

# Embedded Software in Real-Time Signal Processing Systems: Design Technologies

GERT GOOSSENS, MEMBER, IEEE, JOHAN VAN PRAET, MEMBER, IEEE,  
DIRK LANNEER, MEMBER, IEEE, WERNER GEURTS, MEMBER, IEEE, AUGUSLI KIFLI,  
CLIFFORD LIEM, AND PIERRE G. PAULIN, MEMBER, IEEE

## Invited Paper

*The increasing use of embedded software, often implemented on a core processor in a single-chip system, is a clear trend in the telecommunications, multimedia, and consumer electronics industries. A companion paper in this issue [1] presents a survey of application and architecture trends for embedded systems in these growth markets.*

*However, the lack of suitable design technology remains a significant obstacle in the development of such systems. One of the key requirements is more efficient software compilation technology. Especially in the case of fixed-point digital signal processor (DSP) cores, it is often cited that commercially available compilers are unable to take full advantage of the architectural features of the processor. Moreover, due to the shorter lifetimes and the architectural specialization of many processor cores, processor designers are often compelled to neglect the issue of compiler support.*

*This situation has resulted in an increased research activity in the area of design tool support for embedded processors. This paper discusses design technology issues for embedded systems using processor cores, with a focus on software compilation tools. Architectural characteristics of contemporary processor cores are reviewed and tool requirements are formulated. This is followed by a comprehensive survey of both existing and new software compilation techniques that are considered important in the context of embedded processors.*

## I. INTRODUCTION

Software is playing an increasingly important role in the design of embedded systems. This is especially true for personal telecommunications and multimedia systems, which form extremely competitive segments of the embedded systems market. In many cases the software runs on a

processor core, integrated in a very large scale integrated (VLSI) chip. Recent studies indicate that up to 60% of the development time of an embedded system is spent in software coding [1]–[3]. While this figure is a confirmation of an ongoing paradigm shift from *hardware* to *software*, at the same time it is an indication that the software design phase is becoming a bottleneck in the system design process.

### A. A Paradigm Shift from Hardware to Software

By increasing the amount of software in an embedded system, several important advantages can be obtained. First, it becomes possible to include late specification changes in the design cycle. Second, it becomes easier to differentiate an existing design, by adding new features to it. Finally, the use of software facilitates the reuse of previously designed functions, independently from the selected implementation platform. The latter requires that functions are described at a processor-independent abstraction level (e.g., *C* code).

There are different types of core processors used in embedded systems.

- *General-purpose processors.* Several vendors of off-the-shelf programmable processors are now offering existing processors as core components, available as a library element in their silicon foundry [4]. Both microcontroller cores and digital signal processor (DSP) cores are available. From a system designer's point of view, general-purpose processor cores offer a quick and reliable route to embedded software, that is especially amenable to low/medium production volumes.
- *Application-specific instruction-set processors.* For high-volume consumer products, many system companies prefer to design an in-house application-specific instruction-set processor (ASIP) [1], [3]. By customizing the core's architecture and instruction set, the system's cost and power dissipation can be reduced significantly. The latter is crucial for portable and network-powered equipment. Furthermore, in-house

Manuscript received February 1, 1996; revised December 2, 1996.

G. Goossens, J. Van Praet, D. Lanneer, and W. Geurts are with the Target Compiler Technologies and IMEC, B-3001 Leuven, Belgium (e-mail: goossens@imec.be; vanpraet@imec.be; lanneer@imec.be; guerts@imec.be).

A. Kifli is with IMEC, B-3001 Leuven, Belgium (e-mail: kifli@imec.be).

C. Liem is with TIMA Laboratories, INPG and SGS-Thomson Microelectronics, F-38031 Grenoble, France (e-mail: liem@verdon.imag.fr).

P. G. Paulin is with SGS-Thomson Microelectronics, F-38921 Crolles Cedex, France (e-mail: pierre.paulin@st.com).

Publisher Item Identifier S 0018-9219(97)02051-3.

processors eliminate the dependency from external processor vendors.

- *Parameterizable processors.* An intermediary between the previous two solutions is provided by both traditional and new “fabless” processor vendors [5]–[7] as well as by semiconductor departments within bigger system companies [8], [9]. These groups are offering processor cores with a given basic architecture, but that are available in several versions, e.g., with different register file sizes or bus widths, or with optional functional units. Designers can select the instance that best matches their application.

### B. Software, a Bottleneck in System Design?

The increasing use of software in embedded systems results in an increased flexibility from a system designer’s point of view. However, the different types of processor cores introduced above typically suffer from a lack of supporting tools, such as efficient software compilers or instruction-set simulators.

Most *general-purpose microcontroller* and *DSP cores* are supported with a compiler and a simulator, available via the processor vendor. However, in the case of fixed-point DSP processors, it is well known that the code quality produced by these compilers is often insufficient [1], [10]. In most cases these tools are based on standard software compiler techniques developed in the 1970’s and 1980’s, which are not well-suited for the peculiar architecture of DSP processors. In the case of ASIP’s, compiler support is normally nonexistent. Both for parameterizable processors and ASIP’s, the major problem in developing a compiler is that the target architecture is not fixed beforehand.

As a result, current day’s design teams using general-purpose DSP or ASIP cores are forced to spend a large amount of time in handwriting of machine code (usually assembly code). This situation has some obvious economical drawbacks. Programming DSP’s and ASIP’s at such a low level of abstraction leads to a *low designer’s productivity*. Moreover, it results in massive amounts of *legacy code* that cannot easily be transferred to new processors. This situation is clearly undesirable, in an era where the lifetime of a processor is becoming increasingly short and architectural innovation has become key to successful products. All the above factors act as a brake on the expected productivity gain of embedded software.

Fortunately, the research community is responding to this situation with a renewed interest in software compilation, focusing on embedded processors [11]. Two main aspects deserve special attention in these developments:

- *Architectural retargetability.* Compilation tools must be easily adaptable to different processor architectures. This is essential to cope with the large degree of architectural variation, seen in DSP’s and ASIP’s. Moreover, market pressure results in increasingly shorter lifetimes of processor architectures. For example, an ASIP will typically serve for one or two product generations only. In this context, retargetable compilation is the only solution to provide system designers with supporting tools.

- *Code quality.* The instruction and cycle count of the compiled machine code must be comparable to solutions designed manually by experienced assembly programmers. In other words, the compiled solution should exploit all the architectural features of the DSP or ASIP architecture. A low *cycle count* (or high execution speed) may be essential to cope with the real-time constraints imposed on embedded systems. A low *instruction count* (or high machine code density) is especially required when the machine code program is stored on the chip, in which case it contributes to a low silicon area and power dissipation. Note that although cycle count and instruction count are different parameters, compilers usually try to optimize both at the same time.

This paper is organized as follows. First an architectural classification of embedded processor cores is presented, in Section II. Section III introduces the problem of software compilation in an embedded context, and summarizes the main issues. Section IV then focuses on techniques for software compilation. Several traditional approaches are discussed, as well as newer research work in an embedded processor context. Section V formulates conclusions and a future outlook.

## II. A COMPILATION VIEW OF PROCESSOR ARCHITECTURES

The availability of efficient supporting tools is becoming a prerequisite for the fast and correct design of embedded systems. A major requirement is the availability of *software compilation* tools. In Section IV, different techniques for software compilation in the context of embedded processors will be discussed. One of the issues that will be emphasized is architectural *retargetability*, i.e., the ability to quickly adapt the compiler to new processor architectures.

A *retargetable* compiler is normally based on an *architectural model*. The compiler can generate code (of sufficient code quality) for the class of processor architectures that fit its model. Both for users and for developers of software compilers, it is useful to indicate the class of architectures that can be addressed with a given method. In this section we will therefore introduce a *classification scheme* for programmable processors [12]. An overview of programmable DSP architectures has been presented in [13]. Compared to that paper our classification scheme is more specific, in that it emphasizes those aspects that are relevant for a software compiler. It can be used to:

- *characterize a given compiler* (or compiler method), in terms of the classes of architectures that it can handle successfully;
- *characterize a given processor*, so that one can quickly find out whether suitable compiler support can be found.

We classify a processor architecture based on the following parameters: arithmetic specialization, data type, code type, instruction format, memory structure, register structure, and control-flow capabilities. These parameters will be explained in the sequel, and typical parameter values will be given for existing embedded processors in telecom and consumer applications.

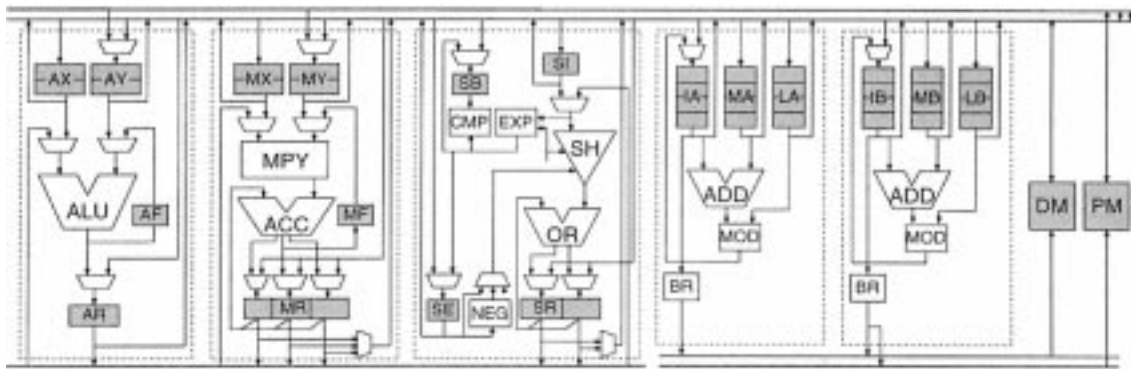


Fig. 1. Structure of the ADSP-21xx processor.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11: ALU/MAC oper'n with DM/PM load		PM load destin. = 00:AY0 01:AY1 10:MY0 11:MY1		DM load destin. = 00:AX0 01:AX1 10:MX0 11:MX1		Arithmetic oper'n = 0: 0000:nop MAC oper'n Opcode = ... 0100:X*Y <sub>ss</sub> 0101:X*Y <sub>su</sub> 0110:X*Y <sub>us</sub> 0111:X*Y <sub>uu</sub> 1000:MR+X*Y <sub>ss</sub> 1001:MR+X*Y <sub>su</sub> ... 1111:MR-X*Y <sub>uu</sub> Y = 00:MY0 01:MY1 10:MF 11:zero X = 000:MX0 001:MX1 010:AR 011:MR0 100:MR1 101:MR2 110:SR0 111:SR1						PM-AGU oper'n = Idx = 00:IB0 01:IB1 10:IB2 11:IB3 Mdf.Len = 00:MB0,LB0 01:MB1,LB1 10:MB2,LB2 11:MB3,LB3						DM-AGU oper'n = Idx = 00:IA0 01:IA1 10:IA2 11:IA3 Mdf.Len = 00:MA0,LA0 01:MA1,LA1 10:MA2,LA2 11:MA3,LA3					
1: ALU oper'n						Opcode = 0000:Y 0001:Y+1 0010:X+Y+C 0011:X+Y 0100:not X ... 1111:abs X Y = 00:AY0 01:AY1 10:AF 11:zero X = 000:AX0 001:AX1 010:AR 011:MR0 100:MR1 ... 111:SR1																	

Fig. 2. Part of the instruction set of the ADSP-21xx processor. Columns show different instruction fields, encoded by the instruction bits listed at the top.

Throughout this section we will refer to an existing DSP processor by means of example: the ADSP-21xx fixed-point DSP of Analog Devices [14]. This processor is chosen because it has many features that are also encountered in ASIP's. The ADSP-21xx architecture is shown in Fig. 1. The instruction-set of this processor supports about 30 different formats, of which the most parallel one is depicted in Fig. 2: an arithmetic operation on the ALU or multiplier, together with two parallel memory loads and two address calculations.

