Code generation for core processors

Peter Marwedel
Universität Dortmund, Informatik 12
44221 Dortmund, Germany
e-mail-alias: marwedel@acm.org

Abstract— This tutorial responds to the rapidly increasing use of cores in general and of processor cores in particular for implementing systems-on-a-chip. In the first part of this text, we will provide a brief introduction to various cores. Applications can be found in most segments of the embedded systems market. These applications demand for extreme efficiency, and in particular for efficient processor architectures and for efficient embedded software.

In the second part of this text, we will show that current compilers do not provide the required efficiency and we will give an overview over new compiler optimization techniques, which aim at making assembly language programming for embedded software obsolete. These new techniques take advantage of the special characteristics of embedded software and embedded architectures.

Due to efficiency considerations, processor architectures optimized for application domains or even for particular applications are of interest. This results in a large number of architectures and instruction sets, leading to the requirement for retargeting compilers to those numerous architectures. In the final section of the tutorial, we will present techniques for retargeting compilers to new architectures easily. We will show, how compilers can be generated from descriptions of processors. One of the approaches closes the gap which so far existed between electronic CAD and compiler generation .

1 Introduction

Embedded systems are systems that are tightly integrated (embedded) into a physical environment. They have to react to changes in their inputs. Various state-based specification methods have been proposed to specify such *reactive systems*. Embedded systems can be found, for example, in automotive electronics and telecommunications. In many cases, embedded systems do not come with a terminal and keyboard. In fact, the user frequently doesn't even realize that some information processing is taking place as a result of his input to a steering wheel or a pedal.

According to market analysts, the market of embedded systems is growing faster than the market for information technology in general. Many segments of the embedded systems market are *consumer markets*, with very short product lifetimes and short market windows. Hence, *time-to-market* is a deciding factor.

Cutting down the time to the market for products that become more and more complex is only feasible through *re-use*. This led to the reuse of larger and larger building blocks, with cores being the currently largest building blocks.

Another characteristic of the embedded systems market is the need for generating very *efficient* products, especially for portable equipment. For example, power consumption of portable equipment must be extremely low.

In order to incorporate late design changes, *flexibility* of the target technology is a must. This led to the use of processors in embedded systems. This, in turn, led to the use of embedded software. Unfortunately, current compiler techniques are not really adequate for applications and architectures of embedded systems. Hence, assembly language programming is still very common. This causes high development costs, an increased time to market and a low product dependability. Worst of all, it makes retargeting to new, more efficient processors almost impossible.

For embedded systems, there are less strict requirements for instruction set compatability between different target processors. Being able to retarget applications to the most efficient processor would be a competetive advantage.

The structure of this text is as follows: Section 2 describes cores in general and processor cores in particular. Current trends in compiler technology for embedded systems are the topic of section 3. Section 4 is exclusively focussing on retargetability. Finally, the last section contains conclusions. This way, each of the terms refered to in the title is covered in a section or a subsection.

2 Cores

2.1 General Cores

There has recently been a huge amount of interest in *cores*. Is there any way of defining this term? It is hard to give an exact definition, but cores can generally be described as pre-designed, pre-characterized building blocks that are larger than macroblocks (the largest blocks used so far). Available cores include: processor cores, communication cores (e.g. Ethernet cores) and controller cores (e.g. VGA controllers). Comprehensive lists of commercial cores are available on the web [11].

One can distinguish between *soft cores* and *hard cores*. Soft cores are essentially synthesizable, technology-independent models, whereas hard cores are layout models that come with data sheets. Both types of cores have their advantages.

The main reason that makes cores so popular is the need for *re-use*. Future chips, which are expected to contain more than a hundret million transistors, can only be designed in acceptable time frames if complex components are re-used.

Re-use of complex components has a number of advantages:

- It cuts down design time to the required level.
- Reuse of cores also improves the efficiency of the design, since cores are usually highly optimized. This applies especially in the case of hard cores.
- Testing is simplified because test engineers know the components they have to test from previous designs.

2.2 Processor cores

The class of *processor cores* represents an intersection between *cores* and *processors*. From their parent classes, they inherit a number of important characteristics. Due to the derivation from cores, processor cores provide re-use. Due to the derivation from processors, they provide flexibility. By changing the executed program, the overall behaviour can be very flexibly adapted to the design task at hand. This way, two otherwise conflicting goals can be met at the same time: core processors provide re-use *and* flexibility. It is this unique combination of features that makes core processors so popular.

