Code Generation for Embedded Processors: An Introduction

P. Marwedel

1 New, flexible target technologies

As the tendency towards more complex electronic systems continues, many of these systems are equipped with embedded processors. For example, such processors can be found in cars, and in audio-, video-, and telecommunication-equipment. Essential advantages of these processors include their high flexibility, short design time and (in the case of off-the-shelf processors) full-custom layout quality. Furthermore, they allow an easy implementation of optional product features as well as easy design correction and upgrading. Furthermore, processors are frequently used in cases where the systems must be extremely *dependable*[1] [32]. In such cases, the re-use of the design of an off-the-shelf processor greatly simplifies dependability analysis.

This contrasts with the limitations of *application-specific circuits (ASICs)*: due to their low flexibility, the cost for the design and fabrication of ASICs is still very high. Furthermore, this low flexibility makes a short time-to-market more difficult to achieve. Dependability analysis costs may even exclude ASICs as a target technology.

A short time-to-market can be achieved with *field programmable gate arrays (FPGAs)*. But FPGAs are not area-efficient. For example, multipliers require a large proportion of the available area. Furthermore, FPGAs with programmable interconnect usually do not allow high clocking frequencies.

Embedded processors come in different types. We will classify them according to three different criteria: flexibility of the architecture, architectural features for certain application domains, and the form in which the processor is available. The three criteria can be used as dimensions to form a 3D processor type space (see fig. 1).

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

1. *Architectural features for certain application domains*

   Processors can be designed for restricted or for larger classes of application areas. The two cases considered here are: *"General purpose architecture" (GPA)* and *"digital signal processors" (DSPs)*.

   The term "general purpose processor" is used for processors which do not have particular support for special applications, such as Fourier transforms or digital filtering.

[*]This is a reprint of material from the book "P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer, June 1995". Kluwer copyright restrictions apply. Refer to the book for correct crossreferences to other contributions in the book. This work has been supported by the Commission of the European Communities under contract BRA 9138 (CHIPS).

[1]The term *dependability* includes all aspects of system safety, for example: absence of design faults, comprehensive testing after manufacturing, *reliable* components, and error-detection and recovery mechanisms.

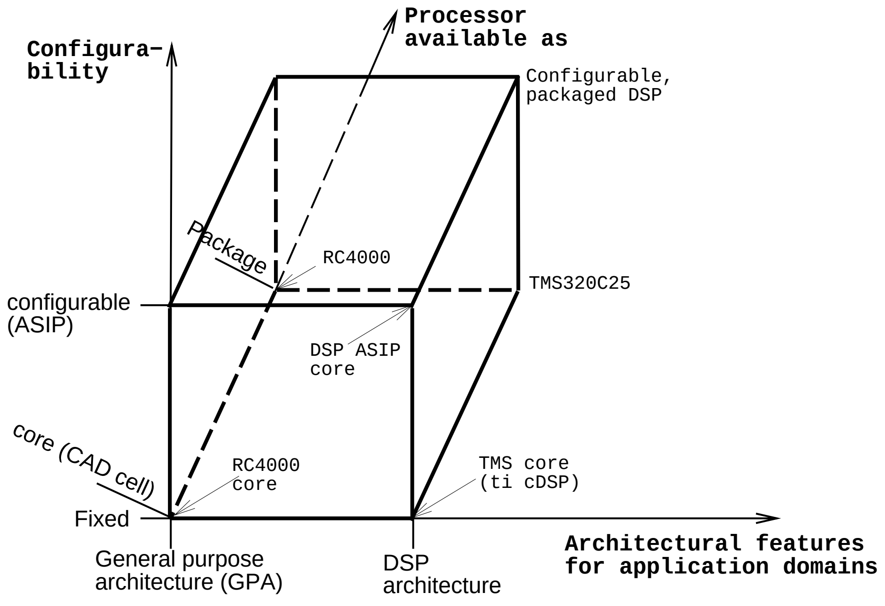

Figure 1: Cube of processor types and some examples

DSP processors [33] 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.

DSPs should also exhibit *data-independent instruction execution times* or should at least exhibit only small variations of the execution time. Otherwise, it would be to hard to predict their real-time response. This requirement affects the design of the memory system (use of static RAM, absence of caches) as well as the design of arithmetic algorithms (e.g. for multiplication and division).

2. *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 also called a *core processor* (see fig. 1). *In-house cores* are proprietary cores available just within one company. They usually have some architectural flexibility. Cores can be instantiated from the library to become part of a larger *heterogenous chip* (see fig. 2). In addition to cores, heterogenous chips may contain RAMs, ROMs, and special *accelerators*. With these, much of the performance penalty caused by the use of flexible processors can be compensated.

