Design Automation for Embedded Systems, vol. 3, no. 1, Jan 1998, , 1–36 ()

Retargetable Code Generation based on Structural Processor Descriptions

RAINER LEUPERS AND PETER MARWEDEL

leupers|marwedel@ls12.informatik.uni-dortmund.de

University of Dortmund, Department of Computer Science 12, 44221 Dortmund, Germany

Abstract. Design automation for embedded systems comprising both hardware and software components demands for code generators integrated into electronic CAD systems. These code generators provide the necessary link between software synthesis tools in HW/SW codesign systems and embedded processors. General-purpose compilers for standard processors are often insufficient, because they do not provide flexibility with respect to different target processors and also suffer from inferior code quality. While recent research on code generation for embedded processors has primarily focussed on code quality issues, in this contribution we emphasize the importance of retargetability, and we describe an approach to achieve retargetability. We propose usage of uniform, external target processor models in code generation, which describe embedded processors by means of RT-level netlists. Such structural models incorporate more hardware details than purely behavioral models, thereby permitting a close link to hardware design tools and fast adaptation to different target processors. The MSSQ compiler, which is part of the MIMOLA hardware design system, operates on structural models. We describe input formats, central data structures, and code generation techniques in MSSQ. The compiler has been successfully retargeted to a number of real-life processors, which proves feasibility of our approach with respect to retargetability. We discuss capabilities and limitations of MSSQ, and identify possible areas of improvement.

Keywords: retargetable compilation, processor modelling, embedded code generation

1. Introduction

Embedded systems are special-purpose computing systems, designed and installed only once to serve a single, particular application. They interact with larger, sometimes non-electronic environments. Today, embedded systems are found in many areas of everyday life, such as telecommunication, control in vehicles and aircraft, home appliances, or medical equipment. The complexity of embedded systems and stringent time-to-market constraints demand for design automation tools that provide CAD support right from the *system level*. Economically reasonable implementations of embedded systems in general consist of both hardware and software components, resulting in the problem of *hardware-software codesign*. Parts of the system, which are not subject to tight computation-speed constraints are implemented through software running on programmable *embedded processors*, such as RISCs, DSPs, and microcontrollers, while other parts require fast, dedicated hardware. It is favorable to implement as much of a system as possible in software, because software is easier to develop, and usage of off-the-shelf processors significantly reduces design costs. Moreover, software is much more flexible than hardware, thus allowing for accomodation of late specification changes and easier debugging.

The most widespread approach to design automation for combined HW/SW systems is to iteratively perform HW/SW partitioning, followed by hardware and software synthesis and co-simulation. This concept is realized in a number of experimental HW/SW codesign systems like VULCAN [1], CHINOOK [2], COSYMA [3], and CODES [4]. In those systems, code generation for embedded processsors is performed in a very abstract fashion: After HW/SW partitioning, *software synthesis* transforms pieces of system functionality, which have been assigned to software, from an internal representation into *program threads*. Program threads are linearized sets of abstract machine-independent operations, which are in turn translated into high-level language source code, typically C. It is assumed that processor-specific compilers are available for mapping C to machine code for standard processors, such as R3000, SPARC, and Intel 8086. By using C code as an intermediate representation, cost estimation during HW/SW partitioning must remain rather coarse, possibly leading to the necessity of many HW/SW partitioning iterations. This disadvantage is partially avoided in systems, which directly generate machine code instead of high-level language programs, and therefore permit more detailed cost metrics: PTOLEMY [5] is capable of code generation for some standard DSPs (Motorola 56001 and 96002) based on macro-expansion. CASTLE [6] uses a C compiler that maps source code into vertical machine code for VLIW architectures, but does not handle predefined processors with non-horizontal instruction formats. Synopsys' commercial DSP synthesis tool COSSAP supports both C and assembly code generation.

In all of the above systems code generation is restricted to a narrow range of target processors. However, one of the major requirements on embedded system design tools is the capability of exploring a large design space in order to find an implementation that meets all design constraints. Design space exploration includes investigation of different programmable processors that execute the software components of the embedded system to be implemented. In an ideal embedded system CAD environment, the processor type is transparent for the user: many alternatives can be tried out by re-compiling the software onto each different processor. The processor which meets the constraints at minimum costs is selected. Such a degree of flexibility or *retargetability* is however hardly provided by current HW/SW codesign tools.

This problem is intensified, if embedded systems are realized as *single-chip* systems. The advent of deep submicron VLSI technology created a trend towards design of complete systems on a single chip, resulting in higher speed and dependability at lower silicon area and power consumption [7, 8]. Several vendors (e.g. Texas Instruments, LSI Logic, Advanced RISC Machines) offer standard processors in form of *cores*, i.e. layout macrocells which can be instantiated by a design engineer from a component library. However, a particular application might not require the full amount of capabilities of a standard processor. Using standard processor cores thus leads to a possible waste of silicon area and power consumption for single-chip systems. As a consequence, system houses are starting to use *flexible, customizable* programmable architectures. For these, the term ASIP (*application-specific instruction-set processor*) has been created. Industrial system design using ASIPs is reported for instance in [9]. For ASIPs, only the coarse architecture is fixed in advance, so that a designer actually can trade silicon area against computation speed. The ASIP is tailored towards a specific application by repeatedly mapping program sources onto different detailed architectures. This process can be considered as HW/SW codesign at the *processor level*. Removing or adding hardware components from/to an ASIP has immediate consequences on the execution speed of software. Obviously, such a scenario for customization of ASIPs demands for retargetable compilers.

