Retargetable Compilers for Embedded DSPs

Rainer Leupers, Peter Marwedel
University of Dortmund
Department of Computer Science 12
44221 Dortmund, Germany
email: leupers|marwedel@ls12.cs.uni-dortmund.de

Abstract

Programmable devices are a key technology for the design of embedded systems, such as in the consumer electronics market. Processor cores are used as building blocks for more and more embedded system designs, since they provide a unique combination of features: flexibility and reusability. Processor-based design implies that compilers capable of generating efficient machine code are necessary. However, highly efficient compilers for embedded processors are hardly available. In particular, this holds for digital signal processors (DSPs). This contribution is intended to outline different aspects of DSP compiler technology. First, we cover demands on compilers for embedded DSPs, which are partially in sharp contrast to traditional compiler construction. Secondly, we present recent advances in DSP code optimization techniques, which explore a comparatively large search space in order to achieve high code quality. Finally, we discuss the different approaches to retargetability of compilers, that is, techniques for automatic generation of compilers from processor models.[1]

1 Introduction

The consumer electronics market can be characterized by rapidly growing complexities of applications and a rather short market window. Therefore, more and more complex designs have to be completed in shrinking time frames. Meeting short time-to-market requirements is only possible, if system design technologies permit *flexibility*, so as to accommodate late specification changes, and *reuse* of predesigned components. Both conditions are met, if *embedded processors* are used as building blocks in system design. They provide flexibility through programmability and enable the reuse of software modules, such as C function libraries. Therefore, a trend towards processor-based design of embedded systems is currently observed. As a consequence, the major part of design effort is frequently spent in development of *embedded software* rather than in design of custom hardware [1]. Fig. 1 shows a design scenario for embedded software. An application source program is specified, typically in C language, and is profiled, so as to identify (and possibly accelerate) "hot spots" in the program. Next, machine code for the embedded processor is generated using a high-level language compiler. The generated code is simulated and debugged, and the source program is adapted in case that errors have been identified. This process is iterated until a feasible implementation has been achieved.

In most cases, code is generated for a *fixed* processor only, for instance a certain standard DSP. However, such an "off-the-shelf" processor frequently is not the most efficient solution in terms of computation speed, chip area, and/or power consumption. Instead, an *application-specific processor* may much better serve the needs of a particular application. Therefore, from a more general viewpoint, the design (or at least: the selection) of a suitable *target processor*

[1]Publication: 7th European Multimedia, Microprocessor Systems and Electronic Commerce Conference (EMMSEC), Florence/Italy, Nov 1997.

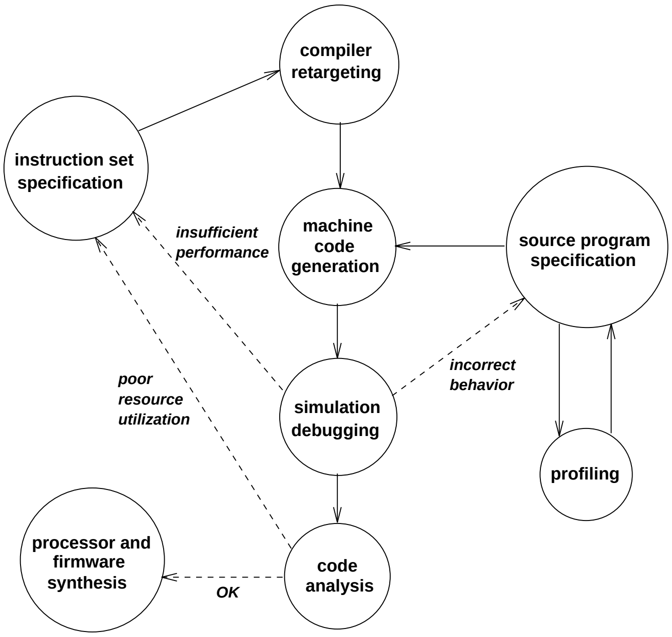

Figure 1: Embedded software design flow

is also part of the embedded software design process. In such a scenario, an initial target processor model, e.g. an instruction set, is specified and the compiler is *retargeted* to that instruction set. Then, the program is compiled as described above. Performance information gained during the simulation/debugging phase or during a subsequent *code analysis* phase can suggest beneficial adaptations of the target processor model. This retarget-and-compile cycle is iterated until a satisfactory solution in terms of both software and target processor hardware has been obtained. Finally, the target processor and the "firmware" machine program can be synthesized. Retargetable (and optimizing) compilers for embedded processors have received much interest recently, because they permit to study the interplay of target hardware architectures and program execution speed [2].