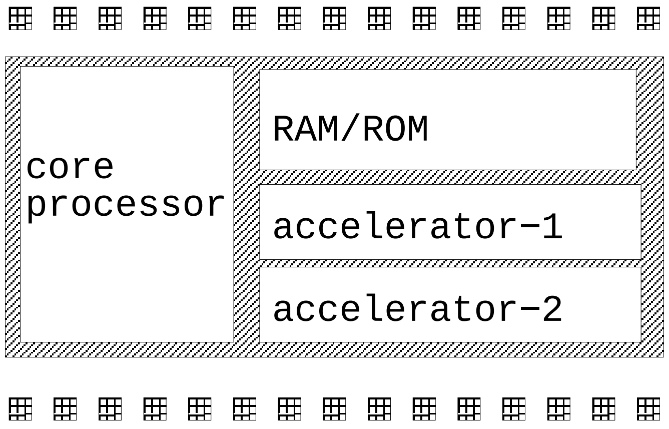

Figure 2: Core processor as part of a heterogenous chip

3. *Configurability of the processor*

   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 *"application-specific instruction set processors" (ASIPs).*

   Processors with a fixed architecture or *off-the-shelf processors* (see fig. 3) have usually been designed to have an extremely efficient layout. Some of them have passed verification procedures, allowing them to be employed in safety-critical applications.

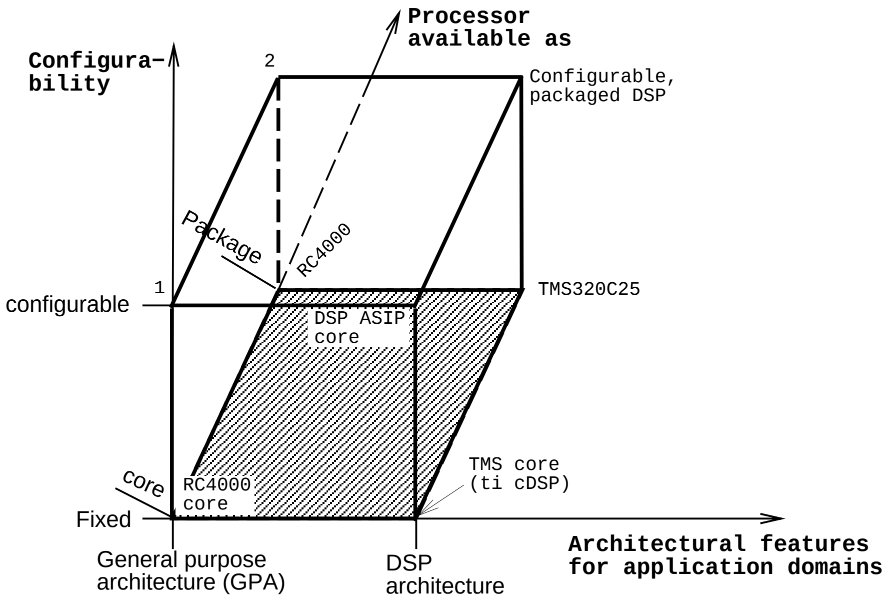

Figure 3: Off-the-shelf processors

In contrast, *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. Optimal selection of instructions, hardware features and values for parameters is a topic which has recently received interest in the literature [5, 38, 23]. ASIPs have the potential of requiring less area or power than off-the-shelf processors. Hence, they are popular especially for low-power applications.

Corners 1 and 2 of fig. 3 correspond to general purpose architectures (e.g. standard microprocessors) which can be tailored towards a certain design, for example, by configuring the number of address lines, interrupt lines, or power vs. speed options. We could imagine to build processors which can be configured just like FPGAs can be configured. For example, it might be possible to save power by disabling parts of the processor. We could also think of processors as blocks in FPGAs. Unfortunately, no such processor is known to the authors.

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

The selection of a certain processor type is very much influenced by the application at hand. For safety-critical automobile applications for example, dependability is the driving requirement. Hence, validated off-the-shelf processors may be preferred. For portable equipment, power consumption may be the driving requirement, leading to the use of power-efficient ASIPs.

2 Design scenarios for embedded processors

In this section we shortly examine procedures for designing with embedded processors. In general, systems will consist of both processors and special application-dependent hardware. Hence, both the software that will run on the processor as well as the hardware have to be designed. Therefore, hardware and software design are related and the resulting design process is called *hardware-software codesign*. The term is not precisely defined, but fig. 4 is a generally accepted view of the design flow in hardware-software codesign.

