaptation of Octave can either interpret high-level language programs, or it can act as a compiler frontend to translate programs into advanced intermediate representation (AIR), suitable for the following stages of the compiler. This intermediate representation can also be translated to the Aterm format, which is used at the Stratego project [17] at the University of Utrecht.

To achieve this functionality, a new software module has been introduced into the octave interpreter. This software module converts the internal program representation of Octave into that of AIR and writes it to a file. Conversion to Aterm is achieved by running the embedded rna2aterm converter.

RNA is a simple library that facilitates the handling of persistent tree and graph data structures. Figure 3 shows an overview of the fundamental data types available in RNA.

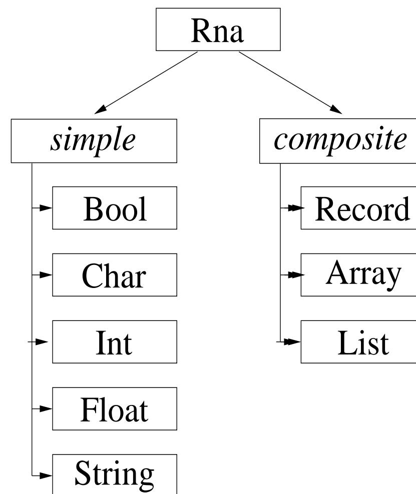

Fig. 3. RNA Architecture

Simple data types include booleans, strings, integers, and floating point values. Composite data types are records, arrays and lists. The composite data types can reference each other deliberately, such that graph data structures can be constructed.

The following features of Matlab are supported by the octave frontend:

4.1 Constants and Variables

- **variable**
  Variables can have an arbitrary name according to standard identifier naming rules. Types of variables are not explicitly specified in Matlab. First releases of the centerpart of the compiler will require a separate description of the types of the used variables. In future versions, automatic type inference can be accomplished either by profile based approaches [18], or by abstract interpretation [6].
- **constant**
  Constants can be integers (e.g.: 1), floating point values (e.g.: 3.14), or a vector of items enclosed in square brackets. (e.g.: [1; 2; 3]). The sophisticated parsing rules for matrices in Matlab apply.

4.2 Types

- **bool**
  Matlab does not allow explicit designation of values to the type boolean. However, the compiler can recognize certain expressions and variables as boolean values through the way of their creation and use. Vectorized boolean values play an important role in the parallelization of if-statements.

- **integer**
  Integer is a basic type of machine precision. In the CATS DSP design, this precision will presumably be 16 bit. Other instances of the DSP platform might work with 8 or 32 bit for this quantity. Currently, there is no support for character data types.
- **float**
  Since floating point computation is not supported by typical DSP platforms, a floating point values can be transformed globally into an integer representation using a suited transformation technique (e.g., [18] or [6]) by the compiler centerpart.
  Currently, all floating point values are represented as q.15 fixed point values by the compiler centerpart. In the process of accumulating results from vector products, 40-bit intermediate results may be generated.
- **vector**
  A vector will be distributed across the slices of the SIMD DSP processor, and the compiler will try to execute operations on vectors in a parallel way across different slices using SIMD parallelization techniques [2]. Vectors with a constant length, and with a length of a multiple of the number of slices will experience a performance gain.
- **matrix**
  A matrix is represented as a two-dimensional vector. Arranging a matrix in memory gives the compiler a degree of freedom: It can either be stored in row order or in column order. Currently, matrices are sorted in the same order as in Fortran programs.

4.3 Expressions

Matlab supports several different kinds of expressions:

- **primary expressions**
  A primary expression is a variable name or a constant (e.g.: a, 5).
- **indexed expressions**
  An indexed expression consists of an expression followed by a comma-separated list of arguments enclosed in parentheses. The individual arguments are expressions, in turn. In Matlab, an indexed expression can represent either a function call or an element access of a vector (e.g.: fun(1,5), a(5)).
- **arithmetic expressions**
  Arithmetic expressions are combinations of expressions linked with arithmetic operators. Currently, the following operators are supported: + - * / & | && || ! (e.g., 1+1, a+5, a(i)+b(i)).
- **range expressions**
  A range expression creates a vector filled with a sequence of values. These values start with a specific initial value, they have a constant increment, and end with a final value (e.g.: 1:n means [1; 2; ...; n], 3:3:18 means [3; 6; 9; 12; 15; 18]).
  Ranges play a special role as range expressions for loops. In this case, the corresponding vector is not created explicitly. Instead, the loop induction variable is initialized with the corresponding range value for each loop body instance.

4.4 Statements

- **assignment statement**
  An assignment statement consists of two expressions separated by an assignment operator. Currently, the following assignment operators are supported: = += -= *= /=. (e.g., a=1; a+=1 a(i)=b(i)+c)
- **if statement**
  If statements are supported without restrictions.
- **while loop**
  While loops are translated directly into loops on the assembly level.
- **break**
- **continue**
- **return**
  The above three control flow statements facilitate expressive structured programming in the absence of goto statements.
- **for loop**
  For loops are subject to vectorization by the centerpart. The compiler tries to find loops in which there are no data dependencies between the individual instances of the loop body over the induction variable. If this case is detected, the compiler can potentially parallelize the loop using SIMD parallelization.
- **statement list**
  Statements can be chained together, separated by full stops or semi-colons.