### A. Definitions

1) *Arithmetic Specialization*: Compared to other micro-processor architectures, a distinguishing feature of a DSP is the use by the latter of a parallel multiplier/accumulator unit. By virtue of this *arithmetic specialization*, the execution of correlation-like algorithms (digital filters, auto, and cross correlation, etc.) can be speeded up significantly.

In ASIP's, the idea of *arithmetic specialization* is even carried further. More specialized arithmetic units are introduced, controlled from the processor's instruction set, in such a way that the critical sections of the target algorithms (e.g., deeply nested loop bodies) can be executed in a

minimal number of machine cycles and without excessive storage of intermediate values. A typical example is the hardware support for a butterfly function in Viterbi decoding, encountered in ASIP's for wireless telecom [7], [15].

2) *Data Type*: Embedded processor cores for consumer and telecom applications normally support *fixed-point arithmetic* only. The reason is that floating-point units (as occurring, e.g., in many general-purpose microprocessors) require additional silicon area and dissipate more power. Floating-point arithmetic can however be avoided relatively easily in the VLSI implementation of consumer and telecom systems, without sacrificing numerical accuracy, by including the appropriate scaling operations in software or in hardware.

In a general-purpose DSP, different fixed-point data types are typically encountered. A distinct case is the ADSP-21xx architecture (Fig. 1), of which the most important data types are: a 16-bit type for ALU and multiplier operands, a 32-bit type for multiplier or shifter results, a 40-bit accumulator type, an 8-bit type for shift factors, and a 14-bit address type. Conversions between data types may be provided in the processor hardware. Consider the ADSP's accumulator register MR, which is 40-bits wide. In this case one 8-bit

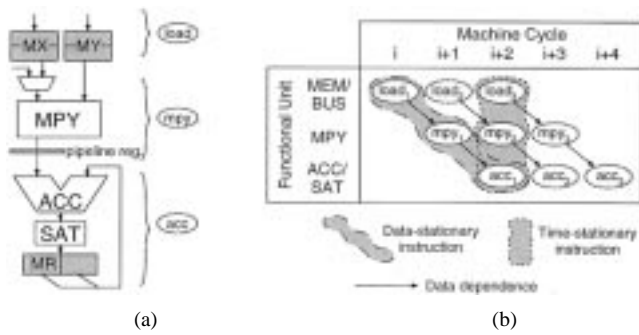


Fig. 3. Different code types, illustrated for a multiply-accumulate instruction (b), on a pipelined datapath (a).

and two 16-bit subwords of MR (called MR2, MR1, and MR0, respectively) are separately addressable, as the source operand of different arithmetic operations.

A comparable variety of data types can typically be found in ASIP's, where the bit-widths of functional units, busses and memories are chosen in function of the application. A good example is the ASIP for a private local telephone switch developed by Northern Telecom [1].

3) *Code Type*: Processors with instruction-level parallelism are often able to execute sequences of operations in a *data pipeline*. Fig. 3(a) shows an example of a multiplier-accumulator that can implement a three-stage data pipeline. In parallel with the *current* multiplication (“mpy”), this architecture can execute the accumulation with the *previous* multiplication result (“acc”) and the load of the *next* multiplication operand from memory (“load”).<sup>1</sup>

To control the operations in the data pipeline, two different mechanisms are commonly used in computer architecture: *data-stationary* and *time-stationary* coding [16].

- In the case of *data-stationary* coding, every instruction that is part of the processor's instruction-set controls a complete sequence of operations that have to be executed *on a specific data item*, as it traverses the data pipeline. Once the instruction has been fetched from program memory and decoded, the processor controller hardware will make sure that the composing operations are executed in the correct machine cycle.
- In the case of *time-stationary* coding, every instruction that is part of the processor's instruction-set controls a complete set of operations that have to be executed *in a single machine cycle*. These operations may be processing several different data items traversing the data pipeline. In this case it is the responsibility of the programmer or compiler to set up and maintain the data pipeline. The resulting pipeline schedule is fully visible in the machine code program.

Both code types are illustrated in Fig. 3(b). According to the authors' observations, time-stationary coding is used more often in *ASIP cores*, whereas *general-purpose processors* can use either type.

<sup>1</sup>Since the multiplication result is kept in the accumulator register MR, there is no need for a fourth stage to store the result in memory. Furthermore, we did not consider any address calculations for the operand loads; the latter could be put in an additional pipeline stage as well.

In addition to the operations in the data pipeline described above, a processor also has to fetch instructions from the program memory and decode them in an instruction decoder. This is done in one or more *instruction pipeline* stages, preceding the *data pipeline* stages. In processors with a *data-stationary* code type, instruction fetch and instruction decode are normally done in two separate instruction pipeline stages. For example, the data-stationary processor of Fig. 3(b) would typically have an overall pipeline depth of five cycles (fetch, decode, load, multiply, and accumulate). In processors with a *time-stationary* code type, instruction fetch and instruction decode are usually done in either a single or in two separate instruction pipeline stages, preceding the single execution cycle.

Processors with a time-stationary code type and a single fetch/decode cycle are often called *microcoded* processors. They have been studied intensively since the 1960's by the “microprogramming community.” In contrast, processors with multiple instruction pipeline stages, whether of time or data-stationary code type, are referred to as *macrocoded* processors [17].<sup>2</sup>

Macrocoded processors may exhibit pipeline hazards [17]. Depending on the processor, pipeline hazards may have to be resolved in the machine code program (statically) or by means of interlocking in the processor controller (dynamically). Macrocoded processors with interlocking are relatively easy to program, although it may be more difficult for a designer to predict their exact cycle time behavior.

4) *Instruction Format*: A distinction is made between orthogonal and encoded instruction formats.

- An *orthogonal format* consists of fixed control fields that can be set independently from each other. For example, very long instruction word (VLIW) processors [18] have an orthogonal instruction format. Note that the instruction bits within every control field may additionally have been encoded to reduce the field's width.
- In the case of an *encoded format*, the interpretation of the instruction bits as control fields may be different from instruction to instruction. The correct interpretation can be deduced from the value of designated bits in the instruction word (e.g., special format bits, like instruction bits 23 and 22 in Fig. 2, or specific opcode bits).

The processor's instruction decoder will translate instruction bits into control signals steering the different units in the processor.

When the processor is used as an *embedded core*, the application program will most often reside on-chip. In this case, processor designers aim at restricting the instruction word's width, in order to reduce the chip area and especially the power dissipation relating to program memory accesses. Should the chip be *field programmable*, it is convenient to choose an instruction width equal to the width of the chip's parallel data port (so that the instructions can be loaded via

<sup>2</sup>Note that the term *microcode* was originally introduced to refer to a lower level of control inside a processor controller, to decode and execute macrocoded instructions.

this port) and/or equal to the width of standard memory components (used for program memory).

For these reasons, many general-purpose DSP's have a 16-, 24-, or 32-bit wide instruction format. In contrast, many ASIP's have more uncommon instruction widths. In both cases, the instruction format is typically *encoded*.

Encoding in general restricts the *instruction-level parallelism* offered by the processor. A challenging task in the design of an ASIP is to determine an instruction set that can be encoded using a restricted number of instruction bits, while still offering a sufficient degree of parallelism for critical functions in the target application. Speed requirements for typical *telecom* and *consumer* applications make it possible to design efficient ASIP's that have a relatively high degree of instruction encoding. In contrast, *image processing* and *multimedia* applications may require a higher amount of instruction-level parallelism to meet their speed requirements. Most current ASIP's for these application domains therefore have orthogonal instruction formats [19]–[21].

5) *Memory Structure*: Many DSP and ASIP cores have *efficient memory and register structures*, which ensure a high communication bandwidth between the different datapath units, and between datapath and memory. In this section we will discuss memory structures; register structures will be treated in the next section.

a) *Memory access*: Memory structures are often classified on the basis of accessibility of data and program memory:

- *Von Neumann architecture*. These processors have a single memory space that is used to store both *data* and *program*. This was always the case in older microprocessor architectures of the CISC type.
- *Harvard architecture*. This term refers to the case where data and program are *accessible through separate hardware*. When applied to general-purpose RISC processors, this means that the data and program busses are separated. When applied to DSP processors, it means that the data and program memory spaces are separated. In many cases even *two* data memory spaces are provided, each with their own address generator. This is the case for the ADSP-21xx of Fig. 1.

In the remainder of this paper we will always assume that the processor has a Harvard architecture, with separate data and program memory spaces. This is the case for most current DSP's and ASIP's.

From a software compiler point of view, the choices of *addressing modes* and *operand location* are important issues. These will be discussed next.

b) *Addressing modes*: Processors usually support *multiple addressing modes* for data memories, such as *immediate*, *direct*, and *indirect* addressing. In the case of DSP's and ASIP's, indirect addressing is typically implemented on one or more address generation units. Often these units support specialized address operations, such as modulo counting to implement circular buffers for filter applications, counting with reversed carry propagation for FFT applications, and address post-modify instructions which allow to compute the "next" memory address simultane-

ously with the current memory access. It is essential that these features are supported by the compiler.

c) *Operand location*: With respect to *operand location*, the following classification of memory structures is most relevant [17]:

- *Load-store architecture* (also called *register-register architecture*). In a load-store architecture, all arithmetic operations get their operands from, and produce results in *addressable registers*. Communication between memories and registers requires separate "load" and "store" operations, which may be scheduled in parallel with arithmetic operations if permitted by the instruction set. The load-store concept is one of the basic ideas behind RISC architectures.

An example is the ADSP-21xx processor, of which one instruction format is shown in Fig. 2. As can be seen, all arithmetic operations belonging to this format operate on registers (addressed by instruction bits 12–8). Multiplication results are always written to register MR, while ALU results are written to AR. In parallel with the arithmetic operation, two load operations are executed to prepare the arithmetic operands for the next instruction cycle (in the registers addressed by instruction bits 21–18).

- *Memory-memory and memory-register architecture*. In this case, arithmetic instructions can be specified with *data memory* locations as operands. An example is the TMS320C5x DSP processor, which can execute a multiplication on two operands, respectively residing in a memory and in a register (and eventually store the result in an accumulator register).

Processor cores encountered in embedded systems can be of any of the above types. Note that in the case of a core processor, data and program memories are often placed on-chip to reduce board cost, access time (allowing for single-cycle access) and power dissipation.

6) *Register Structure*: Any processor will contain a *register set* for temporary storage of intermediate data values. Before discussing register structures in more detail, the following terms are introduced.

- *Homogeneous register set*. This is a register set in which all registers are interchangeable. If an instruction reads an operand from or writes a result to the register set, the programmer (or compiler) is allowed to select *any* element of the set.
- *Heterogeneous register set*. This type of register set consists of special-purpose registers. In this case, a register can only serve as an operand or result register of *specific* instructions. Likewise, an instruction can only read its operands from or write its results to *specific* elements of the register set.

Consider again the example of the ADSP-21xx processor. For the arithmetic operations belonging to the format of Fig. 2, the left and right operands are *restricted* to the registers indicated in the fields of instruction bits 10 to 8, and 12 to 11, respectively. Results can *only* be stored in MR (for multiplications), and AR (for ALU operations).

**Table 1** Scope of Retargetability of the Chess Compiler Using the Classification Scheme

Parameter	Supported values
Data type	Fixed and floating point Standard and user-defined data types
Code type	Time-stationary Harvard, multiple data memories
Instruction format	Load Store Addressing modes with post-modification
Register structure	Heterogeneous and homogeneous
Control flow	Zero overhead loops Residual control ...

The homogeneous case is an extreme point in the solution space: practical register sets are always more or less heterogeneous. In other words, the processor can be positioned anywhere on the axis from homogeneous to heterogeneous.

The register set of a processor can be partitioned into different *register classes*. A register class is a *subset* of the processor's register set, that can be viewed as homogeneous from the point of view of a certain instruction's operand or result. For example, {MY0, MY1, MF} constitutes a register class in the ADSP-21xx processor, since all elements of this set can serve as the right operand register of a multiplication in the format of Fig. 2. Note that register classes can be contained in each other or overlap with each other. The total number of register classes in a processor can now be considered as a measure for its heterogeneity.

The following is a rough classification of existing processor types.