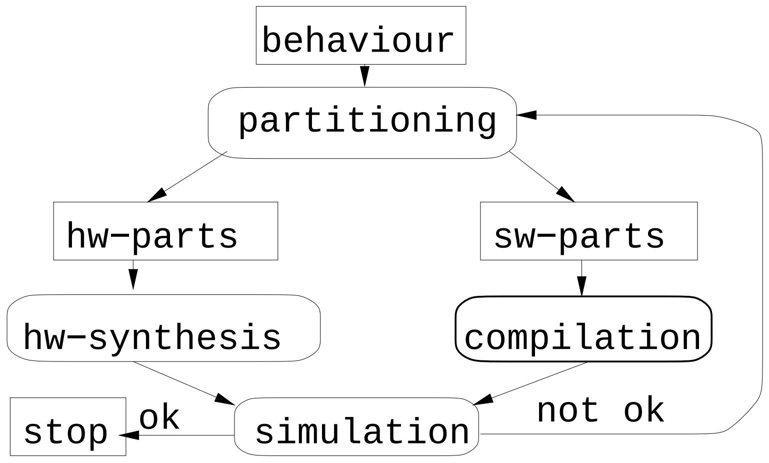

Figure 4: HW-SW-codesign flow

The designer starts with an overall behavioural specification, for example using SpecCharts [16], Hardware-C [25], a single C process [13] or sets of C processes. The specification is then partitioned into software parts and hardware parts. Software parts (e.g. a fraction of the C program) are later compiled onto an envisioned processor. Hardware parts (possibly translated into Hardware-C) are used as input to a hardware synthesis system. Currently, the state-of-the-art does not guarantee that the combined design meets performance and other requirements. If the requirements are not met, the design process must be repeated.

Let us now have a closer look at the compilation process within this design flow. Currently, compilers for fixed target architectures are employed for this. We argue that they do not provide the flexibility we need. During the design, we want to experiment with different target processors. We want to try out different ASIP parameters, and we want to leave out or add certain processor features. Code generation which supports this process has to be retargetable. "Retargeting" in this context means: *fast and easy* retargeting, simple enough to be *handled by the user*.

In the codesign environment, simulations are needed at different levels. First of all, the specification has to be simulatable. This is required in order to check whether or not the specified algorithm really performs the intended function. Later, the generated code will be simulated using an instruction set model of the processor. This simulation can take the generated hardware parts into account. Finally, the processor may also be simulated at the structural level.

If either the design procedure or the generated design could be proven to be correct, this simulation would not be required. However, at the current state of the art, neither of the two can be proven correct except in very limited cases.

Many of the codesign applications in this book will consider DSP applications in particular. We will therefore zoom-in on the design of DSP systems.

An immediate observation in this area is the fact that large amounts of data have to be handled in simulations. For example, in video applications, one would like to simulate digital signal processing of movies in real-time. This speed cannot be obtained with general simulators. Therefore, special simulators have been designed.

Furthermore, considerable effort for programming DSP processors seems to be typical for this application area (this was confirmed by several design groups and also mentioned as result of a survey at Bell Northern Research [27]). Currently, simple assemblers for fixed architectures are the most frequently used code generation tools. Assemblers are used, because current compilers have problems with exploiting the special architecture of DSP processors (heterogenous register sets etc.). The use of assemblers results in a high development effort. Also, the reusability is rather low.

The situation is slightly better if compilers are employed. Currently, compilers for fixed targets are dominating the market. Switching from one target architecture to the next, however, requires changing the compiler. This can result in a number of side-effects: e.g. different compiler pragmas[2], different code quality and a different compilation environment.

We conclude, that a number of design tools for designing with embedded processors is urgently needed. Especially important are: fast simulators, hardware/software partitioning tools, and compilers. In this book, we will focus on compilers and code generation.

3 Requirements for design tools

An analysis of the above applications reveals that the design tools for embedded processors have to be different from design tools for larger computer systems in the following respects:

1. **Need for high dependability**

   Embedded processors directly interact with their environment and therefore must be extremely dependable. This is especially true for safety-critical applications, where this requirement domi-

[2]Pseudo comments used to control the compiler.

nates all others. 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.

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

   Embedded processors have to guarantee a certain real-time response to external events. This requirement is not considered by current development kits. 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 and should at least be able to compare it against timing constraints. More sophisticated compilers could use timing constraints to control optimization efforts.

3. **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. Increased power consumption is frequently not acceptable for portable applications.

   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.

4. **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.

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. On the other hand, there is also good news for compiler writers: some language constructs causing a lot of troubles otherwise are hardly needed in this area. For example, pointers can usually be avoided.

6. **Support for DSP architectures**

   Many of the embedded processors are DSP processors. Hence, their features should be supported by development platforms. Compilers for DSP architectures should be able to exploit

   - *Specialized, non-homogenous register sets*

     DSP processors frequently come with specialized, non-homogenous register sets. Such register sets are important for the performance and presumably cannot be removed to simplify the task of writing compilers.

   - *The (possibly limited) form of parallel execution usually possible with such processors*

     Note that even off-the-shelf processors such as the TMS 320C25 require exploitation of parallelism. For example, the MAC (multiply and accumulate) instruction performs three assignments. Some *very long instruction word (VLIW)* core processors allow even more parallelism. The inability of current compilers to exploit parallelism seems to be one major source for their inefficiency.