Besides retargetability, also *code quality* is of major concern, in particular for embedded systems involving signal processing under real-time constraints. For those systems, DSPs are the preferred type of embedded processors. Standard DSPs are available with software development tools such as assemblers, debuggers, simulators, and C compilers. The code quality of current compilers for DSPs is, however, rather poor. The overhead of compiler-generated code compared to hand-crafted code sometimes reaches several hundred percent [10], which is unacceptable in most cases. Essentially, this is due to the fact, that DSPs show highly specialized architectures and instruction sets, which demand for dedicated code generation techniques. For instance, exploitation of potential instruction-level parallelism is a must for DSPs, but is often not provided by current compilers. While code generation for standard DSPs suffers from insufficient code quality of compilers, the situation for ASIPs is even more unsatisfactory: Since ASIPs are *in-house designs*, only used for a small number of applications before becoming obsolete, there is hardly any high-level language compiler support for ASIPs. Insufficient quality and availability of compilers lead to the fact that even nowadays DSPs are mostly programmed in assembly languages, which implies all the well-known disadvantages of low-level programming. As time-to-market is now the most important issue for VLSI system houses, taking the step towards high-level programming seems mandatory.

In summary, design automation for embedded systems demands for code generators *as a part of ECAD systems*. Such code generators bridge the gap between software synthesis tools in HW/SW codesign and machine programs running on embedded processors. Two major areas need to be tackled in order to provide more powerful code generators than currently available:

Retargetability: General-purpose compilers are processor-specific. If different target processors are to be investigated during embedded system design, then a number of different (potentially costly) compilers must be employed. Also interfacing problems may arise, if the compiler is part of a larger design environment. A *single* retargetable compiler, capable of mapping algorithms to different target processors, alleviates these problems. Furthermore, usage of processor-specific compilers restricts the range of possible target processors to standard components. However, due to the trend towards single-chip systems, ASIPs are expected to gain more and more importance in the near future. Usage of ASIPs demands for very flexible compilers and a close link between compilers and hardware design tools.

Code quality: In embedded systems operating under hard real-time constraints, insufficient quality of compiler-generated code may demand for higher clock rates than actually necessary. In turn, this increases power consumption, which is particularly a problem in portable devices. In case of single-chip systems, where program code is stored in on-chip ROM, any overhead in code quality immediately contributes to silicon area requirements. Therefore, high code quality is much more important for embedded systems than for general-purpose computers. This justifies usage of computation-time intensive algorithms, aiming at code optimization beyond the scope of traditional compilers, for which high compilation speed is a primary goal. However, high-quality code generation is aggrevated by the fact, that, in contrast to general-purpose processing, embedded processors often have a highly irregular architectures and a moderate degree of instruction-level parallelism.

Retargetability is inversely related to code quality. The more a compiler is tailored towards a certain target processor, the higher is the code quality and vice versa. In the MSSQ compiler, which is subject of this paper, the degree of retargetability is fixed. MSSQ generates code for a defined class of target processors, and within this class aims at code optimization, mainly by exploiting potential parallelism. We show how retargetability can be realized by means of target processor models specified as RT-level netlists in a hardware description language. The main advantages of this approach compared to related work are the following:

1. Usage of a hardware description language for target processor modelling provides a natural link to ECAD environments. In contrast to all other approaches, which often use particular and tool-specific description formalisms, a single and uniform model is sufficient for the complete design process, i.e. hardware design, code generation, and simulation. All aspects needed for code generation are automatically derived from that model.

2. While many publications on retargetable code generation do not clearly state the range of processors that can be handled, MSSQ operates on a well-defined class of target processors. Within this class, code can be generated for any target processor.

3. Obviously, the concept of RT-level netlists results in very detailed target processor models, which also capture primitive hardware components like wires, busses, and multiplexers. Although such detailed models sometimes demand for a higher description effort compared to pure instruction-set models, they are the most natural representation from a hardware designer's point of view. Compilers capable of using netlists instead of instruction-set models avoid any risks

in the communication between hardware designers and compiler writers. Moreover, RT-level models permit fast adaptation of local changes in the processor architecture, which might have global impact on an equivalent instruction-set model. This is especially important for designs based on ASIPs.

The organization of this paper is as follows: In section 2, we give a short overview of the MIMOLA hardware design system, of which the MSSQ compiler is a central component. Furthermore, we outline the functionality of MSSQ and describe the class of processors that can be handled with this compiler. Section 3 discusses previous work in the areas of compiler construction and microprogramming, as well as more recent work concerning embedded code generation. A detailed description of the input format of MSSQ is given in section 4. Section 5 presents the two main internal data structures in MSSQ: the connection operation graph and I-trees. Code generation techniques based on these data structures are described in section 6. Section 7 provides experimental results for several embedded processors, and the paper ends with concluding remarks.

2. System overview

The MIMOLA Design System (MDS) is a high-level hardware design environment, providing design automation between algorithmic and register-transfer level for digital programmable processors. The user interface of the MDS is the MIMOLA hardware description language. Besides the necessary frontends and utilities, four tools constitute the core of the MDS (fig. 1), which operate on the common intermediate representation TREEMOLA.