Core processors include core versions of general purpose processors, such as core versions of various RISC architectures [16, 1, 28]

Examples:

1. The MiniRISC family of processors, which is instruction set compatible with the MIPS instruction set [28].

   As a member of this family, the CW 4001 requires just 4 mm^2 of silicon if manufactured in 0.5 μ technology and consumes 40 mW if running at 25 MHz

2. The ARM processor from Advanced Risc Machines Ltd.

   This processor is well-known for its low power requirement.

In addition to the standard processors, there are core versions of application *domain-specific* digital signal processors (DSPs). There are even *application*-specific instruction set processors (ASIPs).

A classification of processors is shown in fig. 1. This *processor cube* results from using three main criteria for classifying processors: availability of domain-specific features, availability of application-specific features, and the form in which the processor is available.

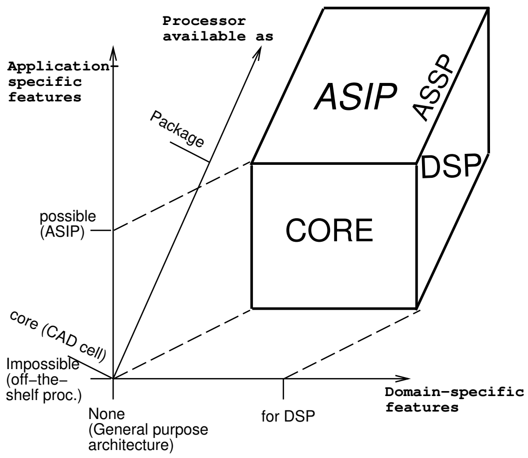

Figure 1: Cube of processor types

The meaning of these dimensions and their values is as follows:

1. *Form in which the processor is available*

   At every point in time, the design and fabrication processes for a certain processor have been completed to a certain extent. The two extremes considered here are represented by completely fabricated, packaged processors and by processors which just exist as a *cell* in a CAD system. The latter is called a *core processor*. In addition to cores, systems-on-a-chip may contain RAMs, ROMs, and special *accelerators*. With these, much of the performance penalty caused by the use of flexible processors can be compensated.

2. *Domain-specific features*

   Processors can be designed to be *domain-specific*. Possible domains are *digital signal processing* or *control-dominated applications*.

   DSP processors [20] contain special features for signal processing: multiply/accumulate instructions, specialized ("heterogenous") register sets, multiple ALUs, special DSP addressing modes (for example, for ring buffers), and saturating arithmetic operations.

3. *Application-specific features*

   At any point in time, the internal architecture of a processor may either be fixed or still allow configurations to take place.

   The two extremes considered here are: processors with a completely fixed architecture and ASIPs. Processors with a fixed architecture or *off-the-shelf processors* have usually been designed to have an extremely efficient layout.

   ASIPs are processors with an application-specific instruction set. Depending upon the application, certain instructions and hardware features are either implemented or unimplemented. Also, the definition of ASIPs may include *generic parameters*. By "generic parameters" we mean compile-time parameters defining, for example, the size of memories and the bitwidth of functional units. A very nice set of references to ASIPs is contained in a recent contribution by Paulin [32]. A well-known example is the EPICs architecture [41]. Optimal selection of instructions, hardware features and values for parameters is a topic which has recently received interest in the literature [3, 34, 15]. ASIPs have the potential of requiring less area or power than off-the-shelf processors. Hence, they are popular especially for low-power applications.

   The special class of ASIPs optimized for DSP is also called *application specific signal processors* (ASSP). These processors correspond to one of the edges of the processor cube.

In addition to the three coordinates, there are of course other criteria for classifying processors.

Currently, there is a huge amount of activity on designing processor hardware. However, designers frequently restrict their scope to optimizing the efficiency of processor hardware. The loss of efficiency due to the deficiencies in the current compiler technology usually does not lead to significant activities on compiler optimizations.