- General-purpose *microprocessors* usually have a relatively *homogeneous* register set. In the case of fixed-point processors, the register set is normally divided in two register classes: the data-register class and the address-register class. In the case of floating-point architectures, the floating-point registers constitute a third class.
- General-purpose *DSP's* typically have a parallel multiplier. Compared to their microprocessor counterparts, this introduces at least one additional register class to store multiplication results. When the instruction format is encoded, some restrictions may exist on the choice of source and destination registers, which results in additional register classes.
- *ASIP's* typically have a strongly *heterogeneous* register set. The reasons are twofold. First, ASIP's may support many different data types, which often result in different register classes. Secondly, ASIP designers aim at a high degree of instruction encoding without significantly compromising the available instruction-level parallelism for the target application. This can be done by reducing the number of instruction bits for register addressing, in favor of the instruction bits for arithmetic or memory access operations. In this way a larger number of arithmetic and/or memory access operations can be executed in parallel, but the register structure becomes heterogeneous.

From the above discussion, it becomes clear that the optimization of the register structure is an important

task in the design of an ASIP architecture. However, the design of machine code that exploits such a heterogeneous register structure in an efficient way is nontrivial as well. As a matter of fact, the inefficient use of heterogeneous register structures is one of the prime reasons for the reported low code quality in the case of commercially available compilers for fixed-point DSP's (see Section I).

7) *Control Flow*: Many DSP's and ASIP's support standard control-flow instructions, like conditional branching based on bit values in the condition code register. However, several additional measures are usually taken to guarantee good performance in the presence of control flow. The following examples are typical.

- First, branch penalties are usually small, i.e., zero or one cycles. The branch penalty is the delay incurred in executing a branch due to the instruction pipeline.
- Furthermore, many DSP's and ASIP's have *zero-overhead loop* instructions. This allows to execute the body of a repetitive algorithm without spending separate cycles for loop control. This feature is essential for many time-critical applications.
- Several arithmetic or move instructions are *conditionally executable*. In many specific cases, this avoids the overhead of conditionally loading the program counter.
- Some arithmetic operations can be *residually controlled*. In this case the behavior of the operation depends on specific bit-values in a residual control register, which can be written by other operations. Typical examples are saturation modes for ALU or accumulate operations.
- The interrupt controller sometimes supports specialized context saving mechanisms like register shadowing, to minimize context switch times.

### B. Use of the Classification Scheme

The classification scheme introduced above can be used for different purposes. First of all, it can be used to characterize a given (retargetable) compiler, and indicate its "scope of retargetability." As an example, Table 1 indicates the scope of retargetability of the current version of the Chess compiler [22].

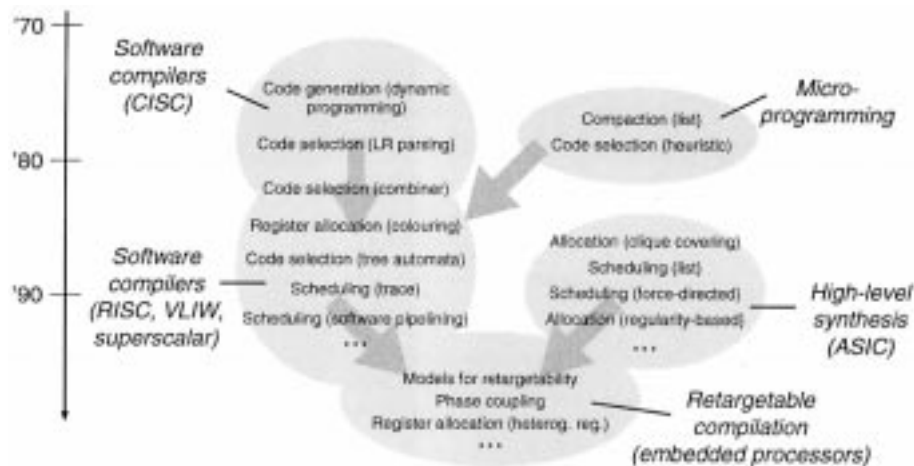
Second, the classification scheme can be used to characterize a given processor and quickly identify the issues related to compiler development. In this case, the model gives an indication of how easily a compiler can be built for the processor, and which existing compilers might be suited. For example, Table 2 shows the classification of a number of existing DSP and ASIP architectures.

## III. ISSUES IN SOFTWARE COMPILATION

Software compilation has been addressed since the 1950's. The aspect of architectural retargetability has been taken into consideration since the early 1970's. Due to the continuous evolution of processor architectures, software compilation has never lost its importance, both from a researcher's and from a practical user's point of view (see Fig. 4). The software compiler community has

**Table 2** Classification of Existing DSP-ASIP's Based on Six Parameters of Classification Scheme

Parameter	EPICS10 [8]	TMS320C54x [15]	LODE [7]	TCEC MPEG [19]
Arithmetic specialization	Plug-in applic.-spec. unit	Viterbi ALU	Dual multipl.-accumulator	
Data type	Fixed point	Fixed point	Fixed point	Fixed point
Code type	Time-stationarity	Data-stationarity	Data-stationarity	Time-stationarity
Instruction format	Encoded	Encoded	Encoded	Orthogonal
Memory structure	Harvard with two data memories	Harvard with two data memories	Harvard with two data memories	Harvard with four data memories
	Load-store	Memory-reg.	Memory-reg.	Load-store
	Address. modes with post-modification	Address. modes with post-modification	Address. modes with post-modification	Address. modes with post-modification
Register structure	Heterogenous	Heterogenous	Heterogenous	Heterogenous



**Fig. 4.** Evolution of retargetable compiler research in the past decades.

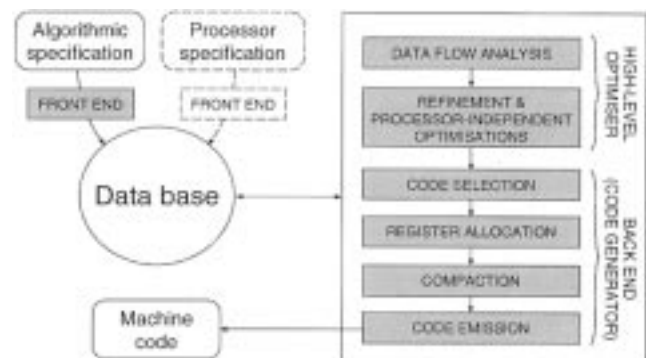
been focusing mostly on general-purpose microprocessors, which evolved from traditional CISC architectures, over RISC's, to more parallel VLIW and superscalar architectures. Until recently, DSP processors—and obviously ASIP's—received relatively little attention.

Most processor vendors offer *C* compilers with their processors. In several cases these compilers are ports of *GCC*, a compiler framework distributed by the Free Software Foundation [23]. *GCC* combines a number of techniques developed by the compiler community, primarily for general-purpose microprocessors. With the free distribution of its *C* source code, *GCC* has been ported to countless processors and has been retargeted to even more. Examples of existing DSP's for which commercial retargets of *GCC* are available include: Analog Devices 2101, AT&T 1610, Motorola 56001, and SGS-Thomson D950. It has become the *de facto*, pragmatic approach to develop compilers from a freely available environment. For processors close to the intent of *GCC* this can be fairly quick.

Nonetheless, as mentioned in Section I and in the companion paper [1], the code generated by the available compilers for fixed-point DSP's is too often of unacceptable quality for industrial use, so that design teams have to resort to manual assembly coding.

Fortunately, the emerging market of embedded processors has initiated a revival of software compilation research for DSP's and ASIP's, since the early 1990's [11] (Fig. 4).

In Section IV a survey will be presented of some traditional software compilation techniques that are relevant



**Fig. 5.** Anatomy of a software compiler.

in the context of embedded processors. In addition, an outline will be presented of recent developments in software compilation for embedded processor architectures.

Fig. 5 shows the typical anatomy of a software compiler. The starting point of the compilation process is an application program in an *algorithmic specification language*. Most compilers for embedded processors use *C* as the algorithmic specification language. A drawback of standard *C* is its restricted support for different data types. DSP's and ASIP's often accommodate a wide variety of (fixed-point) data types. For these cases, the *C* language is sometimes augmented to support user-definable data types [24].

The algorithmic specification is translated into an *intermediate representation*, by means of a language-dependent front-end. The intermediate representation, which is kept in the compiler's data base, is accessible by the subsequent

compilation phases. Well-known intermediate representations for representing the algorithm include the *static single assignment form* (SSA form) [25], and the *control/dataflow graph* (CDFG) [26], [27].

In addition to the algorithmic specification, a *retargetable* compiler will also use a processor specification, that must be available in an internal model in the compiler's data base. This model may be generated automatically, starting from a *processor specification language*. Examples of specification languages and internal compiler models for representing processors will be discussed in Section IV-A–B. A *compiler generator*<sup>3</sup> is a tool that automatically builds a processor-specific compiler, with its internal model, from a description in a processor specification language.

The software compilation process is traditionally divided into high-level optimization and back-end compilation. In the *high-level optimizer*, a data-flow analysis [28] is carried out to determine all required data dependencies in the algorithm, needed to build the SSA form or CDFG. Processor-independent optimizations are carried out, to reduce the number of operations or the sequentiality of the description. The set of optimizations is quite standard, and includes common subexpression elimination, dead code removal, constant propagation and folding, etc. [28]. The *back-end* performs the actual *code generation*, whereby the intermediate representation is mapped on the instruction set of the target processor. In this code generation process, different *phases* can be distinguished:

- *Code selection*: The operations in the algorithmic model are bound to the *partial instructions*, supported by the target processor's instruction set. Multiple operations can be combined in the same partial instruction. This is determined by *covering* the operations in the model with (complex) patterns, each representing a partial instruction.
- *Register allocation*: Intermediate computation values are bound to *registers* or *memories*. If necessary, additional data move operations are added.
- *Scheduling*: In this phase the code generator attempts to exploit the remaining *instruction-level parallelism* that is available in the processor architecture. Partial instructions that can execute in parallel are *grouped* into complete instructions, and assigned to *machine cycles*. Whereas the set of partial instructions after code selection and register allocation is usually called *vertical code*, the final instructions after scheduling are referred to as *horizontal code* [29]. The transformation from vertical to horizontal code is sometimes also called *code compaction*.

It is important to note that in many compilers, a partial ordering of operations is already determined during the earlier phases of code selection and register allocation. As a matter of fact, determining a vertical ordering of partial instructions is a critical issue in several code selection and register allocation algorithms, that affects the eventual code quality.

<sup>3</sup>Also termed a *compiler compiler* or (when restricted to the compiler's back end) a *code-generator generator*.

The above described code generation phases are encountered in software compilers for general-purpose microprocessors as well as in more recent compilers for embedded processors. In traditional compilers for *CISC* processors, code selection was the most important code generation phase. In this case, local register allocation was included in the code selection phase. In current compilers for *RISC* processors, code selection and (global) register allocation are typically done in separate phases.<sup>4</sup> Furthermore, instruction ordering has become an important issue in these compilers, due to the possible occurrence of pipeline hazards. With the advent of *VLIW* and *superscalar* processors, more emphasis is being put on efficient scheduling, to cope with the larger amount of instruction-level parallelism in these architectures [30].

In today's context of *embedded processors*, the following new aspects are added to the problem: *architectural retargetability* and the requirement of *high code quality* for irregular architectures (see Section I). The latter requirement has a dual impact on the compilation methodology:

- Because of instruction-level parallelism and the occurrence of heterogeneous register structures, the different compilation phases become strongly interdependent. In order to generate high quality code, each code generation phase should take the impact on other phases into account. This is called *phase coupling* [31].
- In order to generate high quality code, more specialized compiler algorithms may be required, that explicitly take into account aspects like heterogeneous register structures. Examples will be given in Sections IV-C–F. An important point is that larger compilation times can be tolerated in the case of embedded processors, compared to general-purpose microprocessors.

In order to tackle these new challenges, several compiler researchers are investigating synergies between *software compilation* techniques for *general-purpose* processors and techniques for *high-level synthesis* of *application-specific hardware* [11]. This approach is motivated by the fact that several high-level synthesis tools are targeting irregular architectures with instruction-level parallelism.