MSSH: MSSH performs high-level synthesis, i.e. it generates RTL structures from behavioral descriptions at the algorithmic level. MSSH exploits realistic, user-definable component libraries Furthermore, also partially predefined structures are taken into account. Details on MSSH techniques are also to be found in [11, 12].

MSSQ: In contrast to MSSH, MSSQ maps algorithms to completely predefined RTL structures by generating microcode. Many design decisions for MSSQ are based on its predecessor MSSV [13], which was, however, often too slow to be applied to real-life problems.

MSST: MSST aims at exploiting self-test capabilities of programmable processors. User-defined high-level self-test programs are compiled to machine code for predefined RTL structures [14]. A more recent self-test program compiler, which is based on experiences with MSST, is described in [15].

MSSB/U: MSSB is a simulator for algorithmic-level descriptions, which uses compiled-code simulation. Its counterpart MSSU is an event-driven RTL structure simulator.

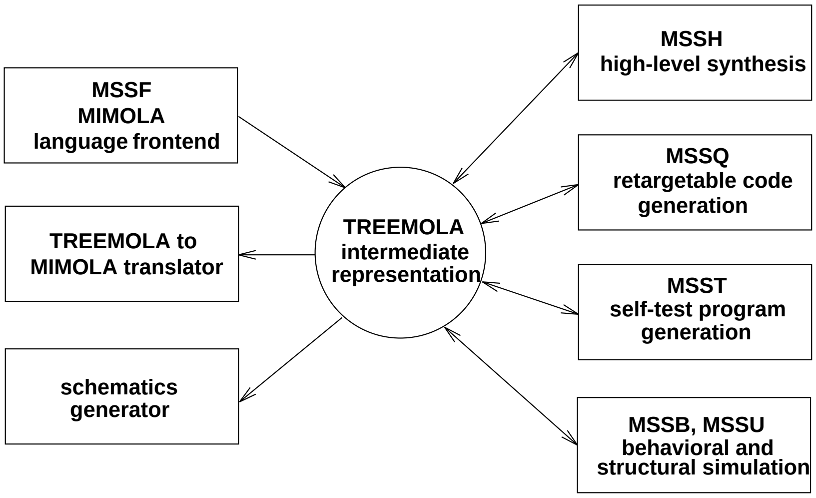


Figure 1. The MIMOLA Design System

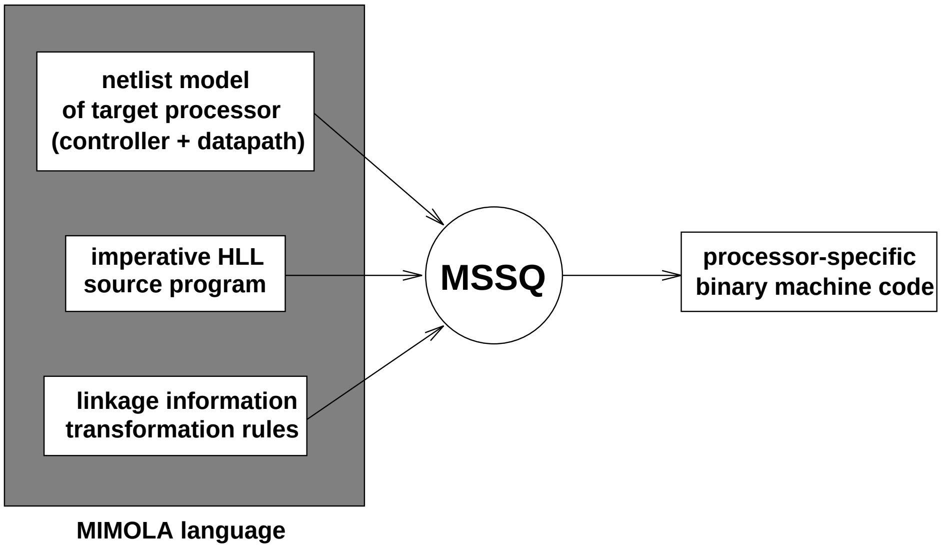


Figure 2. Functionality of the MSSQ compiler

As exemplified in [16], this combination of tools permits high-level design of digital programmable processors under realistic conditions. Typically, this involves several design iterations guided by user interaction.

Fig. 2 shows the functionality of the MSSQ compiler. MSSQ reads an external netlist model of the target processor. This netlist model comprises both the controller and the datapath. The source algorithm is specified as an imperative, high-level language program. Both netlist and program are described in the MIMOLA language. Additionally, the user can optionally specify information concerning linkage of hardware and software. MSSQ emits processor-specific, binary machine code, which implements the source algorithm on the specified target processor. This code listing can be interpreted as the program ROM specification for an embedded processor. The class of processors handled by MSSQ is defined by the following characteristics:

Microprogrammable controller: MSSQ assumes a microprogrammable controller architecture, in which all control lines originate from a dedicated instruction memory or register. The primitive operations steered by this controller are single-cycle *microoperations* or *register transfers* (RTs), of which several may be executable in parallel in each machine cycle. Instruction pipelining and multicycle operations are assumed to be invisible at this level. The detailed instruction format is part of the processor model and therefore completely user-definable. Possible instruction formats range from purely horizontal (VLIW) to strongly encoded formats. For encoded formats, all inter-instruction restrictions are derived from the processor model, which typically comprises instruction decoders in this case.