In this paper, we focus on retargetable code generation for embedded DSPs. The organization of the paper is as follows. First, in section 2, we summarize the demands on compilers for embedded DSPs. In section 3, we consider code generation and, in particular, optimization techniques. Approaches to retargetable compilation are treated in section 4, and an example for a retargetable DSP compiler system is presented in section 5. Finally, conclusions are given in section 6.

2 Demands on compilers for embedded DSPs

DSPs are programmable processors with instruction sets that are tuned to fast execution of arithmetic-intensive programs. The general demands on DSP compilers arise from the characteristics of DSP applications as well as from the characteristics of system environments which DSPs are embedded in.

Extremely efficient code: DSP programs frequently operate under real-time constraints. Meeting real-time constraints usually requires exploitation of the full amount of capabilities of a DSP, so that very efficient code must be generated. Furthermore, as program code is typically stored in on-chip memories, also the size of machine programs is critical. Thus, any loss of efficiency caused by a compiler increases the silicon area. In addition, violations of real-time constraints in compiler-generated code must be compensated by higher clock rates. In turn, this increases the power consumption. In general compiler construction, generation of extremely efficient code has traditionally been treated with lower priority than high compilation speed. Therefore, many C compilers available for standard DSPs show an unacceptable code quality [3], and most of the DSP software is still written at the assembly level.

Support for DSP algorithms and architectures: DSP algorithms show characteristics usually absent in general-purpose computing. These include bit-true specifications, rounding behavior of arithmetic operators, and cyclic buffers. DSP processor architectures include special hardware to accommodate such characteristics. However, common programming languages like C do not directly support DSP-specific programming constructs, but certain work-arounds have to be used. In turn, this often disables the compiler to efficiently map such constructs to the corresponding hardware. One way to avoid this problem is to use a DSP-specific programming language, such as DFL [4]. Nevertheless, DSP compilers still face the problem of mapping program constructs to highly irregular architectures, as they are typically found in DSPs.

Retargetability: A large number of processor *cores*, including microcontrollers, RISCs, and DSPs, are currently available from vendors [5]. Cores are macro cells, which can be instantiated like library components, and which are shipped in form of "soft" register-transfer (RT) level VHDL models or in form of "hard" VLSI layout cells. For such cores, compilers might even be not available at all. In order to avoid the necessity of developing a new compiler for each new processor core, retargetable compilers are a promising solution. Such compilers can be adapted, so as to generate machine code for each member of a defined class of processors. Retargetable compilers are also important tools for the extended software development cycle outlined in section 1.

3 Code generation and optimization

In classical compiler construction, code generation and optimization are often treated as separate compilation phases. First, a program is translated into a valid (possibly poor-quality) machine program, which is later optimized by applying transformation rules. For DSPs, such an approach is not viable, because irregularities in the processor architecture cause a high interdependence between all different code generation phases. In order to avoid code quality overhead due to separate execution of code generation phases, *phase coupling* is necessary. For instance, instruction selection should be done while taking into account its impact on scheduling. Simultaneously, allocation of registers for program values must be considered. In order to minimize combinational delays and to permit pipelined execution of instructions, DSPs typically show *special-purpose registers* connected to specific functional units in the data path. Obviously, an unfavorable instruction selection may cause a large number of data moves between such registers.

An important source of potential optimization is *parallelism at the instruction level.* Most DSPs are capable of executing a set of RT operations in parallel in each machine cycle. This includes parallel operations such as "multiply-accumulate" and also parallel computation of memory addresses. In order to cope with these special requirements, a number of new code generation/optimization techniques have been developed. Important projects in this area include:

SPAM: In the SPAM project, graph-based approaches to tight coupling of instruction selection, register allocation, and scheduling have been developed for different standard DSPs, such as TI TMS320C2x and Motorola DSP56k [6, 7, 8]. In certain cases, optimality has been proven, and commercial compilers have been outperformed in terms of code quality. Further contributions from the SPAM project include optimization techniques for parallel memory address computation [9, 10], aiming at high utilization of auto-increment capabilities of address registers.