- *Special DSP algorithm support*

  DSP architectures contain special hardware for supporting DSP algorithms, such as ring buffers, bit-reversed addressing for supporting fast Fourier transforms (FFTs), multiplier-adder chains, etc. This hardware has to be exploited by compilers.

7. **Tight coupling between code selection, register allocations, scheduling, and compaction**

   Code generation consists of a number of tasks which can be distinguished: code selection, register allocation, scheduling and compaction. There is no standard definition of these tasks, but the following definitions seem to reflect the meaning commonly used.

   Code selection is the optimized mapping of a certain intermediate representation of the source program to machine operations. Machine operations can be encoded in (partial) machine instructions. Each partial machine instruction specifies machine instruction bits which cause computed values to be transfered to registers or memory locations. In the case of parallel machines (such as VLIW machines), several such transfers can be encoded in one machine instruction. Otherwise, only a single transfer can be encoded in an instruction.

   Register allocation maps program variables and intermediate results to machine registers or register sets. This task also includes the allocation of registers for passing the arguments and results of procedures and functions.

   Scheduling is the task of establishing a partial order among the machine operations selected during code selection. This partial order has to maintain the semantics of the program. In the case of parallel machines, it has to allow as much parallel execution as possible.

   Compaction is the task of assigning partial machine instruction to machine instructions. As a result, a total order of machine operations is fixed. Of course, this order has to be compatible with the partial order computed during scheduling. Compaction is not needed if each transfer corresponds to one instruction. Note that the MAC instruction of the very popular TMS320C25 encodes three transfers. Since compaction is not needed for many machines, the distinction between scheduling and compaction is frequently not made.

   Unfortunately, code selection, register allocation, scheduling, and compaction are mutually dependent (see pp. **??**). Choosing any sequence for these tasks can result in non-optimal code. Clever cooperation of these tasks is called *phase-coupling*. Various forms of phase-coupling have been used, e.g. prediction of the effect of following phases or iterations between phases. Tight, backtrackable *integration* of all phases has usually been avoided.

   Code generation requires a model of the target machine. The processor model used by the compiler should by preference contain the necessary structural information, to model e.g. pipelining effects and effects of "busy" functional units [19].

8. **Retargetable compilers**

   Especially for the design with ASIP core processors, one would like to create code for the range of available architectures. One would like to allow application-dependent changes of architectures and instructions and still have compiler support.

   In the current application domain, there is no need for instruction set compatability between systems, because there are no "user programs". Hence, target processors can be selected according to the requirements at hand. Unfortunately, this selection is made difficult due to restricted support by development platforms for some processors. This means: the underlying hardware technology is rather flexible (especially in the case of ASIPs), but CAD technology is not. Therefore, the current CAD technology is a bottleneck that should be removed by designing *retargetable compilers.* Such compilers can be used for different target processors, provided that a target machine description is given to them.

Many of the current compilers are more or less target-specific. We believe that retargetability will be required, at least for a (possibly limited) range of target architectures. Processor cells 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.

4 Related Work

In this section on related work, we will focus on approaches for designing retargetable compilers. Methods addressing other requirements will be mentioned only briefly.

4.1 Retargetable Compilers

Techniques for retargetable compilers have been published in three different contexts: compiler construction, microprogramming and computer-aided design. We will give a short overview of the contributions for each of the three areas.

4.1.1 Compiler construction

Retargetability has been a design goal for compilers for quite some time.

In the UNCOL [11] approach, it was proposed to compile from m source languages to n target machine languages by using m *front-ends* to compile to a common intermediate format and then using n *back-ends* to translate from that format to the target language. This way, $m + n$ tools are required instead of the $m * n$ tools for a direct translation from each source language to each target language.

This approach turned out not to be feasible in general but to work well for restricted sets of source and target languages. For example, it worked quite well for the compilation from imperative languages to all processors used in Apollo workstations. Furthermore, it worked quite well in a compiler based on formal methods [40]. In both cases, backends were written manually.

Compilers for new architectures can be generated in less time, if pattern matching and covering are used, such as in the portable GNU C compiler [41]. The following is a short introduction to pattern matching and covering techniques. Any internal format for an intermediate language is based on *dataflow graphs (DFGs)* (see fig. 5 (left)). In that figure, `ref` denotes a reference to (background) memory. The result is implicitly assumed to be stored in a register.

In the simplest form, each of these graphs represents an assignment. In more elaborated forms, these graphs are generated by dataflow analysis algorithms. Each of the target machine instructions can be represented by a small graph, too (see fig. 5 (right)). This graph describes the behaviour implemented by executing that instruction. Nodes labelled `+`, `*` or `ref`, these are assumed to have register arguments and register destinations, if arguments or destinations are not shown. The character `#` represents any constant. In order to implement the source program, the dataflow graph has to be covered by machine instruction patterns.