Arbitrary datapath: The datapath is an arbitrary, user-definable netlist of RT-level hardware components. The basic components, such as registers, ALUs, decoders, multiplexers, memories, and bus drivers, are described by their behavior. Interconnections between these components are described in terms of wires and busses. No assumptions are made about the overall architecture of the datapath in advance.

3. Related work

Early contributions to retargetable code generation mainly stem from the areas of compiler construction and microprogramming. Code generation for embedded processors as a separate research area has been established only in the beginning of this decade, and since then has evolved rapidly. We therefore divide the overview of related work into three categories.

3.1. Compiler construction

Retargetability has already been a goal in the UNCOL project [17]. It was proposed to compile from m source languages to n target machine languages by using m *front-ends* to compile to a common intermediate format and then using n *back-ends* to translate from that format to the target language. This way, $m+n$ tools are required instead of the $m*n$ tools for a direct translation from each source language to each target language. This approach turned out to be infeasible in general but to work well for restricted sets of source and target languages.

In Glanville's approach [18], machine-independent code selectors could be generated, based on shift-reduce parsing of source code statements with respect to an instruction-set grammar. Satisfactory results were reported for IBM 370 and PDP-11 machines, but instruction-level parallelism was not treated. Although relying on a well-defined formal background, the parser-based approach suffers from the ambiguity of grammars and therefore leads to suboptimal code selection on parallel machines.

Cattell [19] proposed a new target-independent code selection method ("maximal munching method"), based on heuristic tree covering. However, his machine description formalism ISP suffered from deficient readability, and neither captured parallelism. The survey by Ganapathi et al. [20] summarizes the techniques available in the early eighties and concludes with the demand of shorter compilation times and more versatile machine description languages. Retargetability was afterwards put into practice within the GNU project [21]: The GNU C compiler `gcc` could be successfully retargeted to a number of CISC and RISC machines, and is now widely spread in workstation and personal computer environments. In many cases, `gcc` outperforms commercial, processor-specific compilers. Unfortunately, `gcc` requires an exhaustive target machine description in a specific language, in turn including C constructs. Therefore, `gcc` does not permit frequent changes in the target architecture, as for instance required in customization of ASIPs. Furthermore, `gcc` has problems with DSPs. An attempt has been made to port `gcc` to Motorola's DSP56000, but the results are poor in terms of exploiting parallelism.

Retargetability with respect to code selection is provided by *code generator generators* (or *compiler-compilers*). Tools like BEG [22], Twig [23], and `iburg` [24] are capable of generating fast processor-specific tree pattern matchers from instruction-set descriptions given as *tree grammars*. In turn, these matchers can generate optimal covers for *data-flow trees*, i.e. subgraphs of *control/dataflow graphs* (CDFGs), by dynamic programming. The strength of those tools lie in fact, that also *complex instructions* can be handled. Moreover, runtime of the generated matchers is only linearly dependent on the tree size. Their cost metrics however inherently exclude instruction-level parallelism.

3.2. Microprogramming

Much input for work on embedded code generation also comes from the area of microprogramming, which traditionally uses more hardware-oriented machine models and also treats instruction-level parallelism as a central issue. In contrast to compiler construction, no separation is made between assembly-level and machine-level code generation. Most approaches to microcode generation employ *code compaction* algporithms: The general idea is to first translate source code into *vertical machine code*, consisting of separate, partially interdependent RTs. Afterwards, RTs are rearranged to form valid microinstructions. Unfortunately, being a resourceconstrained scheduling problem, already optimal local compaction (restricted to basic blocks) is NP-hard [25]. A number of heuristic local compaction techniques, which are extensively compared in [26], turned out to be useful in practice.

The first global compaction technique is Fisher's Trace Scheduling [27]. Trace Scheduling determines the critical path through a set of basic blocks with respect to given branching probabilities. Ignoring branches, this path is then treated as a "large" basic block which often reveals a large amount of parallelism. Then, a standard local compaction technique (*list scheduling*) is applied. However, Trace Scheduling tends to significantly increase the *code size* due to "compensation code" that needs to be inserted, so that it is not recommendable for embedded code generation. Percolation scheduling [28] is another well-known technique for global compaction, which has also been applied to high-level VLSI synthesis [29].

Many researchers have treated retargetable generation of microcode from high-level programming languages. Most approaches make use of special machine description languages for specification of instruction formats, binary encodings, available RTs, and conflicts between RTs due to encoding or resource limitations. In the MPG system [30], focus was on microcode generation for mainframes, including a complex mechanism for next-address generation for the control store. Vegdahl [31, 32] emphasized the necessity of phase coupling in microcode generation. His compiler postponed decisions regarding code selection (following Cattell's approach) to the compaction phase. Mueller and Varghese [33] proposed code generation based on a *graph model* of the target machine, instead of an instruction-set model. These early approaches however required much manual interaction by the user.

3.3. Embedded code generation

Although embedded processors also include CISCs, RISCs, and microcontrollers, most recent publications on embedded code generation focus only on standard DSPs and ASIPs. Emphasis is on high-quality code generation for irregular architectures and retargetability. Common to nearly all aproaches is the assumption of a micro-programmable controller.