4.5 Function Definition

For the development of structured programs, the definition of functions is supported by the compiler. At this time, variable length argument lists are not supported.

5 Benchmark Application: DVB-T

To evaluate the effectiveness of the proposed design flow, the implementation efforts of a system are compared under two different development environments: An implementation in Matlab, and an implementation in assembly language.

As benchmark application, DVB-T was chosen. It is a complex signal processing application, including Reed/Solomon, Viterbi, FFT, and interleaver algorithms. The structure of the system is shown in Figure 4.

The DVB-T system is implemented according to the ETSI standard [7]. We considered a standard system with 2048 carriers, 16-QAM, and 2/3 puncturing. This yields a typical data rate of 14 Mbps.

The system is composed by the following modules:

- **Scrambler**
  This is a simple XOR-operation with a pseudo-random bit-sequence generated by a feedback shift register. The operation requires 2 MIPS with a stored PRBS sequence.

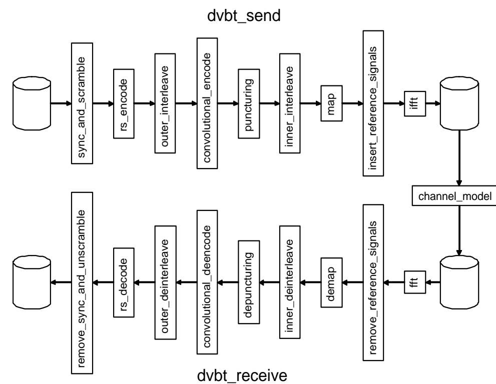

Fig. 4. DVB-T Overview

- **Reed/Solomon**
  This is a shortened (188,204,16) decoder. It is implemented using Berlekamp-Massey, Chien-Search, and Forney. The data rate requires 11 MIPS to perform the decoding. However, this kind of algorithm is quite control flow intensive for a DSP application, and thus serves as a good indicator for the behavior of the CATS DSPs in terms of performance, power-consumption and code density.
  Our group has also developed a Reed/Solomon decoder in assembly language, written by a student worker [13]
- **Outer Interleaver**
  A convolutional interleaver with parameters I=12, M=17.
- **Viterbi Decoder**
  This is a 64-state, soft-decision Viterbi decoder. In Matlab, we use the implementation of Kammeyer and Kuehn [10]. This is the computationally most intensive part of the system, requiring 990 MIPS for Trellis computation and 140 MIPS for traceback.
  Our group has also developed an assembly language implementation of a Viterbi-decoder which uses either 4 or 16 bit of fixed point precision [14].
- **Inner Interleaver**
  A block interleaver on OFDM symbols.
- **Mapping**
  Currently, the DVB-T system supports 16-QAM mapping.
- **OFDM-demodulation**
  This module consists of a FFT. It is computationally expensive, requiring 720 MIPS.
  The Matlab implementation uses a call to the `fft` runtime library function.

As shown above, the system requires a high data processing rate, and, for mobile applications, a low power consumption. A performance analysis of the data rate required for DVB-T shows that Viterbi needs 1130 MIPS and Reed-Solomon requires 11 MIPS for the signal processing part. Also, the whole system is supposed to run battery-powered, i.e. with a power consumption around 1 Watt.

The implemented system runs in the Matlab environment, as well as on the MOUSE Octave Frontend. It is available for download under a BSD license from the MOUSE homepage [5].

Although automatic parallelization does not work reliably in all circumstances because of Turing computability reasons, it can be concluded that with minor program modifications for fine tuning, compiled code can be made comparably fast as handwritten assembly code.

6 Results

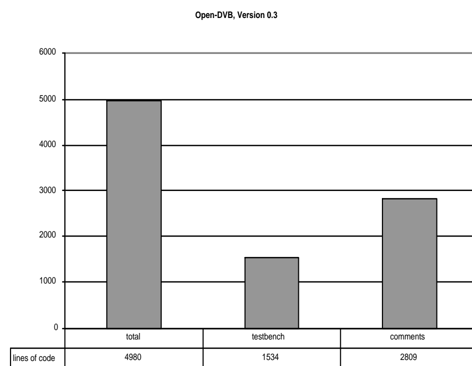

Fig. 5. DVB-T: total system

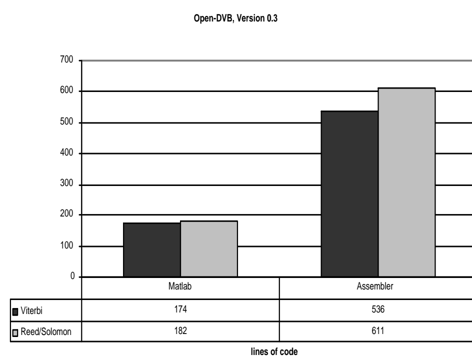

Fig. 6. DVB-T: lines of code

As depicted in Figure 5, the Matlab implementation of Open-DVB, Version 0.3, has about 5000 lines of code in total. A significant number of these lines are comments. In addition to that, code of testbenches is more than one third of the total code. In the following comparison, comments and testbenches will not be counted. The entire DVB-T code was implemented within about three man-month.