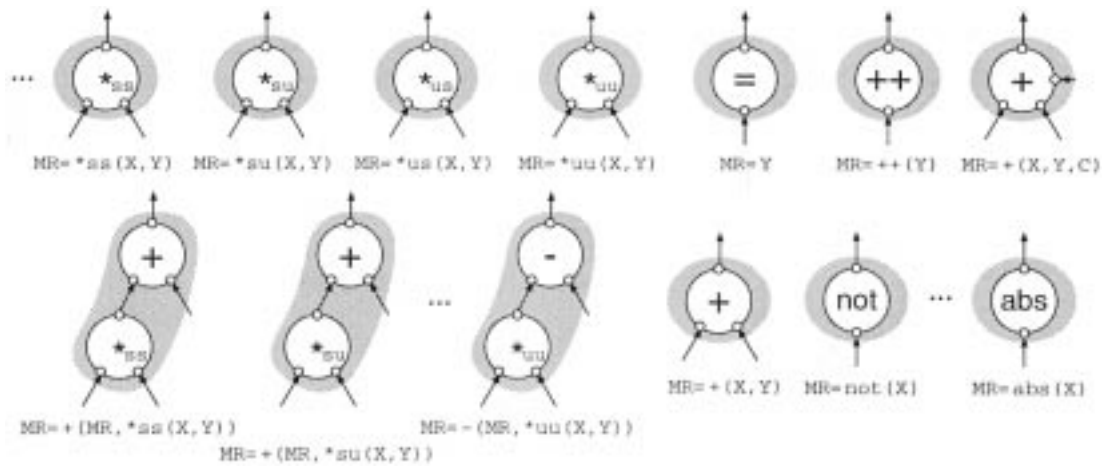
#### IV. A SURVEY OF COMPILATION TECHNIQUES

Following the discussion of processor architectures and of general compiler issues, next a survey is provided of existing techniques for processor modeling and software compilation.

##### A. Processor Specification Languages

The use of efficient and powerful models to represent all required characteristics of a processor is a key aspect in making the software compilation process retargetable. Although many compilers use *separate, specialized models* for each of the compilation phases, attempts have been made to use a *single processor model* for retargetable compilation, supported with a user-friendly *processor specification language*. In this section, processor specification

<sup>4</sup>The terms *local* and *global* register allocation will be defined more precisely in Section IV-C.



**Fig. 6.** Part of a tree pattern base, derived for the ADSP-21xx instruction format of Fig. 2. Below each tree the corresponding grammar representation is shown. In this example, source operand registers are not modeled in the pattern base.

languages are discussed. Processor models for compilers are treated separately in Section IV-B.

1) *Netlist-Based Languages:* A first type of processor specification languages describe the processor as a *netlist of hardware building blocks*, including datapath, memories, instruction decoder, and controller. This approach is followed in the MSSQ compiler, which accepts a processor specification in the *Mimola* language [32]. The advantage of these languages is their completeness. However, a netlist may not always be available to the compiler designer. Furthermore, this approach requires that the architectural design is completed, which precludes building compilers during the architecture exploration phase in ASIP design.

2) *High-Level Languages:* As an alternative to netlist-based formalisms, several high-level processor description languages have been proposed. The idea behind these languages is to capture the information that is available in a *programmer's manual* of a processor. Usually such a description contains a structural skeleton of the processor (essentially a declaration of storage elements and data types), and a description of the actual instruction set. A first example is the *ISP* language [33], with its descendant *ISPS* [34]. In *ISP* the instruction set is captured by specifying the behavior that corresponds to specific sets of instruction bits. For the latter, a procedural formalism is used. More recently, the *nML* language was proposed [35]. *nML* uses an attributed grammar. The grammar's production rules define the composition of the instruction set, in a compact hierarchical way. The semantics of the instructions (e.g., their register-transfer behavior and their assembly and binary encoding) are captured by attributes. *nML* is used by the CBC [36] and Chess [22] compilers.

## B. Processor Models for Compilation

1) *Template Pattern Bases:* A first approach, used by traditional compilers for general-purpose CISC and RISC processors [28], [37], is to represent the target processor by means of a *template pattern base*, that essentially enumerates the different partial instructions available in the instruction set. Each partial instruction is represented as a

pattern, expressed by means of the algorithm intermediate representation. Fig. 6 shows an example of a pattern base, where a graphical representation is used in the form of CDFG patterns. Often the patterns are expressed using a *grammar*.

As will be explained in Section IV-C, several code generators restrict the allowed template patterns to *tree structures*. This is the case in Fig. 6, where each pattern computes a result value from one or more operand values. The corresponding grammar model is a "regular tree grammar," in which each production rule describes a partial instruction as a pattern in (usually prefix) linearized form. Terminal grammar symbols correspond to operations executed by an instruction, while nonterminal symbols may correspond to possible storage locations. To reduce the number of grammar rules, common subpatterns can be factored out; the rule describing a subpattern is then connected to the remaining rules via additional nonterminals.

Examples of code-selector generators using regular tree grammars include *Twig* [38], *Burg* [39], *Iburg* [40], and the *Graham-Glanville* code generators<sup>5</sup> [41]. Several recent compilers for embedded processors have adopted *Iburg* for the code selection phase, such as *CBC* [36], *Record* [42], and the *Spam* project compiler [43]. In the *CBC* compiler the regular tree grammar, that serves as the input to *Iburg*, is derived automatically from an *nML* specification of the target processor [36]. Similarly, in *Record* this grammar is derived from a *Mimola* specification [44]. In *Spam* a regular tree grammar is specified by the user, in the format supported by the *Olive* code-selector generator which is similar to *Iburg*. Other compilers for embedded processors using a pattern base include *CodeSyn* [45].

Although a template pattern base in the first place describes the processor's *instruction set*, it is often also extended with additional *structural* information to reflect the processor's register structure. For example, additional patterns, called *chain rules*, may be inserted to describe

<sup>5</sup>As will be discussed in Section IV-C, Graham-Glanville code generators actually use *string* grammars, which have the same textual representation as regular tree grammars.

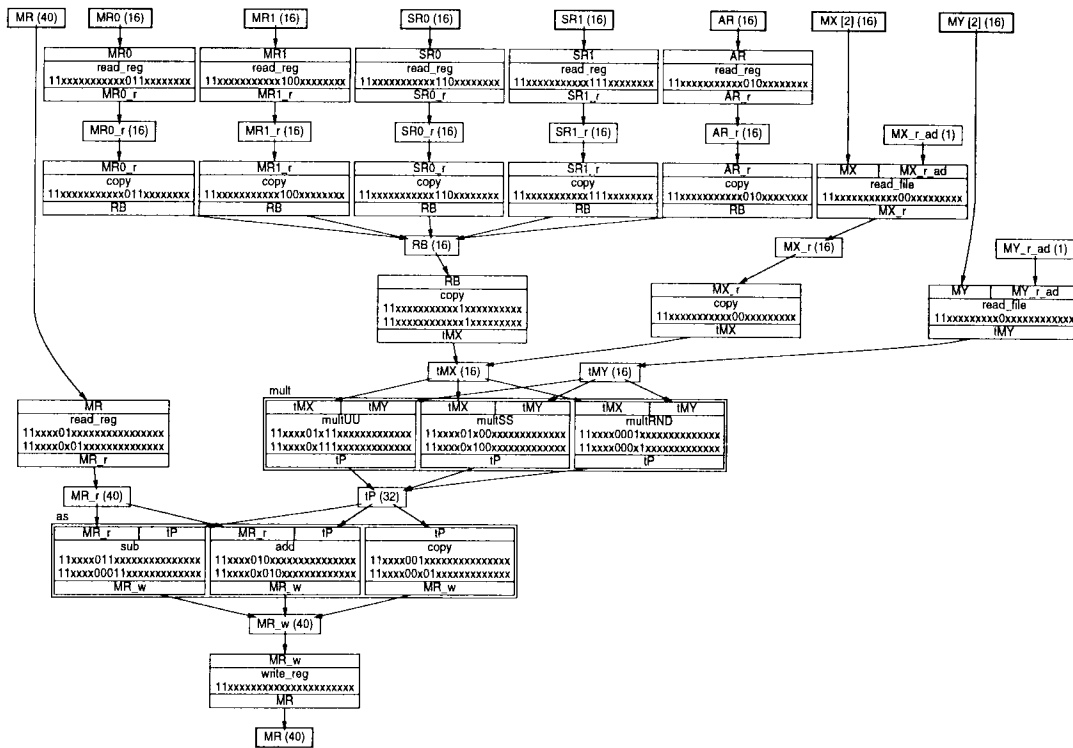


Fig. 7. Part of the ISG representation of the ADSP-21xx processor.

possible moves between storage elements. The grouping of registers into register classes can be modeled. In [46], a more specialized processor model is proposed, based on *Trellis diagrams*, which in essence combine tree patterns with structural information.

2) *Graph Models*: An alternative approach, mainly used in compilers for embedded processors, is to represent the processor by means of a *graph model*. Such a model has the advantage that it can more readily represent structural information, which makes it possible to describe several of the peculiarities of ASIP architectures.

The *MSSQ* compiler [47] uses a “connection-operation graph” that is derived from a detailed processor netlist. In the *RL compiler* [48] a “place-time graph” is used, i.e., a graph that captures all legal data moves in the processor. The *Chess* compiler is built around an “instruction-set graph” (ISG) [49], which is used by all code generation phases. Fig. 7 shows the ISG representation of a part of the ADSP-21xx processor, introduced in Fig. 1. The ISG captures both *behavioral* (instruction set) and *structural* (register structure, pipeline behavior, and structural hazards) information. The ISG is a bipartite graph, with vertices representing *structural elements* (small boxes in the figure representing registers, memories, or connections) and *microoperations* (large boxes), respectively. These objects are annotated with their enabling condition, to indicate the binary instruction format(s) to which they belong. Edges indicate the legal *dataflow* between the structural elements.

### C. Code Selection

The phase of code selection has received a lot of attention in the software compiler community. In early compilers

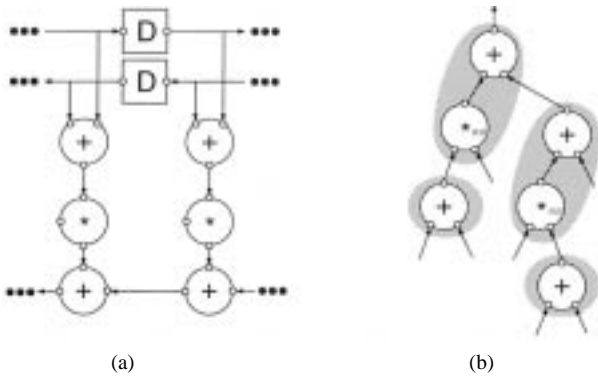
for CISC architectures, code selection was the main code generation phase. In these compilers, code selection also determines local register allocation as a by-product. This is possible because CISC’s always have a memory-register or a memory-memory structure, so that the number of available registers is restricted.

Later on, the same techniques have been applied to RISC compilers as well. In these compilers, register allocation is normally deferred to a separate code generation phase. However, the issue of phase coupling between code selection and register allocation has received renewed interest in recent compilers for DSP’s and ASIP’s, due to the occurrence of heterogeneous register structures.

It has been shown that code generation is an NP-complete problem, for intermediate representations that take the form of a directed acyclic graph (DAG) [50], [51]. However, *optimal vertical code* (i.e., without instruction-level parallelism) can be generated in polynomial time, when the following conditions are satisfied [37], [52]:

- 1) the intermediate representation is an *expression tree* (hereafter called the *subject tree*);
- 2) the template pattern base is restricted to contain only *tree patterns* (as in Fig. 6), i.e., it can be represented as a *regular tree grammar* (see Section IV-B);
- 3) the processor has a *homogeneous register structure*.