Rimey and Hilfinger [34] introduced the concept of *data routing* in code generation for ASIPs: After operations are bound to functional units in the target processor, data routing deals with transporting data between functional units, so as to minimize the amount of transport operations. Greedy scheduling orders operations in time based on information about "routability" of data. In this way, scheduling and register allocation are closely coupled. Practical application is, however, restricted to a very simple class of target processors. The data routing approach was refined by Hartmann [35], who also stressed the main disadvantage of data routing, namely the possibility of deadlocks during scheduling. Consequently, a complex mechanism for deadlock avoidance was employed. Hartmann's technique was applied to one realistic example, for which very high code quality was achieved.

Wess [36] proposed the usage of *normal form programs* [37] for DSP code generation. Normal form programs are essentially sequences of "small" programs, each implementing computation of one expression tree. Optimal code selection on expression trees is possible in linear time, with respect to the number of selected instructions. In order to account for special-purpose registers, *trellis diagrams* are used. Trellis diagrams can be regarded as state diagrams representing possible operations on a target processor. An iterative code compaction method aims at exploitation of parallelism [38]. High-quality code generation is reported for standard DSPs (DSP56000, TMS320C2x, ADSP-2100). However, no mechanism is provided for constructing trellis diagrams from more common models.

Fauth's CBC compiler [39] makes use of Hartmann's scheduling and data routing techniques. The nML language permits concise, hierarchical instruction-set descriptions. CBC uses tree covering by dynamic programming for instruction selection, and also exploits user-defined transformation rules. However, nML is a rather tool-specific language. CBC has been applied to a realistic design at Siemens (Digital European Cordless Telephone, DECT), and has later been enhanced by global optimization techniques [40], e.g. chaining of operations beyond basic block boundaries, and heuristically generalizing code selection to *directed acyclic graphs* (DAGs) instead of trees.

The nML language and the data routing approach are adopted in IMEC's CHESS compiler [41, 42], which targets ASIPs with load-store architectures. So far, however, basically modelling results have been reported [43].

The CodeSyn compiler by Paulin et al. [44] maps C programs into machine code for industrial in-house ASIPs at BNR. Target processors are described by three separate items: the set of available instruction patterns, a graph model representing the datapath, and a resource classification that accounts for special-purpose registers. Code generation follows a more classical direction: code selection is performed with dynamic programming, however without exploiting available code generator generators. Register allocation is based on the user-specified resource classification and the left-edge algorithm [45]. A heuristic postpass compaction phase aims at exploiting parallelism. High code quality has been achieved, but results are reported only for few ASIPs, presumably due to the fact, that retargeting CodeSyn requires too much manual effort.

Research within the SPAM project focuses on code optimization for standard DSPs rather than on retargetability: Araujo and Malik partially integrate register allocation into code selection [46], thereby taking into account special-purpose registers in DSPs. A theoretical instruction-set criterion is developed, under which generation of spill-free schedules is possible in linear time. Due to usage of code generator generators, the approach is user-retargetable. Optimal code generation for data-flow trees is reported for the TMS320C25 DSP, however excluding consideration of parallelism, memory addressing, and mode registers. Liao and Wang et al. propose address assignment for DSP-specific address generation units (AGUs) as a means of advanced code optimization [47], and provide extensions of previous work by Bartley [48]. They also investigate improved code selection by DAG matching instead of tree matching and optimization of mode register usage [49]. High code quality is achieved, but the techniques are tailored only towards the

TMS320C25. Another promising technique is *delayed memory binding* of variables after code generation [50]: On standard DSPs with distributed memory banks (such as Motorola 56000), this technique significantly increases parallelism in a machine program.

Worth mentioning is also Mutation Scheduling (MS) by Nicolau et al. [51], which aims at code optimization by complete phase coupling. MS dynamically maintains a set of *mutations* for each value occuring in a source program. Mutations are essentially obtained via algebraic transformations. Driven by scheduling, MS selects an appropriate mutation for each value to be computed based on availability of resources. Apparently, traversing the search space for possible mutations is very crucial, and MS also demands for a large amount of potential parallelism in order to be effective. Although intended for ASIP code generation, promising results of MS so far have only been presented for idealized, homogeneous processor architectures, comprising a large number of registers and parallel functional units. In contrast, Schenk's compiler [52] focuses on realistic RISC architectures. Complete phase coupling is ensured by means of extensive *backtracking*, based on a fine-grain unified data structure (*resource usages*). Results are however not yet reported.

Specific contributions to embedded code generation have also been made by Wilson et al. [53] and Philips [54], whose techniques permit very high quality code generation for a narrow class of ASIPs, but are not sufficiently retargetable. The state-of-the-art in embedded code generation is summarized in [55].

The main conclusion that can be drawn from previous and related work is, that there is a strong trade-off between retargetability and code quality. The highest code quality is achieved in those approaches, which concentrate on very particular classes of target processors and thus do not provide much flexibility. On the other hand, HW/SW codesign for embedded systems demands for such flexibility in a *user-friendly* fashion. Retargetable compiler systems, which do provide a user-friendly interface by using an editable, external processor model, are still few. Processor models in those approaches are mostly *behavioral*, i.e. a pure instruction-set description is used. However, such models have problems with special-purpose registers and instruction-level parallelism. Therefore, compilers like CHESS and CodeSyn make use of *mixed models*, i.e. instruction-set models also including some structural information. The corresponding processor description formalisms are necessarily very tool-specific.

In contrast to all other approaches, the MSSQ compiler uses a *real hardware description language* for processor modelling and uses purely *structural* models. The next section describes the input format of MSSQ.