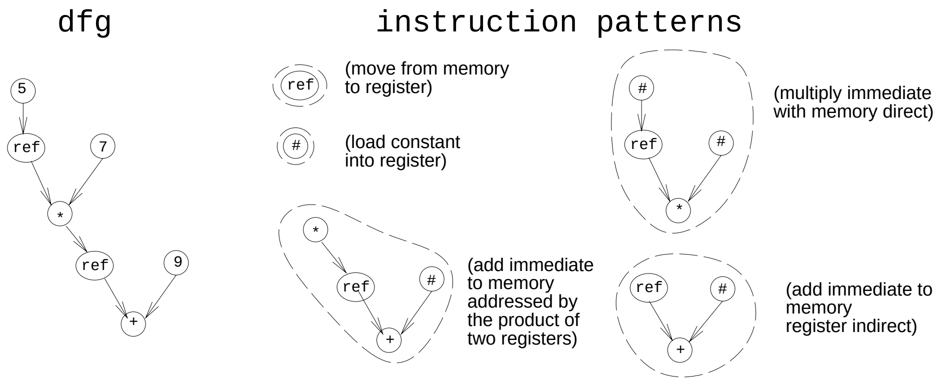
dfg
5
ref
7
*
ref
9
+
instruction patterns
ref
(move from memory to register)
#
(load constant into register)
*
ref
#
+
(add immediate to memory addressed by the product of two registers)
#
ref
#
*
(multiply immediate with memory direct)
ref
#
+
(add immediate to memory register indirect)

Figure 5: Dataflow graph and instruction patterns

Early compilation schemes were based on very simple instruction sets. It was shown that the problem of generating optimal coverings is NP-complete even for these instruction sets. Optimal polynomial algorithms were described for restricted cases, like that of dataflow *trees* [2].

Exploitation of complex instruction sets was a major goal in the *production quality compiler-compiler project* (PQCC) at CMU. Cattell [9] proposed the heuristic *maximum munching method* (MMM) to generate "good" coverings. With this technique, the largest instruction matching a section of the dataflow graph is selected and matching then continues for the remaining parts of the graph. The result of this technique can be seen in fig. 6 (left). Due to the presence of the large 4-node instruction, the DFG is covered by a total of four instruction patterns. Edges connecting instruction patterns correspond to registers.

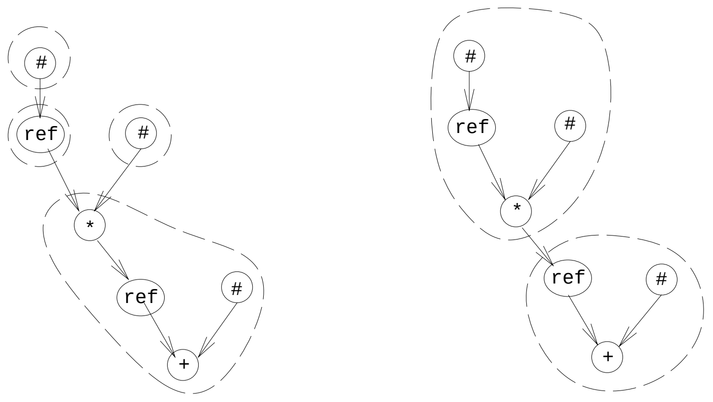

Figure 6: Coverings generated by 2 pattern matching methods

The state-of-the-art in the early eighties was described in a survey by Ganapathi, Hennessy et al. [17].

A method which generates optimum coverings for complex instruction sets for dataflow *trees* was proposed by Aho, Ganapathi and Tjiang [1]. Fig. 6 (right) shows the result of this algorithm for our earlier example, assuming equal costs for each instruction. The method is based on dynamic programming. It is optimal in the sense that the accumulated cost for covering instructions is minimal, provided that the processor has a single, homogenous register set. Also, it is assumed that the cost of instructions can be defined independently of each other. Keutzer and Wolf have applied this technique to technology mapping in logic synthesis [24].

Paulin has applied the method of Aho et al. to the design of a retargetable compiler for DSP applications (see pp. **??** of this book).

Another important example is that of the CBC-Compiler [15]. The CBC-Compiler assumes that the target machine is described in the special language `nML`. Information about this language and its applications can be found on pp. **??** of this book.

A major new entry into the scene is the CHESS compiler. Details about this compiler and associated references can be found on pp. **??**.

Code selection has also been considered in detail by Emmelmann [12].

4.1.2 Microprogramming

Additional work on compilation was done in the microprogramming context. In contrast to work on early compilers, this work had to take scheduling and other pecularities into account.

More than a decade ago, several researchers discovered independently that code generation can be modelled as parsing the source language with respect to a grammar which can be generated from a structural hardware description (see e.g. Evangelisti [14] and Anceau [3]). Unfortunately, the sequence in which applicable derivations are selected, affects the resulting code quality. General, good methods for selecting derivations were not described.