Different code generation algorithms have been developed to solve this canonical problem. However, since the above conditions are often not satisfied in practice, code generators often incorporate *extensions* of those basic algorithms. Some of these basic algorithms, as well as various practical extensions, are surveyed below. The discussion in this



**Fig. 8.** Code selection for a symmetrical filter, using the tree pattern base of Fig. 6: (a) CDFG of the application and (b) tree-structured intermediate representation with a possible cover.

section will be restricted to the *code selection* phase of code generation, including those approaches that incorporate *local register allocation* in the code selection process. In this context, “local” means that register allocation is done for values *within the subject tree* only. Techniques for global register allocation, as a *separate* code generation phase, are discussed in Section IV-D.

1) *Dynamic Programming*: Several code-selector generators are based on a stepwise partitioning of the code selection problem, using *dynamic programming*. It is assumed here that conditions 1) and 2) of the canonical problem above are satisfied. Two phases are distinguished in the code selection problem:

- *tree pattern matching*, i.e., locating parts of the subject tree that correspond to available tree patterns in the pattern base.
- *tree covering*, i.e., finding a complete cover of the subject tree with available patterns (see Fig. 8).

The use of dynamic programming was first proposed by Aho and Johnson [37], assuming a *homogeneous* register structure [see condition 3)]. *Tree pattern matching* is done in a straightforward way, in a bottom-up traversal of the subject tree (i.e., from the leaves to the root). For each tree node, the method computes the minimal *cost* to cover the subtrees rooted at that node. Cost calculations are done using the principle of dynamic programming. The cost takes into account both the register utilization of the subtrees and the different possible orderings of these subtrees. In this way, both local register allocation and scheduling (within the expression tree under consideration) are included in the algorithm. During a top-down traversal of the subject tree, the *tree cover* is finally found, by determining the minimal cost at the tree’s root node. In [37] it is proven that the program, resulting from the application of dynamic programming, is a *strong normal form* program. This is a program that consists of sequences of vertical code which are strongly contiguous, meaning that for each node it is guaranteed that one subtree of the node is completely executed before the next of its subtrees is executed. [37] also proves that these strong normal forms are optimal for the canonical problem introduced above.

Dynamic programming is also used in more recent code selection approaches for processors with a *heterogeneous*

*register structure*, such as the code selectors generated by *Twig* [38], *Beg* [53], and *Iburg* [40]. Again, dynamic programming is used for cost calculation. However, this time a *separate cost* is calculated *per register class*, at each of the nodes in the subject tree. To keep the problem tractable, a number of simplifications are made. First of all, it is assumed that every register class has an *infinite number of registers*. Secondly, the costs that are calculated do not reflect any local register allocation nor operation ordering. These issues are delayed to the subsequent code generation phases. Yet the technique will insert data *move* operations between different register classes of the processor, when appropriate.

The implementations of *Twig*, *Beg*, and *Iburg* use a *tree automaton* to traverse the subject tree in the tree pattern matching step. An improvement of tree automaton based code selection is provided by the “bottom-up rewrite system” (BURS) theory [54]. Whereas the original methods calculate the intermediate costs *during* the actual code selection, BURS theory allows to shift these calculations to the *generation phase* of the code selector. This results in much faster code selection. However, a drawback of BURS theory is that the cost function is restricted to a constant additive model. *Burg* [39], [55] is a code-selector generator based on BURS theory. A formal treatment of tree-automaton based code selection can be found in [56].

Several recent compilers for embedded processors are using adaptations of one of the dynamic programming methods described above. The following adaptations can be mentioned.

- In practice, intermediate representations are often *graphs* rather than trees. The traditional method to partition a DAG representation into different expression trees is to cut the DAG at each edge representing a value that is used multiple times. This is illustrated in Fig. 8 by the derivation of the tree (b) from the graph (a). Dynamic programming based techniques are then applied to the individual trees, and afterwards the results are combined by allocating registers for the values that are shared among trees. Based on this extension, dynamic programming has been adopted among others by the following compilers: *CodeSyn* (using straightforward pattern matching) [57], *CBC* [36], *Record* [44], and *Spam* [43] (all three using *Iburg* or variations of it).
- As mentioned in Section II, many DSP’s and ASIP’s have a *heterogeneous* register structure. While dynamic programming based code selection has been extended to heterogeneous structures (see above), these methods suffer from a rather weak phase coupling with *register allocation*. For example, they do not consider optimizations like spilling of data values to memory, or the constraint that the register capacity (i.e., the number of registers) in a register class is restricted. Both issues are then deferred to a separate register allocation step. An alternative approach is provided in [46], presenting a combined method for code selection and register allocation for heterogeneous structures. Spilling is considered, as are

register capacity constraints. The covering problem is reformulated as a path search problem in *Trellis trees*, which produces strong normal form programs. For heterogeneous register structures, these programs cannot be guaranteed to be optimal. In the *Spam* project a different approach is followed: in [43] a subclass of architectures with a heterogeneous register structure is identified for which *optimal* vertical code can be produced using a dynamic programming based code selector. These architectures do not necessitate spilling. In this approach strong normal form programs are produced. The method includes register allocation and ordering. In practice, this formulation is applicable to the TMS320C25 architecture.

2) *LR Parsing*: When the processor model is a *regular tree grammar* (see Section IV-B), code selection can be viewed as a problem of *parsing* the subject tree using the specified grammar. As a matter of fact, the tree-automaton based methods described above (see dynamic programming) are parsing methods for regular tree grammars. For most practical processors, the regular tree grammar is highly ambiguous: several derivations may be obtained for the same expression, which represent the optimization space for the code selector. The dynamic programming method allows to find an optimized solution within this space.

Parsing approaches to code selection were however known long before the introduction of tree-automaton based methods. Graham–Glanville code generators [41], [58], developed in the 1970’s, use the same type of grammar but interpret it as a *string* grammar. Subject trees are linearized expression trees, which are then parsed with a *left-right (LR) parsing technique*. Because of the linearization of patterns, Graham–Glanville code selectors perform tree pattern matching in a left-operand biased fashion, i.e., when generating code for a subtree, the code for the left operand of the root node is selected without considering the right operand. This may produce inferior results, compared to the dynamic programming based methods.

Other recent parsing approaches to code selection have been described in [59], [60].

3) *Graph Matching*: As explained above, dynamic programming-based approaches to code selection suffer from the restriction that the pattern base and the intermediate representation must consist of tree structures. Some authors therefore proposed pattern matching algorithms that directly support DAG structures. In [51] a code selection algorithm was presented that can generate optimal vertical code for DAG’s, on a processor with only a single register. In [61] this algorithm has been further refined to support commutative operations and multiregister architectures similar to the TMS320C25 processor.

4) *Bundling*: The code selection techniques described hitherto rely on the availability of a template pattern base, possibly in the form of a regular tree grammar, which essentially enumerates *all* legal partial instructions *in advance*.

An alternative approach to code selection is to use a *bundling* algorithm, in which *only the required patterns are constructed* on the fly during a traversal of the intermediate

representation. Whether or not such a pattern (also called a *bundle*) is legal, can be derived from the processor model, which in this case is given in the form of a graph model (see Section IV-B).

An early example of a bundling approach to code selection is the *combiner* algorithm [62], a variation of the peephole optimization technique presented in [63] and used in the *GCC* compiler. More recently, bundling algorithms have been developed for compilers for embedded processors, such as *MSSQ* [47], *MSSV* [64], and *Chess* [22].

An advantage of these bundling algorithms is that they support intermediate representations and partial instructions that are graphs rather than trees. These features are useful in a DSP and ASIP context. A disadvantage is the increased algorithmic complexity, compared to dynamic programming or parsing methods. Note that the bundling algorithms in *MSSQ* and *MSSV* include local register allocation. In *Chess*, register allocation is however deferred to a separate compiler phase. Phase coupling is supported though, primarily through the principle of late binding: when several legal bundles exist for the same group of operations in the intermediate representation, the choice will be deferred to the register allocation or even the scheduling phase.

The approach is illustrated in Fig. 9 for the example of Fig. 8, assuming that the processor is specified using the *ISG* graph model. First all possible mappings of individual CDFG operations to partial instructions are determined [Fig. 9(b)]. Next the CDFG is traversed to find combinations of operations that correspond to more complex partial instructions, while taking into account that their operands and results can be read from, resp. written to, available storage elements [Fig. 9(c)].

5) *Rule-Driven Code Selection*: Rule-driven approaches to code generation have been explored, e.g., in [65], and have more recently been used in the *FlexCC* compiler [66]. These approaches combine a progressive set of refinement phases to produce machine code. At *each phase* of compilation, a *set of rules* is provided in a well-structured programming environment which guides each transformation.

The critical phase of the process is code selection, where the compiler developer defines a *virtual machine* which resembles as closely as possible the instruction-set of the real machine, but is sequential in operation. This virtual machine does not support any instruction-level parallelism. The issue of parallelism is deferred to the code compaction phase. Using the definition of the available register sets and addressing modes of the architecture, the developer specifies a set of rules which map operation patterns onto instructions of the virtual machine (Fig. 10). Although there is no fundamental restriction, practical implementations assume *tree* structured patterns.

To produce the rule base, the developer has at his disposal a set of primitives to manipulate the standard set of tree patterns onto the virtual machine instructions, the available register sets, and the addressing modes.

Operands of operation trees are allocated to register sets based on matchings to the *C* data types (char, int, ptr, float, long, etc.) which are declared in the specification.

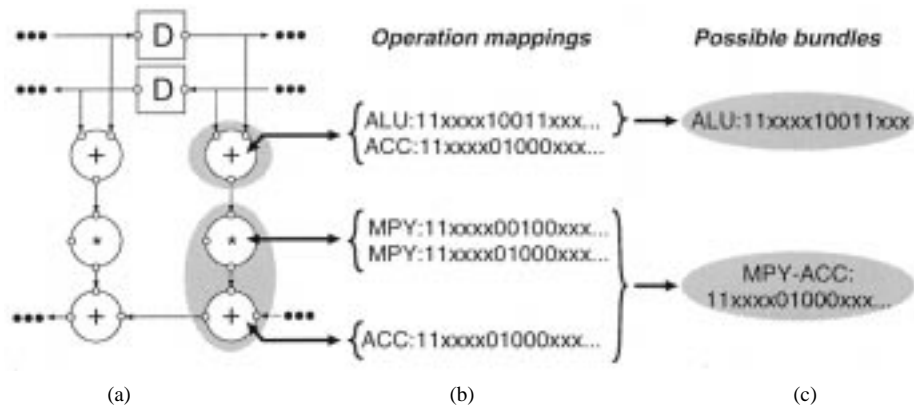


Fig. 9. Code selection using a bundling approach: (a) CDFG, to be mapped on the ISG of Fig. 7, (b) initial mappings of CDFG on ISG vertices, and (c) construction of bundles.

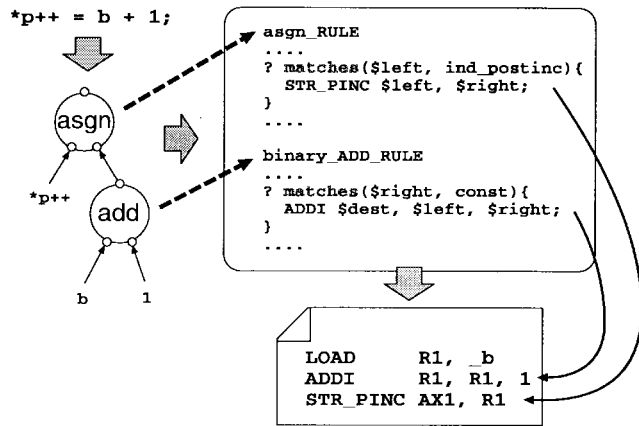


Fig. 10. Virtual code selection.

Optionally this allocation may be constrained to specific registers or register sets of the virtual machine instructions within the rules for selection. This flexibility is important for the support of the specialization of register functions in embedded processors. Register assignment within each register set is performed independently of the code selection process.

After mapping operation patterns onto the virtual machine, user-supplied transformation rules are used to optimize the description and generate instructions for the actual processor.

While rule-driven compilation provides a fast way to compiler generation, the quality of the compiler is directly dependent on the skills of the user to write adequate transformation rules. Furthermore, as illustrated in [66], to generate high quality code the user may have to rewrite the source code program to a level close to the target instruction set. For example, pointer referencing may have to be used for array variables, and the register allocation may have to be predetermined partly by the user.