4. Description of source program and target processor

One main concept of MSSQ is the usage of the unified description language MIMOLA both for the target processor and the application to be compiled. An advantage of using a unified language is that no distortion exists between algorithm and hardware, for instance regarding the available data types. In MIMOLA only the data type "bit vector" is predefined. Complex data types may be defined by the user. In contrast to VHDL, the arithmetic interpretation (integer, unsigned) of bit vectors is encoded in the MIMOLA operators. A MIMOLA input for MSSQ consists of three sections: the algorithm to be compiled, the target processor model, and additional linkage and transformation rules. In the following we illustrate the language features using examples. A detailed description of MIMOLA can be found in [56].

4.1. Program specification

The algorithmic part of MIMOLA is essentially a superset of the PASCAL programming language, except for the data types SET, REAL, and FILE, which are not available. Extensions to PASCAL include

- **Predefined variable locations:** The statement
  ```
  VAR x :  (15:0) AT Reg1;
  ```
  declares a 16 bit variable `x` located at register `Reg1`.

- **References to physical storages:** Instead of using abstract variables, physical registers and memories can be directly referenced, e.g. in an assignment to the accumulator:
  ```
  ACCU := ACCU + M[1];
  ```

- **Bit-level addressing:** Subranges of operands may be referenced by appending a bit vector index range. The following assignment loads variable `x` with the least significant 16 bits of register `ACCU`:
  ```
  x := ACCU.(15:0);
  ```

- **Module calls:** Hardware components can be called like procedures with parameters, so as to enforce execution of certain operations. For instance, if the processor description contains a component named `AdderComp`, this component can be "called" in an assignment:
  ```
  x := AdderComp(y,z);
  ```

- **Operator binding:** Operations can be bound to certain hardware components, e.g. in the assignment
  ```
  x := y +_AdderComp z;
  ```
  the addition is bound to component `AdderComp`.

Since all these extension are optional, the user can select from a variety of "programming styles", either more abstract or more hardware-specific. Pure PASCAL programs are possible as well as programs close to the assembly level.

4.2. Processor modeling

MIMOLA permits modeling of arbitrary (programmable or non-programmable) hardware structures. Similar to VHDL, a number of predefined, primitive operators exists. The basic entities of MIMOLA hardware models are *modules* and *connections*. Each module is specified by its port interface and its behavior, and modules together with connections form a netlist. The following example shows the description of a multi-functional ALU module:

```
MODULE Alu (IN i1, i2: (15:0); OUT outp: (15:0);
            IN  ctr: (1:0));
CONBEGIN
 outp <- CASE ctr OF
           0: i1 + i2;
           1: i1 - i2;
           2: i1 AND i2;
           3: i1;
         END;
CONEND;
```

The CONBEGIN/CONEND construct includes a set of concurrent assignments. In the example a conditional assignment to output port `outp` is specified, which depends on the two-bit control input `ctr`. Also don't care conditions with respect to control inputs can be specified by means of "`X`" CASE tags. The next example shows a model of a register module.

```
MODULE ACCU (IN inp: (15:0); OUT outp: (15:0);
             IN enable: Bit; CLK clock: Bit));
VAR reg: (15:0);
CONBEGIN
 CASE enable OF
  0: NOLOAD;
  1: AT clock UP DO reg := inp;
 END;
 outp <- reg;
CONEND;
```

Variable `reg` of module `ACCU` stores a new input value at the rising clock edge if the enable signal has the value one. Concurrently, the register value is assigned to output port `outp`. The predefined `NOLOAD` statement specifies a "no operation" mode.

Connections between module ports are defined by `CONNECTION` statements, e.g.

```
CONNECTIONS      Alu.outp     ->   ACCU.inp;
                 ACCU.outp    ->   Alu.i1;
```

MSSQ also has a notion of bidirectional tristate busses, which are declared by `BUS` statements. In presence of tristate busses, MSSQ assumes that in all modules driving a bus can be set to a "`TRISTATE`" mode by approriate control signals.

4.3. Linkage and transformation rules

In case of programmable hardware structures, two distinguished storage locations exist from a compiler's point of view: the program counter and the instruction memory. A code generator operating on netlists can hardly identify these locations only by inspecting the structure. Therefore, these two locations (which are part of the netlist model) need to be explicitly labelled. This is done by `LOCATION` statements:

```
LOCATION_FOR_PROGRAMCOUNTER PCReg;
LOCATION_FOR_INSTRUCTIONS IM[0..1023];
```

Other `LOCATION` statements may be optionally used for restricting the register allocation search space, i.e. storages can be identified as physical locations for declared user variables or for intermediate results.

An important feature of MSSQ is the *replacement mechanism*. This mechanism works through user-defined rewrite rules, which can be used in two ways: for implementing operations in the algorithm, which are not available in the target processor, and for increasing the degrees of freedom for code generation. if the resulting code is more favorable. Replacement rules may comprise arbitrary MIMOLA expressions with formal parameters. The following example shows two replacement rules.

```
REPLACE_ALWAYS &a * 2 WITH &a + &a;
REPLACE_CONDITIONALLY &a + 1 WITH "INCR" &a;
```

Rules with an `ALWAYS` attribute are unconditionally applied already during intermediate code generation. In the example, multiplication of a formal parameter `&a` by 2 is replaced by an addition `&a + &a`. This rule permits code generation for `&a * 2`, even if the target processor does not contain a multiplier. Rules with a `CONDITIONALLY` attribute are applied on demand, e.g. if an addition `&a + 1` is required, the compiler *may* decide to bind an addition of value 1 to an incrementer if the resulting code is more favorable. By means of the replacement mechanism, a high degree of flexibility is offered by the MSSQ frontend. As indicated by the above examples, typical applications of replacement rules include implementation of operators which are unimplemented in hardware and strength reduction of arithmetic operators. Furthermore, replacement rules permit to cope with unforeseen idiosyncrasies of new target processors.