Baba et al. [6] presented work on retargetable microprogramming for mainframes. Due to the application area, much effort went into handling the complicated addressing mechanisms for micro-stores.

Vegdahl [43] extended the work of Cattell to microprogramming. In Vegdahl's approach, no automatic retargetability was reached. For example, routines for the generation of constants had to be written manually for each target machine.

Mueller et al. [35], also followed a partially manual approach. For each possible register transfer operation, corresponding paths in the target structure had to be selected manually. The control code enabling this path had to be specified as a PROLOG clause. Due to the involvement of manual actions, the compiler by Mueller (like that of Vegdahl) could only be used for infrequent remapping to new targets.

Recently, Mavaddat revived the grammar-based approach (see pp. **??**). He discovered more details about this technique, including a nice relation to Lindenmayer-systems.

4.1.3 Computer-aided design

Additional work was done in the area of electronic-CAD.

We are aware of two so-called *checkers of executability* [4, 42], which try to map a given behaviour onto a given structure. They do perform pattern matching. However, in the case of programmable processors, they do not generate binary instructions.

Some high-level synthesis systems are producing programmable processors. They also generate the required binary code. The Cathedral system [30] and the MSS system [29] fall into this category. However, they do assume that they are allowed to define or modify the final target structure.

4.1.4 Different levels of retargetability

An analysis of the literature on retargetable compilers reveals that the time required to retarget a compiler may be quite different:

- In certain cases it may require several months of work, including the rewriting of parts of the compiler amounts of software. Compilers of this class should be called *portable*.
- In other cases, new compilers are generated by *compilation* using a description of the target processor. The piece of software generating the compiler is called a *compiler-compiler*. In this case, retargeting basically consists of writing the target machine description.
- In the extreme case, retargeting a compiler does not even require a recompilation of the compiler itself. In this case, it does contain all the pattern matching routines. Such a compiler is *machine independent*[3]. Machine independent compilers allow frequent changes of the target (>1 per day).

4.2 Other requirements

Due to the focus of this book, work concerning requirements for code generation other than retargetability will be mentioned only briefly.

1. **Dependability**

   There is a huge amount of references on computing systems dependability. The interested reader is referred to [32]. It is expected that the demand for dependable computing will increase in the future and the importance of dependable computing cannot be over-emphasized. Formal methods for software development are one of the building blocks for dependable real-time computing (see e.g. [10]).

2. **Real-time response**

   Real-time programming is an area which has been well-studied over the years. There is a lot of work on real-time programming for process control. This work is essentially based on the assumption that a real-time operating system is available. Most of this work deals with rather soft time constraints.

   A related area is that of scheduling theory. Again, there is a large amount of work in the area. The relevance of this work in the context of this book is described in this book (see pp. **??**).

   Guaranteeing a certain real-time response is an issue that was initially neglected in high-level synthesis, but is now well-studied. This is especially true for DSP systems with given sample rates (see e.g. [**?**] for data-flow dominated applications). Without any particular reference to DSP systems, Gebotys, de Micheli and Landwehr have also addressed the issue (see [18, 25, 31]). Guaranteeing *hard* real-time constraints in processor-based systems was discussed only recently, e.g. in a paper by Boriello [7]. It is expected that much more work is required in this area.

3. **Extremely efficient code**

   Generating efficient code is an area, which is well studied in compiler construction. Most of the optimization techniques can be applied in the current context. However, the special requirements listed above have also to be taken into account. Fortunately, due to reduced compilation speed requirements, additional techniques become applicable. For example, integer programming can

[3]Note that the code still is machine dependent.

be applied for sections of the program (see pp. **??** of this book). For fast compilers, this would be impossible. Langevin takes advantage of the reduced compilation speed requirements in order to generate very efficient code (see pp. **??** of this book).

4. **Support for DSP algorithms**

   Currently available development platforms for DSP processors do already support DSP applications. For example, the development kit for the Motorola *MC 56k* [34] comes with assembly coded libraries for common DSP functions. See pp. **??** for a discussion of advantages and disadvantages of certain languages for describing algorithms.

5. **Support for DSP architectures**

   Available C compilers do generate code for DSP processors. However, they hardly exploit the available hardware features. Hence, it has been reported that the generated code is extremely poor.

   In order to cope with heterogenous register sets, *Trellis-diagrams* have been used (see pp. **??** of this book).

6. **Coupling between code selection, register allocations, scheduling, and compaction**

   Papers by Rimey and Hilfinger [39], by Hartmann [21] and by Bradlee [8] are three of the few papers describing the coupling two compiler phases. A tight integration of phases has been implemented by Schenk (see pp. **??** of this book).

5 Target models for retargetable compilation

In this section we compare different approaches for modelling target processors. See also Heinrich [22] for a discussion of target models for code generation.