FlexWare: In this project, a rule-based C compilation technique for application-specific DSPs has been implemented [11]. There, compilation of source code constructs into target machine instructions is guided by translation templates provided by the user. The more templates are available, the higher is the optimization potential for the compiler. Additionally, the compilation process can be steered by low-level programming constructions, such as manual binding of values to physical registers. It has been shown, that this approach can yield code quality comparable to that of manually written assembly programs. However, detection of good translation rules may be difficult, and the effort of low-level C programming may become close to assembly programming. Another contribution of this project is a C to C translator, which replaces array references in programs by pointer arithmetic operations [12]. Since pointers and pointer arithmetic can be directly mapped to address registers and arithmetic operations on these, a higher exploitation of auto-increment capabilities of address registers can be achieved than in the original array-style C program.

Mutation Scheduling: The Mutation Scheduling (MS) approach developed at UC Irvine [13] aims at a tight coupling of all different code generation phases. The main idea is to maintain different translation schemes ("mutations") for each source program value during compilation. For instance, the value "`x * 2`" may be equivalently written as "`x + x`" or "`x << 1`". If a value has multiple occurrences in a program, it may be recomputed each time or may be kept in a register for later reuse. The most appropriate mutations are selected based on data path resource availability in each program control step. Due to its "ideal" phase coupling, MS can yield very high quality code, however, at the expense of long compilation times.

Another research project – the RECORD compiler – will be described in more detail in section 5.

4 Retargetable compilation

Retargetable compilers are useful if target processors for embedded software change frequently, but the main architectural characteristics remain constant. In that case, these target processors belong to a common *class*, for instance the class of fixed-point DSPs. We call a compiler *retargetable*, if it can be adapted, so as to generate machine code for any processor within a class, in such a way that the largest part of the compiler source code is retained. With retargetable compilers, target processors can be switched by simply adapting the compiler instead of completely developing a new compiler from scratch. This is particularly important in the area of consumer electronics, where a large variety of different application-specific processors are in use. Retargetable compilers are also important for design space exploration in case that the target processor is not completely fixed in advance (see fig. 1).

A common characteristic of retargetable compilers is, that they read another input besides the source program to be translated, namely a *target processor description* given in some modelling formalism. A well-known approach to retargetable compilation is the GNU C compiler [14]. GNU C has been successfully ported to a number of different CISC and RISC machines, but it does not support frequent changes of the target processor due to a complicated target processor modelling formalism. Therefore, researchers have looked at more convenient processor modelling formalisms. In the projects mentioned in the previous section, retargetability is achieved as follows:

SPAM: The target processor model mainly consists of a set of tree-shaped *instruction patterns*, which are textually specified in form of a tree grammar. Code generation is considered as the process of parsing source program assignments with respect to the given tree grammar. Retargeting to a new processor is realized by replacing instruction patterns in the grammar. However, the grammar can become relatively large, and potential parallelism between instructions cannot be captured.

FlexWare: The target processor model is implicit in the set of translation rules that are provided to the compiler, i.e., new instructions are modelled by specifying new translation rules. The main problem in this approach, however, is that mainly the user – and not the compiler – is responsible for efficient exploitation of available instructions, since the compiler only applies the rules by a macro-expansion mechanism.

Mutation Scheduling: Similar to FlexWare, the available target machine instructions are implicitly encoded in the set of possible mutations for each program value. However, the compiler is burdened with appropriately selecting instructions. Therefore, a higher degree of automation and a higher optimization potential are achieved.

All of the above approaches use rather specific processor modelling formalisms. However, as the design of processor-based systems usually takes place within a *hardware design environment*, standardized hardware description languages (HDLs) such as VHDL are a more promising solution, because they permit to use the same processor models for synthesis, simulation, and code generation. The RECORD compiler presented in the next section therefore uses processor models described in an HDL.