A complete MIMOLA description of a very simple 8-bit processor is given in fig. 3. The corresponding schematic is shown in fig. 4.

```
MODULE SimpleProcessor (IN inp:(7:0); OUT outp:(7:0));
STRUCTURE IS                   -- outermost module is a structural one
TYPE InstrFormat = FIELDS      -- 21-bit horizontal instruction word
                    imm:       (20:13);
                    RAMadr:     (12:5);
                    RAMctr:        (4);
                    mux:         (3:2);
                    alu:         (1:0);
                  END;
      Byte = (7:0); Bit = (0);  -- scalar types

PARTS                           -- instantiate behavioral modules
 IM: MODULE InstrROM (IN adr: Byte; OUT ins: InstrFormat);
      VAR storage: ARRAY[0..255] OF InstrFormat;
      CONBEGIN ins <- storage[adr]; CONEND;
 PC, REG: MODULE Reg8bit (IN data: Byte; OUT outp: Byte);
           VAR R: Byte;
           CONBEGIN R := data; outp <- R; CONEND;
 PCIncr: MODULE IncrementByte (IN data: Byte; OUT inc: Byte);
          CONBEGIN outp <- INCR data; CONEND;
 RAM: MODULE Memory (IN data, adr: Byte; OUT outp: Byte; FCT c: Bit);
       VAR storage: ARRAY[0..255] OF Byte;
       CONBEGIN
        CASE c OF: 0: NOLOAD storage; 1: storage[adr] := data; END;
        outp <- storage[adr];
       CONEND;
 ALU: MODULE AddSub (IN d0, d1: Byte; OUT outp: Byte; FCT c: (1:0));
       CONBEGIN                  -- "%" denotes binary numbers
        outp <- CASE c OF %00: d0 + d1; %01: d0 - d1; %1x: d0; END;
       CONEND;
 MUX: MODULE Mux3x8 (IN d0,d1,d2: Byte; OUT outp: Byte; FCT c: (1:0));
       CONBEGIN outp <- CASE c OF 0: d0;  1: d1; ELSE: d2; END; CONEND;

CONNECTIONS
 -- controller:                    -- data path:
 PC.outp       -> IM.adr;          IM.ins.imm    -> MUX.d0;
 PC.outp       -> PCIncr.data;     inp       -> MUX.d1;   -- primary input
 PCIncr.outp   -> PC.data;         RAM.outp  -> MUX.d2;
 IM.ins.RAMadr -> RAM.adr;         MUX.outp  -> ALU.d1;
 IM.ins.RAMctr -> RAM.c;           ALU.outp  -> REG.data;
 IM.ins.alu    -> ALU.c;           REG.outp  -> ALU.d0;
 IM.ins.mux    -> MUX.c;           REG.outp  -> outp;     -- primary output
END; -- STRUCTURE
LOCATION_FOR_PROGRAMCOUNTER PC;
LOCATION_FOR_INSTRUCTIONS IM;
END; -- STRUCTURE
```

Figure 3. Complete MIMOLA description of a simple 8-bit processor

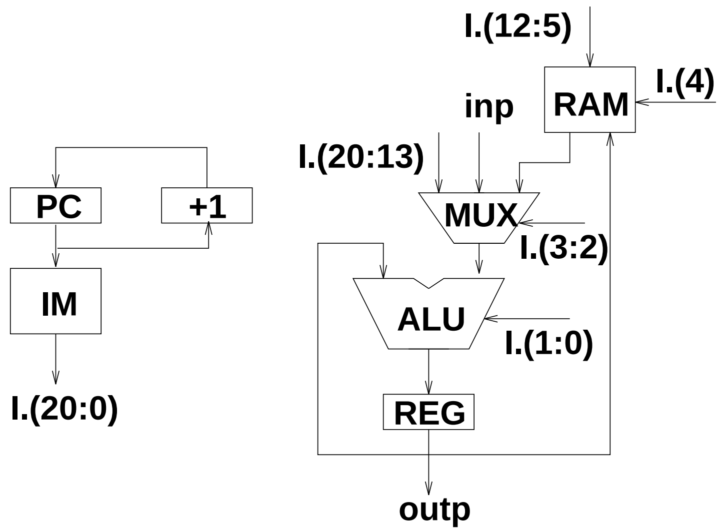


Figure 4. Schematic of the example 8-bit processor

5. Internal data structures

MSSQ uses two main internal data structures, which are described in this section. The *connection operation graph* (COG) represents the RT-level hardware structure and capabilities of RT-level functional units. The COG is used for pattern matching between the intermediate program representation and primitive target processor operations. The necessary control codes for RT-level components involved in a primitive processor operation are maintained by means of *instruction trees (I-trees)*.

5.1. Connection operation graph

The COG is MSSQ's internal representation of the target processor. The COG *nodes* represent hardware operators and module ports, while *directed edges* represent dataflow between nodes. The direction of edges is opposite to the dataflow in the hardware.