5.1 Behavioural models

Behavioural models (instruction set models) of processors have been used in compiler construction for many years. They are the basis for many of the well-known pattern matching methods for code generation, such as the methods of Glanville [20] and Cattell [9]. Behavioural models provide a high abstraction of the underlying hardware. However, they do have problems with capturing the effects of pipelines, busy functional units and multiple assignments coded into one instruction word.

5.2 Structural models

Due to the problems with instruction set models, other models of target structures have been investigated.

In the context of our work on the MIMOLA hardware design system, we have used *structural target models* [37]. Structural models for our compilers are *complete* in the sense that they describe both the data-path and the controller. Amongst others, the advantage of such complete models is that they can be simulated with an RTL-structural simulator.

Structural models contain significantly more details than the instruction set model and the compilation speed that can be achieved with this model may be lower than that of traditional compilers. But structural models are well-established models in computer-aided design. So-called register-transfer *netlists* are available in most design environments and it would be great, if compilers could be generated from these. This would avoid any risks in the communication between hardware designers and compiler writers. Furthermore, structural models are able to describe all the features of modern processors.

5.3 Mixed models

Due to the problems with purely behavioural models and in an attempt to avoid detailed netlists, mixed models have been tried. For example, the target model of FlexWare [27] is such a mixed model. This model describes both the instructions as well as some of the hardware components. A mixed approach is also used with the language `nML` (see pp. **??**). For `nML`, the intention is to capture to instruction set from the programmer's manual and to include just enough structural information to make the code efficient.

The translation from behavioural models to structural models is possible with behavioural synthesis, but sometimes requires special techniques [28]. Mapping mixed models to a canonical form is possible with recent instruction set extraction techniques [26].

6 Summary

In this introduction, we have classified the different types of embedded processors that exist. We have sketched design procedures for systems containing such processors and we have listed the requirements for generating code for these. Furthermore, we have included an overview of related work, with emphasis on retargetability. Finally, we have distinguished different target machine models for use in retargetable code generation.

References

[1] A. Aho, M. Ganapathi, S. Tjiang. “Code Generation Using Tree Matching and Dynamic Programming”, ACM Trans. on Programming Languages and Systems, Vol. 11, 1989, pp. 491-516.

[2] A.V. Aho, S. C. Sethi. ”Optimal Code Generation for Expression Trees”, Journal of the ACM, Vol. 23, 1976, pp. 488-501.

[3] F. Anceau, P. Liddell, J. Mermet, C. Payan. “CASSANDRE: A Language to Describe Digital Systems”, 3rd Symp. on Computer and Information Sciences, Software Engineering, COINS III, 1969, pp. 179-204

[4] F. Anceau. “FORCE: A Formal Checker for Executability”, in: D. Borrione (ed.): From HDL Descriptions to Guaranteed Correct Circuit Designs, Proc. of IFIP WG 10.2 Working Conf., North Holland, 1986.

[5] A. Alomary, T. Nakata, Y. Honma, M. Imai, N. Hikichi “An ASIP instruction set optimization algorithm with functional module sharing constraint”, Int. Conf. on Computer-Aided Design (ICCAD), Santa Clara, November 1993, pp. 526-532.

[6] T. Baba, H. Hagiwara. “The MPG System: A Machine-Independent Efficient Microprogram Generator”, IEEE Trans. on Computers, Vol. C-30, June 1981, pp. 373-395.

[7] G. Boriello. “Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems”, Proceedings of the 31th Design Automation Conference, 1994, pp. 1-4.

[8] D. G. Bradlee, S. J. Eggers, R. R. Henry. “Integrating Register Allocation and Instruction Scheduling for RISCs”, Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1991, pp. 122–131

[9] R.G.G. Cattell. “Formalization and Automatic Derivation of Code Generators”, PhD thesis, Carnegie-Mellon University, Pittsburgh, 1978.

[10] J. P. Calvez. “Embedded Real-Time Systems”, Wiley Series in Software Engineering Practice, 1993

[11] M.E. Conway. “Proposal for an UNCOL”, Communications of the ACM, Vol. 1, 1958.

[12] H. Emmelmann. “Code Selection by Regular Controlled Term Rewriting”, in: G. Giegerich, S.L. Graham (ed.): “Code Generation - Concepts, Tools, Techniques”, Workshops in Computing, Springer 1992, pp. 3-29.

[13] J. Henkel, R. Ernst, U. Holtmann, T. Benner. “Adaption of Partitioning and High-Level-Synthesis in Hardware/Software Co-Synthesis”, ICCAD, 1994, pp. 96-100

[14] C.J. Evangelisti, G. Goertzel, H. Ofek. “Using the Dataflow Analyzer on LCD Descriptions of Machines to Generate Control”, Proc. 4th Int. Workshop on Hardware Description Languages, 1979, pp. 109-115.