3 Compilers

3.1 Performance of Current Compilers

Problems with using current compiler technology for embedded processors have been mentioned by quite a number of industrial designers. Detailed numerical data has been published by a group of reseachers working at the University of Aachen. Researchers at Aachen have compared the size and the speed of assembly language library routines with the size and the speed of compiled code. According the results of this DSPStone benchmark project [42], overhead of compiled code (in terms of code size and clock cycles) typically ranges between 2 and 8. This means that typically up to 7/8th of all processor cycles are waisted if compilers are used. This translates into a huge loss of performance and electrical power and is clearly not acceptable for embedded systems. *Optimizations for low-power design should not be constrained to the hardware level. From an overall point of view, a* *highly optimizing compiler is one of the most important contributions to low power design.*

Due to the current lack of highly optimizing compilers, a major amount of applications is implemented in assembly languages. The exact percentage varies from company to company and from application to application. Paulin has computed this percentage for a closed set of applications [33]. He found that a major percentage of DSP applications and of controller applications is written in assembly languages.

Implementing complex systems using assembly languages has all the well-known disadvantages, for example a long time to the market, a low product quality and the inability of retargeting the application to new processors.

3.2 Requirements for Compilers for Embedded Processors

In order to make high-level programming for embedded systems a reality, adequate compilers have to be designed. These compilers have to meet the following requirements for the generated embedded code:

1. **Demand for compact code**

   In many applications (e.g. on heterogenous chips), not much silicon area is available to store the code. For those applications, the code must be extremely compact.

2. **Demand for extremely fast code**

   Related to the first requirement is the requirement to generate extremely fast code. Efficiency losses during code generation could result in the requirement to use faster processors in order to keep hard real-time deadlines. Such faster processors are more expensive and consume more power.

   The need for generating extremely fast code should have priority over the desire for short compilation times. In fact, compilation times which are somewhat larger than standard compilation times are acceptable in this environment. Hence, compiler algorithms, which so far have been rejected due to their complexity, should be reconsidered.

3. **Need for high dependability**

   Embedded processors directly interact with their environment and therefore must be extremely dependable. The requirement for absence of design faults should lead to the use of high-level languages and should exclude the still wide-spread use of assembly languages in this area.

4. **Constraints for real-time response**

   Embedded software has to guarantee a certain real-time response to external events. The issue of specifying, analyzing and checking timing constraints is covered, for example, in books by Ku, De Micheli and Gupta [18, 13], and papers by Boriello [8] and by Li, Malik, Wolfe [25]. Current compilers have no notion of time-constraints. Hence, generated assembly code has to be checked for consistency with those constraints. In many cases, error-prone, time-consuming simulations are used for this. We believe that it would be better to design smarter compilers. Such compilers should be able to calculate the speed of the code they produce.

5. **Support for DSP algorithms**

   Many of the embedded systems are used for digital signal processing. Development platforms should have special support for this application domain. For example, it should be possible to specify algorithms in high-level languages which support delayed signals, fixed point arithmetic, saturating arithmetic operators, and a definable precision of numbers.

6. **Support for DSP architectures**

   Many of the embedded processors are DSP processors. Hence, their features should be supported by development platforms.

3.3 New Optimization Techniques

Recent research has aimed at the design of new compiler optimizations taking the special characteristics of the application area and the target processors into account.

Standard techniques for assigning values to registers (called *register assigment*) assume homogenous register sets (all registers are equal). Embedded processors usually come with heterogenous register sets (not all register have the same functionality). For heterogenous registers, the overall performance is improved by taking advantage of features associated with some registers (e.g. incrementing certain registers in parallel). Wess [40], Araujo [4], Rimey [36], Bradlee [9] and Hartmann [14] have designed register assignment techniques which take heterogenous register sets into account.

Many of the popular DSPs include so-called *parallel instructions*. For example, the Motorola MC 56000 allows parallel move operations (operations which can be performed in parallel to arithmetic operations). Not taking advantage of this parallelism means loosing a factor of two in the performance. Techniques for exploiting this type of parallelism have been used in the context of microprogramming. Due to recent improvements in combinatorial optimization techniques, optimal algorithms have become feasible [24]. Other compaction algorithms have been described by Timmer [39], Strik [37], and Nicolau [31].

Many DSPs have multiple *operation modes* (in the context of microprogramming, this concept was called *residual control*). For example, DSPs may be in saturating arithmetic mode or in wrap-around arithmetic mode. Switching from one mode to the other requires executing *mode changing instructions*. The issue for compilers is to minimize the number of mode-changing instructions. Liao [26] has designed an algorithm for this purpose.