5 An example: the RECORD compiler system

RECORD is a retargetable compiler for fixed-point DSPs [15]. Its coarse architecture is shown in fig. 2.

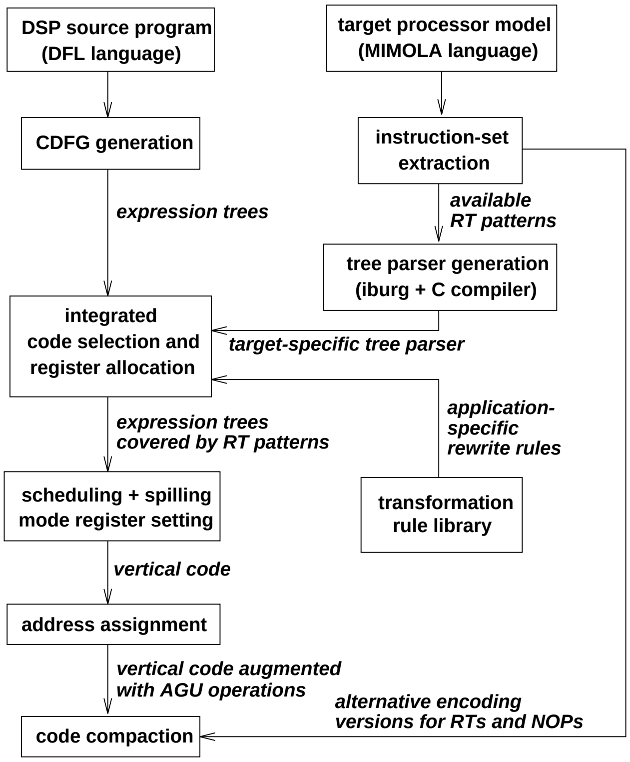

Figure 2: The RECORD compiler system

5.1 Retargeting

The *target processor model* is specified in the MIMOLA HDL, which corresponds to a subset of VHDL. In contrast to other related approaches, RECORD accepts both behavioral (instruction-level) and structural (RT-level) processor models. The user may select the most comfortable modelling abstraction level. Independent of its abstraction level, the target processor model is translated into a canonical behavioral model by *instruction-set extraction* (ISE). ISE eliminates structural details from the model, which are not required for code generation, and yields a set of RT patterns, i.e., primitive data path operations executable on the target processor as well as the corresponding instruction encodings (opcodes). From the extracted patterns set, an executable processor-specific code selector ("tree parser") is constructed by means of standard tools. After performing these steps once, RECORD is retargeted to the specified processor, and programs can be compiled into its machine code.

5.2 Code generation

In order to accommodate the special characteristics of DSP algorithms, RECORD uses DFL [4] as a source program language. The DFL program is compiled into an internal control/data-flow graph (CDFG) representation. The atomic CDFG entities are *expression trees* (ETs), for each of which instruction selection and register allocation are invoked. These phases are executed in an integrated fashion by means of the automatically generated processor-specific tree parser. The result is an optimal covering of the ET by means of RT patterns with respect to the accumulated costs of selected patterns. Covered ETs are passed to a scheduling phase which heuristically minimizes spill code for special-purpose registers. The result of scheduling is vertical (i.e. sequential) machine code. The vertical code is augmented with additional operations which implement necessary memory address computations. In this "address assignment" phase, several graph-based optimizations are applied, so as to achieve a high utilization of parallel address generation unit (AGU) capabilities. As a last compilation phase, the generated machine code is compacted, i.e., potential parallelism at the instruction-level is exploited. During this phase, constraints imposed by the instruction format need to be taken into account. This information is passed to code compaction by means of the encoding information obtained by ISE. A novel compaction techniques implemented in RECORD ensures optimal exploitation of parallelism within basic program blocks. The final result is a binary machine code listing.