#### D. Register Allocation

In the register allocation phase, the compiler assigns intermediate computation values to storage locations in the processor. In Section IV-C several techniques have already been discussed that essentially perform *code selection*, but

are able to carry out local register allocation decisions on the fly. In this section, techniques will be reviewed that essentially perform *global register allocation*, as a separate code generation phase. However, it will be shown that several of these techniques are also able to perform remaining code selection decisions on the fly. This illustrates that the exact partitioning of the code generation process in different phases is nontrivial, and has to be decided by compiler developers based on the architectural context.

1) *Graph Coloring*: A standard formulation of the register allocation problem, on which several practical implementations are based, is in terms of *graph coloring* on an interference graph [67]. To explain the basic graph coloring formalism, we make the following initial assumptions:

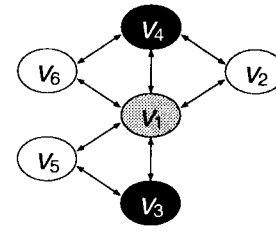
- 1) the processor has a *homogeneous* register structure;
- 2) the register set's capacity (i.e., number of registers) is *restricted* to a predefined value, say  $N$ ;
- 3) *code selection* has been accomplished in a preceding phase;
- 4) an *execution ordering* of the different instructions has been determined (e.g., in the code selection phase).

The execution order determines a *live range* for every intermediate computation value. Based on these live ranges, an interference graph is constructed. This is an undirected graph of which the vertices correspond to live ranges of values, and edges connect the vertices of interfering (i.e., overlapping) live ranges. Register allocation then is equivalent to finding an acceptable vertex coloring of the interference graph, using at most  $N$  colors. Heuristic graph coloring algorithms are used.

Fig. 11 shows the interference graph constructed for a set of values with given live ranges. In this example the vertices of the interference graph can be colored using at most three colors, each resulting in a different register. If the interference graph cannot be colored with  $N$  colors, the register capacity is exceeded. In this case, a standard solution is to temporarily *spill values to memory*. Alternatively, values that serve as an operand of multiple operations, can be recomputed prior to every use. This transformation is called *rematerialization*. Chaitin proposed a number of heuristics for spilling and rematerialization,

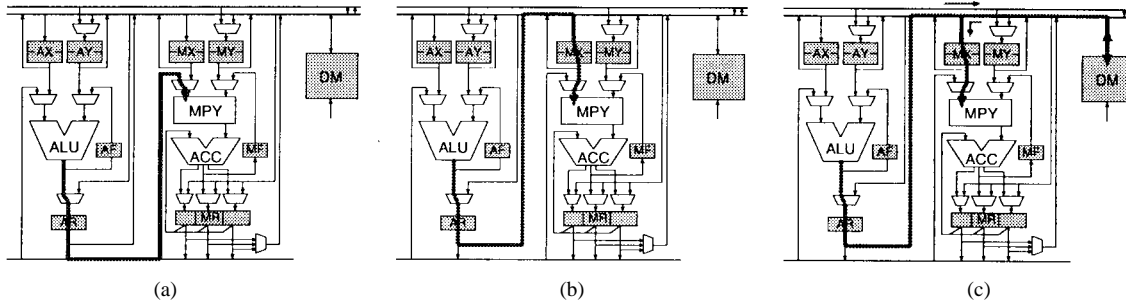
Value	Live range
$v_1$	_____
$v_2$	_____
$v_3$	_____
$v_4$	_____
$v_5$	_____
$v_6$	_____

(a)



(b)

Fig. 11. Register allocation based on graph coloring: (a) live ranges displayed on a time axis and (b) interference graph.



(a)

(b)

(c)

Fig. 12. Three alternative register allocations for the multiplication operand in the symmetrical FIR filter. The route followed is indicated in bold: (a) storage in AR, (b) storage in AR followed by MX, and (c) spilling to data memory DM. The last two alternatives require the insertion of extra register transfers.

in which the graph coloring procedure is called iteratively [67]. Further improvements of these principles have been described [68]–[70].

In practice, several of the assumptions made above may not be satisfied. First of all, most practical processors have a *heterogeneous* register structure. Extensions of the technique have been proposed, to take register classes into account during graph coloring [68], [70]. Furthermore, the graph coloring approach assumes that the live range of each value is known *beforehand*. Recent papers investigate the interaction between register allocation and scheduling [71], [72].

2) *Data Routing*: The above mentioned extension of graph coloring toward heterogeneous register structures has been applied to general-purpose processors, which typically have a *few* register classes (e.g., floating-point registers, fixed-point registers, and address registers). DSP and ASIP architectures often have a strongly heterogeneous register structure with *many* special-purpose registers.

In this context, more specialized register allocation techniques have been developed, often referred to as *data routing* techniques. To transfer data between functional units via intermediate registers, specific routes may have to be followed. The selection of the most appropriate route is nontrivial. In some cases indirect routes may have to be followed, requiring the insertion of extra register-transfer operations. Therefore an efficient mechanism for phase coupling between *register allocation* and *scheduling* becomes essential [73].

As an illustration, Fig. 12 shows a number of alternative solutions for the multiplication operand of the symmetrical

FIR filter application, implemented on the ADSP-21xx processor (see Fig. 8).

Several techniques have been presented for data routing in compilers for embedded processors. A first approach is to determine the required data routes *during the execution of the scheduling algorithm*. This approach was first applied in the *Bulldog* compiler for VLIW machines [18], and subsequently adapted in compilers for embedded processors like the *RL compiler* [48] and *CBC* [74]. In order to prevent a combinational explosion of the problem, these methods only incorporate local, greedy search techniques to determine data routes. The approach typically lacks the power to identify good candidate values for *spilling* to memory.

A global data routing technique has been proposed in the *Chess* compiler [75]. This method supports many different schemes to route values between functional units. It starts from an unordered description, but may introduce a partial ordering of operations to reduce the number of overlapping live ranges. The algorithm is based on branch-and-bound searches to insert new data moves, to introduce partial orderings, and to select candidate values for spilling. Phase coupling with scheduling is supported, by the use of probabilistic scheduling estimators during the register allocation process.

### E. Memory Allocation and Address Generation

A problem related to register allocation is the allocation of data memory locations for (scalar) data values in the intermediate representation. This is important, e.g., when

memory spills have been introduced in the register allocation phase, or for passing argument values in the case of function calls. Often these values will be stored in a stack frame in data memory. The memory assignment for array data values, e.g., [97], is beyond the scope of this paper.

In [76], an approach was described for memory allocation, using a graph coloring technique comparable to the register allocation method described previously.

An important issue is the addressing of values in a stack frame in memory. Typically a pointer is maintained to the stack frame. In conventional architectures, updating the pointer for the next access may require several instructions. However, as discussed in Section II, DSP processors and ASIP's typically have specialized address generation units which support address modifications in parallel with normal arithmetic operations. Often this is implemented by means of *postmodification*, i.e., the next address can be calculated by adding a modifier value to the current address while the current memory access is taking place. In this way the address pointer can be updated without an instruction cycle penalty. In some cases the modifier value is restricted to  $+1$  or  $-1$ .

When pointer modification is supported, it is advantageous to allocate memory locations in such a way that consecutively ordered memory accesses use adjacent memory locations—or locations that are close enough to each other to permit the use of pointer modification. This optimization, which is typical for DSP processors, was first described by Bartley [77] in the context of the TMS320C2X processor which supports post-increment and post-decrement instructions. In Bartley's formalism, an undirected graph is used with vertices corresponding to data values and edges reflecting the preferences for using neighboring storage locations for value pairs. A solution with a maximal number of post-increments and -decrements is obtained by finding a Hamiltonian path in the graph. Since this is an NP-complete problem, heuristic algorithms are proposed. Bartley's approach has been refined by other authors [78], [79].

## F. Scheduling

As already explained in the previous sections, compilers for CISC processors typically integrate code selection, local register allocation and instruction ordering in a single phase. Due to the lack of instruction level parallelism, no additional scheduling (or code compaction) phase is required.

The scheduling task is essential however for architectures that exhibit *pipeline hazards* or *instruction-level parallelism*. The former is the case in many RISC architectures. In VLIW and superscalar architectures, both features are found. Therefore the scheduling task has gained importance in software compilation, with the introduction of these architectural paradigms.

DSP processors and ASIP's can have a moderate to high degree of instruction-level parallelism. For example, these processors typically allow several data moves in parallel with arithmetic instructions (see the example of Fig. 2). Even when the parallelism is restricted, sched-

uling is a crucial task for these targets, because of the requirement of *high code quality*, which implies that the scarce architectural resources should be used as efficiently as possible, including the possibilities for data pipelining. This is especially true for deeply nested blocks in the algorithmic specification.

1) *Local Versus Global Scheduling*: A *local scheduler* is a scheduler that operates at the level of basic blocks (i.e., linear sequences of code without branching) in the intermediate representation. A well-known local scheduling technique is *list scheduling* [80]. More recently, in the context of embedded processors, integer-programming based scheduling formalisms have been described [81]–[83].

When the architecture has only a restricted amount of instruction-level parallelism, a local scheduling approach may already produce efficient results. Note however that this assumes that the scheduler has access to a detailed conflict model for partial instructions, describing precisely all structural and instruction encoding conflicts.

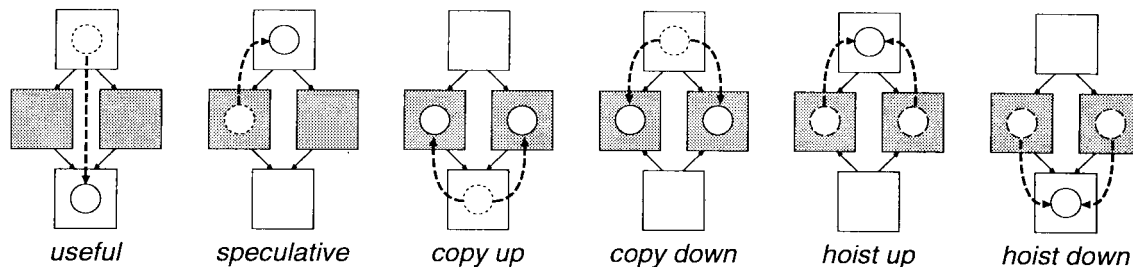
However, in the case of more parallel architectures, including most DSP's and ASIP's, there may be a mismatch between the *architectural* parallelism offered by the processor and the *algorithmic* parallelism within individual basic blocks. To use the processor's resources effectively, a *global scheduling* approach is required, whereby partial instructions can be moved across basic block boundaries. These moves are also termed *code motions*. Code motions may only be applied when they do not change the semantics of the overall program.

Fig. 13 illustrates several important types of code motions, in the presence of *conditional branches* [84]. A *useful* code motion moves instructions across a complete conditional branch. *Speculative execution* implies that a conditional instruction will be executed unconditionally. Special care is required to assure that the result has no effect when the branch in which the instruction resided originally is not taken. *Copy up* and *copy down* motions result in code duplication into conditional blocks. *Code hoisting* means that identical instructions in mutually exclusive conditional branches are merged and executed unconditionally.

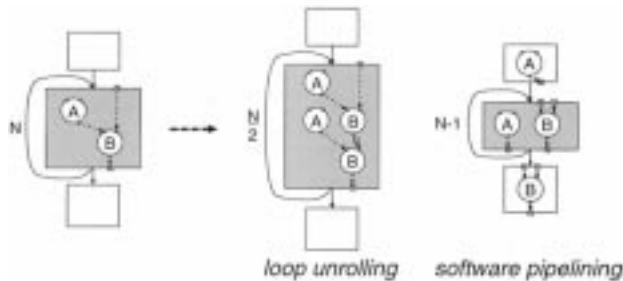
Another important class of code motions relates to *iterators* in the program, as illustrated in Fig. 14. *Loop unrolling* is a standard transformation whereby consecutive iterations of a loop are scheduled as a large basic block. *Software pipelining* is a transformation that restructures the loop, by moving operations from one loop iteration to another. Both transformations result in a larger amount of parallelism in the eventual loop body. Due to the frequent occurrence of iterators in signal processing applications, these transformations are of crucial importance in compilers for DSP's and ASIP's.