A few DSPs support *multiple memory banks*. Whenever the arguments of a binary operation are available in two different memory banks, the operation executes faster. Assigning variables to memory banks such that as many operations as possible will find their operands in different banks is an optimization that can be more easily performed by a compiler than by an assembly language programmer. Sudarsanam has published an algorithm [38] implementing this optimization.

Several DSPs include special address generation units. With these, incrementing an address register does not require an extra instruction or cycle. As a result, it is desirable to assign variables to memory such that as many variable accesses as possible refer to adjacent memory locations. Bartley [6], Liao [26] and Leupers [21] have described algorithms for this optimization.

All these optimizations can be implemented in target-specific as well as in retargetable compilers.

4 Retargetable Compilers

4.1 What is Retargetablility?

A design automation tool (e.g. a compiler) is said to be *retargetable* if it can be applied to a range of different targets, in particular to a range of different target processors. This means that the target model cannot be an implicit part of the tool's algorithm, but must be explicit.

There are different levels of retargetability. The lowest level is *portability*, which means that the tool can be easily modified to handle a new target. The highest level is *target independence*, which means that the target model is completely explicit and no assumption is made in the tool's algorithm.

4.2 Why Retargetability?

Retargetable compilers are more difficult to write than target-specific compilers. Why do we go through the effort of designing retargetable compilers? There are several reasons for this:

- Core processors are mostly used for embedded systems and for these systems efficiency is extremely important. Hence, processor architectures may vary from application to application. This is possible, since there is no need for code compatability, because there are no "user programs".

  Most current compilers are target-specific. We believe that retargetability will be required, at least for a (possibly limited) range of ASIP target architectures. ASIPs frequently come with generic parameters, such as the bitwidth of the data path, the number of registers, and the set of hardware-supported operations. The *user* should at least be able to retarget a compiler to every set of parameter values. A larger range of target architectures would be desirable to support experimentation with different hardware options, especially for partitioning in hardware/software codesign.
- For embedded processors, there may be only a small amount of applications per processor architecture. Hence, designing compilers quickly and economically is important.
- Understanding the basic mechanism that is required for retargeting a compiler also helps designing compilers for fixed architectures.
- We have seen several cases in which the attempt of generalizing a target-specific compiler into a retargetable compiler failed. If retargetability it required, it has to be considered from the very beginning. It cannot be added later on.

4.3 A Detailed Example: RECORD

In order to understand the essential mechanisms in a retargetable compiler, we will use the RECORD compiler [22] as an example.

4.3.1 Overview

Fig. 2 provides an overview over the RECORD compiler (actually: compiler generator).

RECORD compilers compile programs written in the DSP-specific programming language DFL [30] into binary code. RECORD compilers are generated from a description of the target processor.

As a special characteristic of RECORD, this description of the target may be at different levels of abstraction: it may range from an RT-level netlist to an instruction set description. RT-level netlist decriptions are accepted because some ASIPs may be defined at that level and because this simplifies the analysis of architectural tradeoffs. Furthermore, it provides a bridge between ECAD (netlist) and compiler (instruction set) domains.

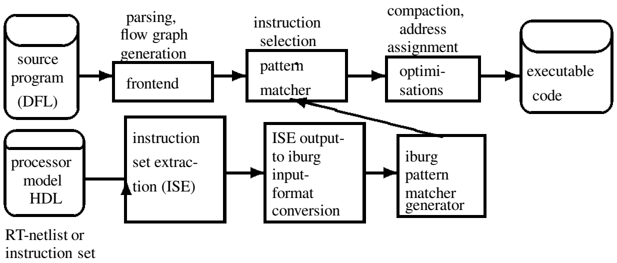

Figure 2: Global view of RECORD

4.3.2 Instruction Set Extraction

The goal of instruction set extraction (ISE) [23] is to generate an instruction set description from an RT-level netlist. For each memory or register input, ISE traverses the netlist from that input to memory or register outputs (opposite to the direction of the data-flow). For each traversal, it collects the transformations that are applied to the data (e.g. add operations) and also the control requirements (e.g. set ALU input to '0' to perform an add). Control requirements have to be met by proper conditions for instructions bits, which can be found by justification. The net effect of ISE is to generate, for each register or memory, a list of assignable expressions and the corresponding instruction bit settings.