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

[16] D. Gajski, F. Vahid, S. Narayan, J, Gong. “Specification and design of embedded systems”, Prentice Hall, 1994

[17] M. Ganapathi, C.N. Fisher, J.L. Hennessy. “Retargetable Compiler Code Generation”, ACM Computing Surveys, Vol. 14, (4) 1982.

[18] C. H. Gebotys, M. I. Elmasry. “Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis”, 28th Design Automation Conference, 1991, pp. 2-7.

[19] R. Giegerich, S. Graham. “Code Generation – Concepts, Tools, Techniques”, Dagstuhl-Seminar-Report, Technical Report 9121, 1991.

[20] R.S. Glanville “A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers”, PhD thesis, University of California at Berkeley, 1978.

[21] Hartmann. “Combined scheduling and data routing for programmable ASIC systems”, EDAC, 1992, pp. 486-490.

[22] W. Heinrich. “Formal Description of Parallel Computer Architectures as a Basis of Optimizing Code Generation”, PhD thesis, Technische Universität München, 1993.

[23] I.-J. Huang, A. Despain. “Generating Instruction Sets and Microarchitectures from Applications”, Int. Conf. on CAD (ICCAD), 1994, pp. 391-396

[24] K. Keutzer, W. Wolf. “Anatomy of a Hardware Compiler”, Proc. of the SIGPLAN ’88 Conf. on Programming Language Design And Implementation, 1988, pp. 95-104.

[25] D. Ku, G. De Micheli. "High Level Synthesis Under Timing and Synchronisation Constraints", Kluwer Academic Publishers, 1992.

[26] R. Leupers, P. Marwedel. "A BDD-based frontend for retargetable compilers", Proc. European Design & Test Conference, March 1995, pp. 239-243

[27] C. Liem, P. Paulin. "Instruction-Set Matching and Selection for DSP and ASIP Code Generation", Proc. European Design & Test Conference, March 1994, pp. 31-37.

[28] R. Leupers, W. Schenk, P. Marwedel. "Retargetable Assembly Code Generation By Bootstrapping", 7th Int. High Level Synthesis Symposium, May 1994, pp. 88-93.

[29] P. Marwedel, W. Schenk. "Cooperation of Synthesis, Retargetable Code Generation and Testgeneration in the MSS", EDAC-EUROASIC'93, 1993, pp. 63-69.

[30] H. De Man, J. Rabaey, P. Six. "CATHEDRAL II: A Synthesis and Module Generation System for Multiprocessor Systems on a Chip",in: G. DeMicheli, A. Sangiovanni-Vincentelli, P. Antognetti: Design Systems for VLSI Circuits–Logic Synthesis and Silicon Compilation–, Martinus Nijhoff Publishers, 1987.

[31] B. Landwehr, P. Marwedel, R. Dömer. "OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming, Proc. Euro-DAC, Sept. 1994, pp. 90-95

[32] J.C. Laprie. "Dependability: Basic Concepts and Terminology", Springer, Wien/New York, 1992

[33] E. Lee. "Programmable DSP architectures, Parts I and II", IEEE ASSP Magazine, Oct. 1988, pp. 4-19 and Jan. 1989, pp. 4-14

[34] Motorola Inc. "Instruction Set Details", included in "DSP Development Software" Kit, 1992

[35] R.A. Mueller, J. Varghese. "Flow Graph Machine Models in Microcode Synthesis", 17th Ann. Workshop on Microprogramming (MICRO-17), 1983, pp. 159-167.

[36] L. Nowak. "SAMP: A General Purpose Processor Based on a Self-Timed VLIW-Structure", ACM Computer Architecture News, Vol. 15, 1987, pp. 32-39.

[37] L. Nowak, P. Marwedel. "Verification of Hardware Descriptions by Retargetable Code Generation", 26th Design Automation Conference, Las Vegas, June 1989, pp. 441-447.

[38] J. V. Praet, G. Goossens, D. Lanneer, H. D. Man. "Instruction Set Definition and Instruction Selection for ASIPs", 7.th Int. Symposium on High-Level Synthesis, Canada, May 1994, pp. 11-16.

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

[40] U. Schmidt. "A new codegenerator-generator based on VDM (in German)", Computer Science Dpt., University of Kiel, Technical Report 4/83, 1983.

[41] R. M. Stallman. "Using and Porting GNU CC", Free Software Foundation, 1993.

[42] S. Takagi. "Rule Based Synthesis, Verification and Compensation of Data Paths", Proc. IEEE Conf.Comp.Design (ICCD'84), 1984, pp. 133-138.

[43] S.R. Vegdahl. “Local Code Generation and Compaction in Optimizing Microcode Compilers”, PhD thesis and report CMUCS-82-153, Carnegie-Mellon University, Pittsburgh, 1982.