2) *Global Scheduling Techniques for Conditional Branches*: Global scheduling has been given a lot of attention in the context of VLIW architectures. Several of these techniques can be reused in the case of embedded processors like DSP's and ASIP's.

*Trace scheduling* is a global scheduling technique developed for VLIW's [85]. Based on execution probabilities of conditional branches, traces (i.e., linear sequences of



**Fig. 13.** Different types of code motions in a global scheduler. White and gray boxes represent unconditional and conditional basic blocks, respectively. Dotted and solid circles represent partial instruction before and after code motion, respectively. Dotted arrows symbolize code motion.



**Fig. 14.** Loop transformations in a global scheduler. Gray boxes represent a loop body, the number of iterations of which is indicated to its left. White boxes represent the loop's pre- and post-amble.

basic blocks) are identified. Each trace is scheduled as if it were a big basic block. Hence, operations in a trace can move beyond the original basic block boundaries. When this happens, a bookkeeping mechanism inserts the necessary compensation code in the other (less critical) traces to guarantee that semantical correctness is preserved. Improvements of the bookkeeping mechanism have been presented in [86].

A related technique is *superblock scheduling* [87]. A superblock is a linear sequence of basic blocks that has a single entry point only. A trace structure can be transformed into a superblock structure by a process called tail duplication. Superblocks are then scheduled in a way similar to traces. Compared to trace scheduling, the advantage is that the bookkeeping mechanism can be simplified considerably.

*Percolation based scheduling* is based on a complete set of semantics preserving transformations to move instructions between basic blocks [88]. Instructions are moved repeatedly in the upward direction, between *adjacent* basic blocks. A drawback of this strategy is that longer moves (e.g., the useful code motion of Fig. 13) are only possible if the incremental moves between adjacent blocks that compose the long move are all beneficial.

In [89] a global code motion technique is proposed, in which only those types of motions are supported that do not require any code duplication. In [90] a technique is proposed based on the concept of *region scheduling*. In [84] a global code motion tool is presented for DSP and ASIP architectures, that makes use of fast probabilistic estimators for schedule length and register occupation, in order to trade different possible code motions. The actual scheduling is

done afterwards, in a separate phase. This has the advantage that the code motion tool can also be invoked at earlier stages in code generation, e.g., before register allocation.

3) *Software Pipelining*: Techniques for software pipelining can be divided in two categories: those that iteratively call a local scheduler to evaluate the effect of certain moves, and those that incorporate software pipelining in a single global scheduling algorithm.

*Modulo scheduling*, presented in [91], first converts conditional branches in the loop into straight line code and subsequently applies a local scheduling algorithm that pipelines the loop. *Loop folding* [92] is an iterative approach to software pipelining. In every step of the algorithm, a local list schedule is computed for the loop body. Based on this schedule, partial instructions are selected and moved between loop iterations. A similar strategy has been added to the global code motion tool of [84] (mentioned previously). As a result, code motions across conditional branch boundaries and software pipelining can be incorporated in the same algorithm.

Examples of global scheduling algorithms that perform software pipelining include *enhanced pipeline scheduling* [93] and *GURPR* [94]. The latter of these methods is based on partial loop unrolling, after which a parallel schedule is composed for the unrolled loop, followed by a rerolling step.

Finally, note that some VLIW architectures have special hardware support to facilitate the implementation of software pipelining [95], [96]. Typically special register sets are dedicated to data communication between loop iterations.

## V. CONCLUSION

As motivated in Section I and in the companion paper on application and architectural trends [1], embedded processor cores represent a key component in contemporary and future systems for telecommunication and multimedia. Core processor technology has created a new role for general-purpose DSP's. In addition, there is a clear and important use of ASIP's. For products manufactured in large volumes, ASIP's are clearly more cost efficient, while power dissipation can be reduced significantly. These advantages are obtained without giving up the flexibility of a programmable solution.

The lack of suitable design technologies to support the phases of processor development and of application programming however remains a significant obstacle for sys-

tem design teams. One of the goals of this paper was to motivate an increased research effort in the area of CAD for embedded system design.

In this paper we have focused primarily on the issue of software compilation technologies for embedded processors. Our starting point was the observation that many commercially available C compilers, especially for fixed-point DSP's, are unable to take full advantage of the architectural features of the processor. In the case of ASIP's, compiler support is nonexistent due to the lack of retargeting capabilities of the existing tools.

Many of these compilers are employing traditional code generation techniques, developed in the 1970's and 1980's in the software compiler community. These techniques were primarily developed for general-purpose microprocessors, which have highly regular architectures with homogeneous register structures, without many of the architectural peculiarities that are typical of fixed-point DSP's.

In the past five years, however, new research efforts emerged in the area of software compilation, focusing on embedded DSP's and ASIP's. Many of these research teams are operating on the frontier of software compilation and high-level VLSI synthesis. The synergy between both disciplines has already resulted in a number of new techniques for modeling of (irregular) instruction-set architectures and for higher quality code generation. Besides code quality, the issue of architectural retargetability is gaining a lot of attention. Retargetability is an essential feature of a software compilation environment in the context of embedded processors, due to the increasingly shorter lifetime of a processor and due to the requirement to use ASIP's.

In this paper we have outlined the main architectural features of contemporary DSP's and ASIP's, that are relevant from a software compilation point of view. A classification of architectures has been presented, based on a number of elementary characteristics. Proper understanding of processor architectures is a prerequisite for successful compiler development. In addition, a survey has been presented of existing software compilation techniques that are considered relevant in the context of DSP's and ASIP's for telecom, multimedia, and consumer applications. This survey also covered recent research in retargetable software compilation for embedded processors.

In addition to retargetable software compilation, there are several other important design technology issues, that have not been discussed in this paper. The authors believe the following will become increasingly important in the future.

- System level algorithmic optimizations. Many specifications of systems are produced without a precise knowledge of the implications on hardware and software cost. Important savings are possible by carrying out system level optimizations, such as control-flow transformations to optimize the memory and power cost of data memories.
- System partitioning and interface synthesis. Whereas the problems of hardware synthesis and software compilation are reasonably well understood, the design of the glue between these components is still done manually, and therefore error-prone.

- Synthesis of real-time kernels. A kernel takes care of run-time scheduling of tasks, taking into account the interaction with the system's environment. In some cases, general-purpose operating systems are used. However, these solutions are expensive in terms of execution speed and code size. Recent research is therefore focusing on the automatic synthesis of lightweight, application-specific kernels that obey user-specified timing constraints.

## REFERENCES

- [1] P. G. Paulin, C. Liem, M. Cornero, F. Naçabal, and G. Goossens, "Embedded software in real-time signal processing systems: Application and architecture trends," *Proc. IEEE*, this issue, pp. 419-435.
- [2] J. Morse and S. Hargrave, "The increasing importance of software," *Electronic Design*, vol. 44, no. 1, Jan. 1996.
- [3] P. G. Paulin *et al.*, "Trends in embedded systems technology: An industrial perspective," in *Hardware/Software Co-Design*, G. De Micheli and M. Sami, Eds. Boston: Kluwer, 1996.
- [4] D. Bursky, "Tuned RISC devices deliver top performance," *Electronic Design*, pp. 77-100, Mar. 18, 1996.
- [5] A. van Someren and C. Atack, *The ARM RISC Chip, a Programmer's Guide*. Reading, MA: Addison-Wesley, 1994.
- [6] S. Berger, "An application specific DSP for personal communications applications," in *Proc. DSPx Expos. and Symp.*, June 1994.
- [7] A. Bindra, "Two 'Lode' up on TCSI's new DSP core," *EE Times*, Jan. 16, 1995.
- [8] R. A. M. Beltman *et al.*, "EPICS10: Development platform for next generation EPICS DSP products," in *Proc. 6th Int. Conf. on Signal Proc. Applic. and Technol.*, Oct. 1995.
- [9] *D950-Core Specification*, doc. no. 2509, SGS-Thomson Microelectron., Grenoble, France, Jan. 1995.
- [10] V. Živojnović *et al.*, "DSPstone: A DSP-oriented benchmarking methodology," in *Proc. Int. Conf. on Signal Proc. Applic. and Technol.*, Oct. 1994.
- [11] P. Marwedel and G. Goossens, *Code Generation for Embedded Processors*. Boston: Kluwer, 1995.
- [12] G. Goossens *et al.*, "Programmable chips in consumer electronics and telecommunications: Architectures and design technology," in *Hardware/Software Co-Design*, G. De Micheli and M. Sami, Eds. Boston: Kluwer, 1996.
- [13] E. A. Lee, "Programmable DSP architectures: Part I & Part II," *IEEE ASSP Mag.*, Dec. 1988 and Jan. 1989.
- [14] "ADSP-2100 user's manual," Norwood: Analog devices, 1989.
- [15] *TMS320C54x, TMS320LC54x, TMS320VC54x Fixed-Point Digital Signal Processors*, Houston: Texas Instrum., 1996.
- [16] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.
- [17] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1990.
- [18] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: MIT Press, 1986.
- [19] L. Bergher *et al.*, "MPEG audio decoder for consumer applications," in *Proc. IEEE Custom Integr. Circ. Conf.*, May 1995.
- [20] G. Essink, "Architecture and programming of a VLIW style programmable video signal processor," in *Proc. 24th ACM/IEEE Int. Symp. on Microarchitecture*, Nov. 1991.
- [21] P. Clarke and R. Wilson, "Philips preps VLIW DSP for multimedia," *EE Times*, p. 1, Nov. 14, 1994.
- [22] D. Lanneer *et al.*, "Chess: Retargetable code generation for embedded DSP processors," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Boston: Kluwer, 1995.
- [23] R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, June 1993.
- [24] J. Van Praet *et al.*, "Modeling hardware-specific data-types for simulation and compilation in HW/SW co-design," in *Proc. 6th Workshop on Synth. and Syst. Integr. of Mixed Technol.*, Nov. 1996.