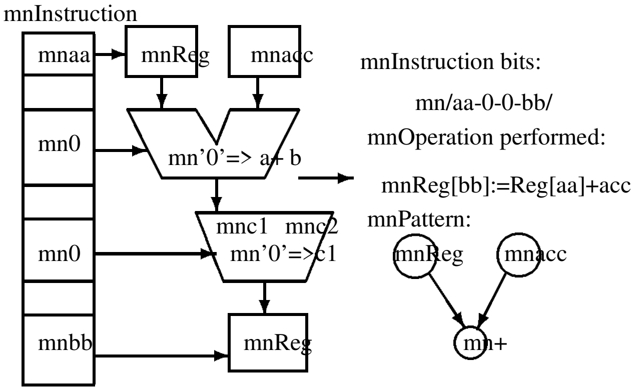

Figure 3: Instruction extraction

As a special case, the netlist used as input to ISE is allowed to contain only a single RT-component having the instruction set as its behaviour. In this case, ISE essentially just generates a normalized description of the processor behaviour, making the processor description more or less independent of syntactical and other variances of the description style.

4.3.3 Generation of the Pattern Matcher

The essential operation of any compiler consists of finding appropriate instruction set patterns for implementing or *covering* the pattern representing the program. Fig. 5 shows how a data flow graph representing a program can be covering by instruction patterns. Note that we have just represented the operations and not the registers.

It is well known that the generation of optimum graph covers is an NP-complete problem. Most approaches are therefore based on heuristic decompositions of graphs into trees or take advantage of special architectural situations in which an optimum decomposition can be found.

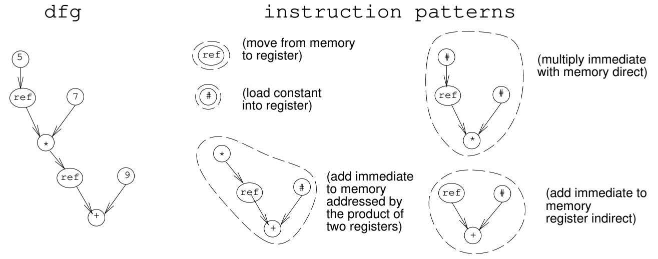

Figure 4: Data flow graph and instruction patterns

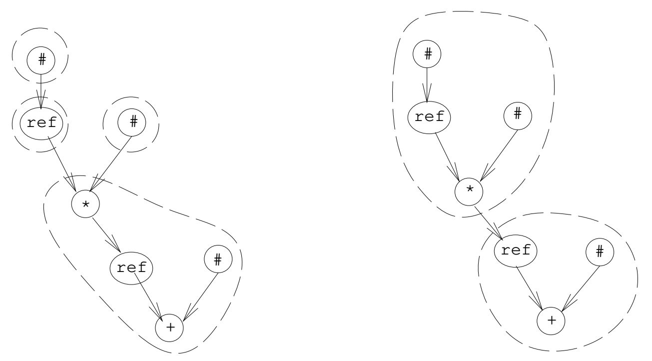

Figure 5: Covering data flow trees with instruction patterns

Early approaches for covering trees were also based on heuristics (like Cattell's maximum munching method [10]). More recently, Aho et al. proposed an dynamic programming algorithm for generating an optimum cover [2]. This approach, however, mainly aims at code generation for homogenous register architectures. Heterogenous register architectures can be handled with tree parsing [5]. The `iburg` tool set allows generating pattern matchers for any given target instruction set automatically.

This is also the tool used in RECORD for selecting instructions. In order to generate optimized code, RECORD uses algebraic rules for transforming the original data flow tree into equivalent ones and calls the `iburg`-matcher with each tree. The tree requiring the smallest number of covering patterns is then selected. This approach is feasible due to the high speed of `iburg`-based matchers. The `iburg`-matcher produces a sequence of instruction patterns.

4.3.4 Optimizations

This sequence can still be optimized, because the `iburg` model does not reflect parallel operations or optimized accesses to memory. Such optimizations (in effect, all optimizations mentioned in section 3) can be implemented in this phase.