5.3 Results

RECORD has been retargeted to several application-specific and standard DSPs, including TI's TMS320C25. Once a new HDL processor model is available, retargeting can be performed with a few CPU minutes on a SPARCstation. Thus, very short turnaround times are achieved for evaluating the impact of architectural changes in the target processor on program size and speed. Due to the use of exhaustive code optimization techniques, the compilation speed is rather low (approx. 1 instr. per CPU second) compared to a target-specific compiler. However, lower compilation speed is usually acceptable for embedded applications, where programs are often rather short. The point is that – at the expense of lower compilation speed – much higher code quality can be achieved. Fig. 3 shows an evaluation of relative code size for DSP benchmark programs [3] and the TMS320C25 target processor. The hand-written reference code is set to 100 %. The left columns show the size of code produced by TI's TMS320C25 C compiler. The right columns show the code size achieved by RECORD. For most cases, the RECORD code is more compact, and in total the average overhead of compiler-generated code as compared to hand-written assembly code has been halved.

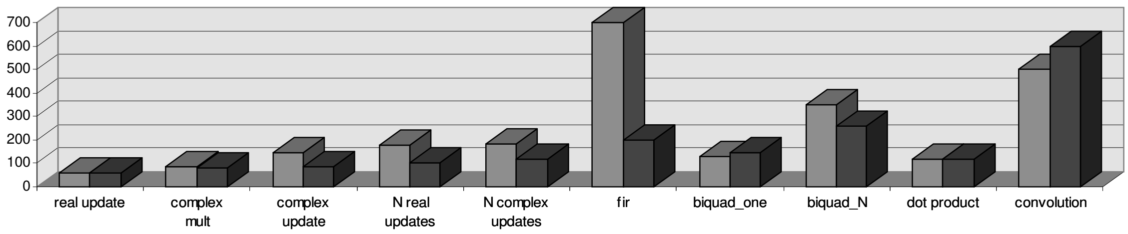

Figure 3: Experimental results: code quality

6 Conclusions

Design of embedded systems based on programmable processors demands for compilers capable of generating very efficient code. The key approach to achieve this goal is the development of novel code optimization techniques, while treating compilation speed with lower priority than in classical compiler construction. Simultaneously, the large variety of application-specific embedded processors create a need for retargetable compilers, that can be quickly adapted to new processors, so as to study the mutual dependence between processor architectures and program execution speed. In this paper, we have presented different recent approaches to retargetable and optimizing compilers with emphasis on DSPs. More and more system designs are primarily based on programmable processors rather than on custom hardware. It is therefore expected that such compilers can enable a productivity breakthrough in embedded system design, as they permit to take the step from assembly-level to high-level language software development.

References

[1] P. Paulin, M. Cornero, C. Liem, et al.: *Trends in Embedded Systems Technology*, in: M.G. Sami, G. De Micheli (eds.): *Hardware/Software Codesign*, Kluwer Academic Publishers, 1996

[2] P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[3] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[4] Mentor Graphics Corporation: *DSP Architect DFL User's and Reference Manual, V 8.2_6*, 1993

[5] World Wide Web: "http://www.eedesign.com/EEdesign/SoftCoretables.html" and "http://www.eedesign.com/EEdesign/HardCoretables.html"

[6] G. Araujo, S. Malik: *Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures*, 8th Int. Symp. on System Synthesis (ISSS), 1995, pp. 36-41

[7] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Code Optimization Techniques for Embedded DSP Microprocessors*, 32nd Design Automation Conference (DAC), 1995, pp. 599-604

[8] G. Araujo, S. Malik, M. Lee: *Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures*, 33rd Design Automation Conference (DAC), 1996

[9] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Storage Assignment to Decrease Code Size*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995

[10] G. Araujo, A. Sudarsanam, S. Malik: *Instruction Set Design and Optimizations for Address Computation in DSP Architectures*, 9th Int. Symp. on System Synthesis (ISSS), 1996

[11] C. Liem, P. Paulin, M. Cornero, A. Jerraya: *Industrial Experience Using Rule-driven Retargetable Code Generation for Multimedia Applications*, 8th Int. Symp. on System Synthesis (ISSS), 1995, pp. 60-65

[12] C. Liem, P.Paulin, A. Jerraya: *Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures*, 33rd Design Automation Conference (DAC), 1996

[13] S. Novack, A. Nicolau, N. Dutt: *A Unified Code Generation Approach using Mutation Scheduling*, chapter 12 in [2]

[14] R.M. Stallmann: *Using and Porting GNU CC* V2.4, Free Software Foundation, Cambridge/Massachusetts, 1993

[15] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997