- [25] R. Cytron *et al.*, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Prog. Lang. and Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.
- [26] M. C. McFarland, A. C. Parker, and R. Camposano, "The high level synthesis of digital systems," in *Proc. IEEE*, vol. 78, pp. 301–318, Feb. 1990.
- [27] J. T. J. van Eijndhoven and L. Stok, "A data flow graph exchange standard," in *Proc. Europe. Design Autom. Conf.*, Mar. 1992, pp. 193–199.
- [28] A. V. Aho *et al.*, *Compilers—Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
- [29] D. Landskov *et al.*, "Local microcode compaction techniques," *ACM Comp. Surveys*, vol. 12, no. 3, pp. 261–294, Sept. 1980.
- [30] M. Johnson, *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
- [31] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," in *Proc. 15th Micro*, 1982, pp. 125–133.
- [32] S. Bashford *et al.*, *The Mimola Language*, vers. 4.1, Techn. Rep., Univ. Dortmund, Sept. 1994.
- [33] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*. New York: McGraw-Hill, 1991.
- [34] M. R. Barbacci, "Instruction set processor specifications (ISPS): The notation and its applications," *IEEE Trans. Computer*, Jan. 1981.
- [35] A. Fauth *et al.*, "Describing instruction set processors using nML," in *Proc. Europe. Design and Test Conf.*, Mar. 1995.
- [36] A. Fauth, "Beyond tool-specific machine descriptions," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Boston: Kluwer, 1995.
- [37] A. V. Aho and S. C. Johnson, "Optimal code generation for expression trees," *J. ACM*, vol. 23, no. 3, pp. 488–501, July 1976.
- [38] A. V. Aho *et al.*, "Code generation using tree matching and dynamic programming," *ACM Trans. Prog. Lang. and Syst.*, vol. 11, no. 4, pp. 491–516, Oct. 1989.
- [39] C. W. Fraser *et al.*, "Burg—Fast optimal instruction selection and tree parsing," *ACM Sigplan Notices*, vol. 27, no. 4, pp. 68–76, Apr. 1992.
- [40] ———, "Engineering a simple, efficient code-generator generator," *ACM Lett. on Prog. Lang. and Syst.*, vol. 1, no. 3, pp. 213–226, Sept. 1993.
- [41] R. S. Glanville and S. L. Graham, "A new method for compiler code generation," in *Proc. 5th ACM Ann. Symp. on Principles of Prog. Lang.*, 1978.
- [42] R. Leupers and P. Marwedel, "Instruction selection for embedded DSP's with complex instructions," in *Proc. Europe. Design Autom. Conf.*, Sept. 1996.
- [43] G. Araujo and S. Malik, "Optimal code generation for embedded memory nonhomogeneous register architectures," in *Proc. 8th Int. Symp. on Syst. Synthesis*, Sept. 1995.
- [44] R. Leupers and P. Marwedel, "A BDD-based frontend for retargetable compilers," in *Proc. Europe. Design and Test Conf.*, Mar. 1996, pp. 239–243.
- [45] P. G. Paulin *et al.*, "Flexware: A flexible firmware development environment for embedded systems," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Boston: Kluwer, pp. 67–84, 1995.
- [46] B. Wess, "Code generation based on trellis diagrams," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Boston: Kluwer, 1995, pp. 188–202.
- [47] L. Nowak and P. Marwedel, "Verification of hardware descriptions by retargetable code generation," in *Proc. 26th ACM/IEEE Design Autom. Conf.*, June 1989, pp. 441–447.
- [48] K. Rimey and P. N. Hilfinger, "A compiler for application-specific signal processors," in *VLSI Signal Processing*, vol. 3. New York: IEEE Press, 1988, pp. 341–351.
- [49] J. Van Praet *et al.*, "A graph based processor model for retargetable code generation," in *Proc. Europe. Design and Test Conf.*, Mar. 1996.
- [50] J. Bruno and R. Sethi, "Code generation for a one-register machine," *J. ACM*, vol. 23, no. 3, pp. 502–510, July 1976.
- [51] A. V. Aho *et al.*, "Code generation for expressions with common subexpressions," *J. ACM*, vol. 24, no. 1, pp. 146–160, Jan. 1977.
- [52] R. Sethi and J. D. Ullman, "The generation of optimal code for arithmetic expressions," *J. ACM*, vol. 17, no. 4, pp. 715–728, Oct. 1970.
- [53] H. Emmelmann *et al.*, "Beg—A generator for efficient back ends," in *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implem.*, 1989, pp. 227–237.
- [54] E. Pelegri-Llopert, "Optimal code generation for expression trees: An application of BURS theory," in *Proc. 15th ACM Symp. Principles of Prog. Lang.*, 1988, pp. 294–308.
- [55] T. A. Proebsting, "Simple and efficient BURS table generation," in *Proc. SIGPLAN Conf. Prog. Lang. Design and Implem.*, 1992.
- [56] R. Wilhelm and D. Maurer, *Compiler Design*. Reading, MA: Addison-Wesley, 1995.
- [57] C. Liem *et al.*, "Instruction-set matching and selection for DSP and ASIP code generation," in *Proc. Europe. Design and Test Conf.*, Feb. 1994.
- [58] S. L. Graham *et al.*, "An experiment in table driven code generation," in *Proc. SIGPLAN Symp. Compiler Construction*, 1982, pp. 32–43.
- [59] M. Ganapathi and C. N. Fisher, "Affix grammar driven code generation," *ACM Tr. Prog. Lang. and Syst.*, vol. 7, no. 4, pp. 347–364, Apr. 1984.
- [60] M. Mahmood *et al.*, "A formal language model of microcode synthesis," in *Formal VLSI Specification and Synthesis*, L. Claesen, Ed. Amsterdam: North Holland, 1990, pp. 23–41.
- [61] S. Liao, "Code generation and optimization for embedded digital signal processors," Ph.D. dissertation, MIT, June 1996.
- [62] J. W. Davidson and C. W. Fraser, "Code selection through object code optimization," *ACM Trans. Prog. Lang. and Syst.*, Oct. 1984.
- [63] ———, "The design and application of a retargetable peephole optimizer," *ACM Trans. Prog. Lang. and Syst.*, vol. 2, no. 2, pp. 191–202, Apr. 1980.
- [64] P. Marwedel, "Tree-based mapping of algorithms to predefined structures," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 586–593.
- [65] R. P. Gurd, "Experience developing microcode using a high-level language," in *Proc. 16th Annu. Microprog. Workshop*, Oct. 1983.
- [66] C. Liem *et al.*, "Industrial experience using rule-driven retargetable code generation for multimedia applications," in *Proc. IEEE/ACM Int. Symp. on Syst. Synthesis*, Sept. 1995.
- [67] G. J. Chaitin, "Register allocation and spilling via graph coloring," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 98–105, June 1982.
- [68] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Trans. Prog. Lang. and Syst.*, Oct. 1990.
- [69] D. Callahan and B. Koblenz, "Register allocation via hierarchical graph coloring," *ACM SIGPLAN Notices*, June 1991.
- [70] P. Briggs, "Register allocation via graph coloring," Ph.D. dissertation, Rice Univ., Houston, Apr. 1992.
- [71] S. Pinter, "Register allocation with instruction scheduling: A new approach," *SIGPLAN Notices*, June 1993.
- [72] W. Ambrosch *et al.*, "Dependence conscious global register allocation," in *Programming Language and System Architecture*, J. Gutknecht, Ed. Berlin: Springer, 1994.
- [73] S. Freudenberg and J. Ruttenberg, "Phase-ordering of register allocation and instruction scheduling," in *Code Generation-Concepts, Tools, Techniques*, R. Giegerich and S. Graham, Eds. Berlin: Springer, 1991.
- [74] R. Hartmann, "Combined scheduling and data routing for programmable ASIC systems," in *Proc. Europe. Conf. on Design Autom.*, Mar. 1992, pp. 486–490.
- [75] D. Lanneer *et al.*, "Data routing: A paradigm for efficient datapath synthesis and code generation," in *Proc. 7th ACM/IEEE Int. Symp. on High-Level Synth.*, May 1994, pp. 17–22.
- [76] A. Sudarsanam and S. Malik, "Memory bank and register allocation in software synthesis for ASIP's," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 1995, pp. 388–392.
- [77] D. H. Bartley, "Optimizing stack frame accesses for processors with restricted addressing modes," *Software-Practice and Experience*, vol. 22, no. 2, pp. 101–110, Feb. 1992.
- [78] S. Liao *et al.*, "Storage assignment to decrease code size," *ACM SIGPLAN Notices*, vol. 30, no. 7, pp. 186–195, June 1995.
- [79] R. Leupers and P. Marwedel, "Algorithms for address assignment in DSP code generation," in *Proc. IEEE Int. Conf. Computer Aided Design*, Nov. 1996.
- [80] S. Davidson *et al.*, "Some experiments in local microcode compaction for horizontal machines," *IEEE Trans. Computers*, vol. C-30, no. 7, pp. 460–477, July 1981.

- [81] T. Wilson *et al.*, "An integrated approach to retargetable code generation," in *Proc. 7th ACM/IEEE Int. Symp. High-Level Synthesis*, May 1994, pp. 70–75.
- [82] R. Leupers and P. Marwedel, "Time-constrained code compaction for DSP's," in *Proc. 8th Int. Symp. Syst. Synthesis*, Sept. 1995, pp. 239–243.
- [83] F. Depuydt, "Register optimization and scheduling for real-time digital signal processing architectures," Ph.D. dissertation, Kath. Univ. Leuven, Belgium, Nov. 1993.
- [84] A. Kifli, "Global scheduling in high-level synthesis and code generation for embedded processors," Ph.D. dissertation, Kath. Univ. Leuven, Belgium, Nov. 1996.
- [85] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Computers*, vol. C-30, pp. 478–490, July 1981.
- [86] T. Gross and M. Ward, "The suppression of compensation code," *ACM Trans. Prog. Lang. and Systems*, Oct. 1991.
- [87] W. M. Hwu *et al.*, "The superblock: An effective technique for VLIW and superscalar compilation," *J. Supercomputing*, 1993.
- [88] A. Nicolau, "Percolation scheduling: A parallel compilation technique," Tech. Rep. TR85-678, Cornell Univ., May 1985.
- [89] D. Bernstein *et al.*, "Code duplication: An assist for global instruction scheduling," in *Proc. ACM Micro-24*, 1991, pp. 103–113.
- [90] V. H. Allan *et al.*, "Enhanced region scheduling on a program dependence graph," in *Proc. ACM Micro-25*, 1992, pp. 72–80.
- [91] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implement.*, 1988, pp. 318–328.
- [92] G. Goossens *et al.*, "Loop optimization in register-transfer scheduling for DSP-systems," in *Proc. 26th IEEE/ACM Design Autom. Conf.*, June 1989.
- [93] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," *2nd Workshop Lang. and Compil. for Paral. Comp.*, Aug. 1989, pp. 213–229.
- [94] B. Su *et al.*, "GURPR: A method for global software pipelining," in *Proc. ACM MICRO-20*, 1987.
- [95] B. R. Rau *et al.*, "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," in *Proc. 9th Annu. Symp. Comp. Archit.*, Apr. 1982, pp. 131–139.
- [96] B. Su *et al.*, "A software pipelining based VLIW architecture and optimizing compiler," in *Proc. MICRO-23*, 1990, pp. 17–27.
- [97] C. Liem *et al.*, "Address calculation for retargetable compilation and exploration of instruction-set architectures," in *Proc. 33rd ACM/IEEE Design Autom. Conf.*, June 1996.

**Gert Goossens** (Member, IEEE), for a photograph and biography, see this issue, pp. 435.



**Johan Van Praet** (Member, IEEE) received a degree in electrical engineering from the Katholieke Universiteit Leuven, Belgium, in 1990. Since 1991 he has been working toward the Ph.D. degree on retargetable software compilation technology at the same university.

In 1996, he co-founded Target Compiler Technologies, Leuven, Belgium, where he is responsible for product development. From 1991 to 1996 he was with IMEC as a Research Assistant. In 1990 he worked on the design

of a chip architecture for a GSM mobile phone, in a joint project of the Interuniversity Micro-Electronics Centre (IMEC) and Alcatel Bell Telephone in Belgium.



**Dirk Lanneer** (Member, IEEE) received a degree in electrical engineering and the Ph.D. degree in applied sciences from the Katholieke Universiteit Leuven, Belgium, in 1986 and 1993, respectively.

In 1996, he co-founded Target Compiler Technologies, Leuven, Belgium, where he is responsible for research and development. From 1986 to 1996 was a Technical Staff Member at the Interuniversity Micro-Electronics Centre (IMEC), Leuven, Belgium, where he started as

a Research Assistant of high-level synthesis projects, and co-initiated work on the "Chess" retargetable software compilation project.



**Werner Geurts** (Member, IEEE) received degrees in electrical engineering from the Industriële Hogeschool, Antwerp, Belgium, and from the Katholieke Universiteit Leuven, Belgium, in 1985 and 1988, respectively. He received the Ph.D. degree in electrical engineering from the Katholieke Universiteit Leuven in 1995.

In 1996, he co-founded Target Compiler Technologies, where he is responsible for product development. From 1989 to 1996 he was with the VLSI Design Methodologies Division of

the Interuniversity Micro-Electronics Centre (IMEC), Leuven, Belgium, where he worked on high-level synthesis techniques for high throughput applications, and on retargetable software compilation for application-specific DSP cores.



**Augustli Kifli** received the B.Sc. degree in electrical engineering from the National Taiwan University, Taiwan, in 1987, and the M.Sc. and Ph.D. degrees in applied sciences from the Katholieke Universiteit Leuven, Belgium, in 1990 and 1996, respectively.

From 1993 to 1996 he was a member of Interuniversity Micro-Electronics Centre's (IMEC) retargetable software compilation group. From 1990 to 1993 was a Research Assistant at IMEC, initially working in the area

of high-level synthesis.

**Clifford Liem**, for a photograph and biography, see this issue, pp. 435.

**Pierre G. Paulin** (Member, IEEE), for a photograph and biography, see this issue, pp. 434.