4.3.5 Results

Table 1 shows a comparison between code produced by RECORD and code produced by a target-specific compiler for the ti C25, using algorithms from the DSPStone benchmark as examples.

Program	TI C Comp.	RECORD
real_update	60	60
complex_multiply	84	79
complex_update	148	86
N_real_updates	180	100
N_complex_updates	182	118
fir	700	200
iir_biquad_one_section	130	145
iir_biquad_N_sections	300	258
dot_product	120	120
convolution	500	600

Table 1: Size of compiled programs in relation to assembly code (%)

In six out of ten cases, RECORD outperforms the target-specific compiler, even though it does not contain any standard optimization technique (such as constant folding). This shows that retargetable compilers can compete with todays target-specific compilers. On the other hand, extremely target-specific optimizations will always require a target-specific compiler.

4.4 Key Approaches To Retargetability

A recent book provides a comprehensive survey of the major projects aiming at (retargetable) compilers for embedded processors [29].

In their project, Paulin et. al have initialy focussed on the FlexWare approach to compiler generation [27]. More recently, they are using a rule-based approach for code generation.

Goossens et. al have designed the CHESS compiler [19, 35], which uses the special language nML [12] for instruction set description.

Wess is using Trellis diagrams for modelling target architectures [40].

4.5 Generation of Self-Test Programs with Retargetable Compilers

Testing of processor cores can be performed by running self-test programs on the processor to be tested. Automatic generation of self-test programs is possible with a special retargetable compiler that is able to propagate values just like ATPG tools. The first approach to this [17] has recently been refined by Bieker [7].

5 Conclusion

Currently, there is a significant shift in how embedded systems are designed. The use of cores, and -in particular- the use of processor cores, for chip-level (rather than board-level) integration leads to the requirement of generating efficient embedded software. Due to the lack of adequate compiler optimization algorithms, new research on compilers is required. The research can take advantage of the special characteristics of embedded software. Some recently published algorithms show that progress in this area is indeed feasible.

Due to the need for domain-specific or even application-specific efficient architectures, a large number of processor core architectures with different instruction sets is predicted to exist during the next years. In order to allow high-level language programming of these architectures, retargetable compilers have been proposed. Recently published algorithms and results indicated that these can in fact be designed and their performance competes favourably with currently available commercial target-specific compilers.

References

[1] Advanced RISC Machines Ltd. ARM. web pages. *http://www.arm.com/*, 1995.

[2] A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. on Programming Languages and Systems, Vol. 11*, pages 491–516, 1989.

[3] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. An ASIP instruction set optimization algorithm with functional module sharing constraint. *Int. Conf. on Computer-Aided Design (ICCAD)*, pages 526–532, 1993.

[4] G. Araujo and S. Malik. Optimal code generation for embedded memory no-homogenous register architectures. *8th Int. Symp. on System Synthesis (ISSS)*, pages 36–41, 1995.

[5] A. Balachandran, D.M. Dhamdere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Comp. Lang.*, 15:127–140, 1990.

[6] D.H. Bartley. Optimizing stack frame accesses with restricted addressing modes. *Software - Practice and Experience*, 22:101–110, 1992.

[7] U. Bieker and P. Marwedel. Retargetable self-test program generation using constraint logic programming. *32nd Design Automation Conference*, 1995.

[8] G. Boriello. Software scheduling in the co-synthesis of reactive real-time systems. *Proceedings of the 31th Design Automation Conference*, pages 1–4, 1994.

[9] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–131, 1991.

[10] R.G.G. Cattell. Formalization and automatic derivation of code generators. Technical report, PhD thesis, Carnegie-Mellon University, Pittsburgh, 1978.

[11] EEDesign. Tables of hard cores and soft cores. *http: //www.eedesign/ Resources.html*.

[12] A. Fauth and A. Knoll. Automated generation of DSP program development tools using a machine description formalism. *Int. Conf. on Audio, Speech and Signal Processing*, 1993.

[13] R. Gupta. *Co-synthesis of Hardware and Software for Embedded Systems*. Kluwer Academic Publishers, 1995.

[14] Hartmann. Combined scheduling and data routing for programmable ASIC systems. *EDAC*, pages 486–490, 1992.