Two important modules, Reed-Solomon [13] and Viterbi [14], were implemented in assembly code by students. Each of these two modules took about three man-month to implement. In the following, the Matlab and assembly language implementations of these modules are compared.

Figure 6 shows the lines of code needed to implement the two algorithms in Matlab and assembly code. It can also be observed that the code lines in Matlab rarely exceed 80 columns. Due to the VLIW instruction set of the target DSP, code lines are much wider in assembly code: up to 536 columns for Viterbi and up to 611 columns for Reed/Solomon. It can be concluded that implementation in Matlab requires significantly less effort.

In addition to that, the implementation of a complex system like DVB-T on the MOUSE compiler framework serves as an indicator for its stability and correctness.

7 Further Work

As a next step, the implementation of the centerpart will show the viability of this approach. Modules currently under work are type inference, memory layout of data, and parallelization by vectorization. A deeper coverage of the centerpart and backend of the MOUSE compiler will be presented in a separate paper.

Acknowledgments

The authors would like to thank K. Kammeyer and V. Kuehn for the contribution of their Viterbi decoder and D. Schoenfeld for her help with the math of Reed/Solomon. Thanks also to the CATS team and to Caroline Cichon.

This work has been sponsored in part by the German Science Foundation (Deutsche Forschungsgemeinschaft, DFG) within SFB358-A6.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, Redding, MA, 1985.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, Burlington, MA, 2001.
3. G. Almasi and D. Padua. Majic: Compiling matlab for speed and responsiveness. In *ACM SIGPLAN PLDI*, Berlin, 2002.
4. C. Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Computer Science Department, Stanford University, March 1992.
5. G. Cichon. MOUSE: Matrix Optimized Universal Signal-processing Environemnt. `http://www.radionetworkprocessor.com/`.
6. A. Cortesi, editor. *Verification, Model Checking, and Abstract Interpretation, Third International Workshop, VMCAI 2002, Venice, Italy, January 21-22, 2002, Revised Papers*, volume 2294 of *Lecture Notes in Computer Science*. Springer, 2002.
7. ETSI. Digital video broadcasting (DVB); framing structure, channel coding and modulation for digital terrestrial television. EN 300 744 V1.4.1, 2001.
8. M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee. A system for synthesizing optimized FPGA hardware from matlab™. In *Proc. of the 2001 IEEE/ACM international conference on Computer-aided design (ICCAD'01)*, pages 314–319, San Jose, CA, Nov. 2001.
9. T. Harriss, R. Walke, B. Kienhuis, and E. Deprettere. Compilation from matlab to process networks realized in FPGA. *Design Automation of Embedded Systems*, 7(4), 2002.
10. K. Kammeyer and V. Kuehn. *Matlab in der Nachrichtentechnik*. J. Schlembach-Verlag, Weil der Stadt, Deutschland, 1. auflage edition, 2001.
11. Matlab™. `http://www.mathorks.com/`.
12. S. Muchnik. *Advanced compiler design and Implementation*. Morgan Kaufmann Publishers, 1997.
13. Rene Beckert. Implementierung eines Reed-Solomon-Decoders fuer DVB-T auf einem hochparallelen Signalprozessor. `ftp://ftp.radionetworkprocessor.com/pub/vodafone-chair/Studienarbeit-Be%ckert.pdf`, 2003.

14. Rene Habendorf. Implementierung eines Viterbi-Decoders fuer DVB-T auf einem hochparallelen Signalprozessor. ftp://ftp.radionetworkprocessor.com/pub/vodafone-chair/Studienarbeit-Ha%bendorf.pdf, 2002.
15. T. Richter, W. Drescher, F. Engel, S. Kobayashi, V. Nikolajevic, M. Weiss, and G. Fettweis. A platform-based highly parallel digital signal processor. In *Proc. CICC*, pages 305–308, San Diego, USA, 2001.
16. T. Richter and G. Fettweis. Parallel interleaving on parallel DSP architectures. In *Proc. of IEEE Workshop on Signal Processing Systems (SiPS'02)*, pages 195–200, San Diego, USA, oct 2002.
17. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
18. M. Willems, V. Buersgens, H. Keding, and H. Meyr. FRIDGE: Fliesskomma-Programmierung von Festkomma-DSPs. In *DSP Deutschland*, Muenchen, 1997.
19. Zima and Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, Redding, MA, 1990.