[15] I.-J. Huang and A. Despain. Generating instruction sets and microarchitectures from applications. *Int. Conf. on CAD (ICCAD)*, pages 391–396, 1994.

[16] Intel Corp. web pages. *http://www.intel.com/product/tech-briefs/-index.html*, 1996.

[17] G. Krüger. A tool for hierarchical test generation. *IEEE Trans. on CAD, Vol. 10*, pages 519–524, 1991.

[18] D. Ku and G. De Micheli. *High Level Synthesis Under Timing and Synchroniszation Constraints*. Kluwer Academic Publishers, 1992.

[19] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: retargetable code generation for embedded DSP processors. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer Academic Publishers*, 1995.

[20] E. Lee. Programmable DSP architectures, parts i and ii. *IEEE ASSP Magazine*, Oct. 1988 & Jan. 1989, 1988.

[21] R. Leupers. Algorithms for address assigment in DSP code generation. *ICCAD*, 1996.

[22] R. Leupers. Retargetable generator of code selectors from HDL processor models. *European Design and Test Conference (ED &TC)*, 1997.

[23] R. Leupers and P. Marwedel. Instruction set extraction from programmable structures. *Proc. Euro-DAC*, 1994.

[24] R. Leupers and P. Marwedel. Time-constrained code compaction for DSPs. *Int. Symp. on System Synthesis (ISSS)*, 1995.

[25] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *Int. Conf. on Computer-Aided Design (ICCAD)*, pages 380–387, 1995.

[26] S. Liao, S. Devadas, K. Keutzer, S. Tijang, and A. Wang. Storage assignment to decrease code size. *Programming Language Design and Implementation (PLDI)*, 1995.

[27] C. Liem and P. Paulin. FlexWare – A flexible firmware development environment. *Proc. European Design & Test Conference (EDAC-ETC-EUROASIC)*, pages 31–37, 1994.

[28] LSI Logic Inc. web pages. *http://www.lsil.com/products/unit5_5.html*, 1996.

[29] P. Marwedel. Introduction. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors*, Kluwer, 1995.

[30] Mentor Graphics Corporation. DSP architect DFL user's and reference manual. *V 8.2_6*, 1993.

[31] S. Novack, A. Nicolau, and N. Dutt. A unified code generation approach using mutation scheduling. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer Academic Publishers*, 1995.

[32] P. Paulin, M. Cornero, C. Liem, F. Na abal, C. Donawa, S. Sutarwala, and C. Valderrama. Trends in embedded systems technology: An industrial perspective. *in: M.G. Sami, G. De Micheli: Hardware/Software Codesign, Kluwer Academic Publishers*, 1996.

[33] P. Paulin, C. Liem, T. May, and S. Sutarwala. DSP design tool requirements for embedded systems: A telecommunications industrial perspective. *Journal of VLSI Signal Processing*, pages 23–47, 1995.

[34] J. V. Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for ASIPs. *7.th Int. Symposium on High-Level Synthesis*, pages 11–16, 1994.

[35] J. V. Praet, D. Lanneer, G. Goossens, W. Geurts, and H. De Man. A graph based processor model for retargetable code generation. *European Design & Test Conference*, 1996.

[36] Rimey and Hilfinger. Lazy data routing and greedy scheduling for application-specific processors. *21st Annual Workshop on Microprogramming (MICRO-21)*, pages 111–115, 1988.

[37] M. Strik, J. van Meerbergen, A. Timmer, and J. Jess. Efficient code generation for in-house DSP cores. *European Design & Test Conference*, pages 244–249, 1995.

[38] A. Sudarsanam and S. Malik. Memory bank and register allocation in software synthesis for ASIPs. *Intern. Conf. on Computer-Aided Design (ICCAD)*, pages 388–392, 1995.

[39] E. Timmer. Conflict modelling and instruction scheduling in code generation for in-house DSP cores. *32th Design Automation Conference*, 1995.

[40] B. Wess. Code generation based on Trellis diagrams. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer Academic Publishers*, 1995.

[41] R. Woudsma. EPICS, a flexible approach to embedded DSP cores. *Int. Conf. on Signal Processing and Applications and Technology*, 1994.

[42] V. Zivojnovic, J.M. Velarde, and C. Schlager. DSPstone: A DSP-oriented benchmarking methodology. *Internal Report, Aachen University of Technology*, 1994.