Fig. 5 shows two connected RTL modules in MIMOLA, as well as the corresponding partial COG: Operation `dat` denotes a *transparent mode*, i.e. an identity operation on the input data. Module `Mux` can perform two `dat` operations, either on input `d0` or `d1`. The edges to port `c` reflect that selection of a certain operation depends on the value assigned to that control port. The output `d` of `Mux` ("`Mux.d`") is connected to input port `i1` of module `AddSub`. This module performs either addition or subtraction on `i1` and `i2`, depending on the value of control signal `ctr`, and assigns the result to output port `p`.

```
MODULE Mux (IN d0,d1:(7:0);
                   FCT c:Bit; OUT d:(7:0));
 CONBEGIN
  d <- CASE c OF  0: d0; 1: d1;
 CONEND;

MODULE AddSub (IN i1,i2:(7:0);
                   FCT ctr: Bit;
                   OUT p:(7:0));
 CONBEGIN
  p <- CASE ctr OF
         0: i1 + i2;
         1: i1 - i2;
        END;
 CONEND;

CONNECTIONS  Mux.d  -> AddSub.i1;
```

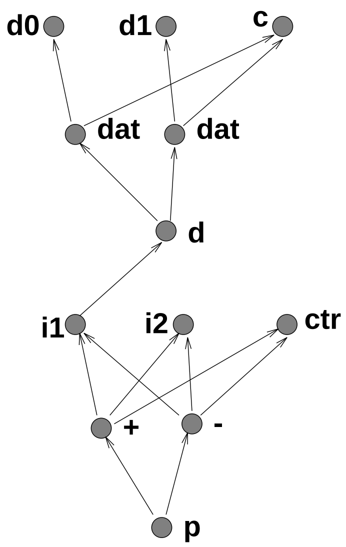


Figure 5. Partial connection operation graph for two MIMOLA modules

In this way, the complete target processor structure is represented by a COG. For variables in sequential modules, separate *read* and *write* operation nodes are present in the COG. The direction of COG edges permits to trace back the dataflow from module output ports to input ports, and further to other module output ports.

In a preprocessing step, the COG is constructed from the textual MIMOLA processor model. The complexity of COG construction linearly depends on the number of hardware operators in the target processor. Even for complex target processors (e.g. TMS320C25), COG construction takes less than one minute of CPU time on a SPARC-20. COG nodes representing hardware operations are annotated with the corresponding settings of module control ports. These settings can be directly derived from `CASE` statements in module bodies. For example, consider again the partial COG shown in fig. 5. The `dat` operation on input port `d0` of module `Mux` is selected by setting control port `Mux.c` to zero, so that "0" would be annotated. Similarly, "1" would be annotated for the "–" node for module `AddSub`. During preprocessing, MSSQ also checks for *reachability* of module control ports: A control port is *directly* reachable, if a direct connection to the instruction memory exists. If a control port is indirectly connected to the instruction memory via other combinational modules, whose control ports are reachable in turn, then the control port is called *indirectly* reachable. Indirect reachability is, for instance, important in presence of instruction decoders in the processor model.

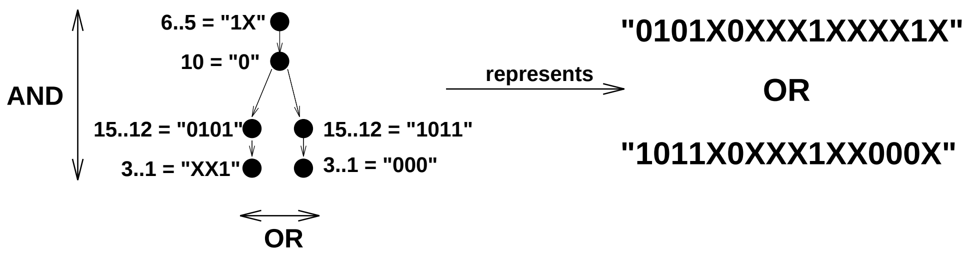


Figure 6. I-tree representing two alternative versions

5.2. I-trees

MSSQ assumes all module control signals to originate at the labelled instruction memory, either directly or indirectly via decoders. Therefore, all control signals can be represented by *partial instructions* or *versions*. For a target processor with instruction wordlength W, a version is a bitstring $B \in \{0, 1, X\}^W$. For processors with single-cycle operations, partial instructions are sufficient for checking both for encoding and resource conflicts: Due to explicit modelling of multiplexers and busses, all resource conflicts are reflected in instruction conflicts. MSSQ uses *I-trees* for efficiently and dynamically maintaining *sets of versions* and checking for instruction conflicts. I-trees are constructed on-the-fly during code generation. They are associated with register transfers and represent the necessary partial instructions and possible alternatives.

I-tree nodes contain an *instruction field*, i.e. an instruction bit index subrange and a bitstring over $\{0, 1, X\}$. The relative position of nodes decides on the relation between instruction fields: Nodes on the same path in the tree are related by "AND", i.e. they must be *simultaneously* valid. Nodes on different paths are related by "OR", i.e. different paths represent *alternatives*. The set of versions represented by an I-tree is obtained by separately traversing all paths from the root to the leaves. Instruction fields not contained on a path are assumed to be don't care. For each path, the instruction fields in the nodes are bitwise combined by operation "$*$" defined as follows:

$*$	0	1	x
0	0	E	0
1	E	1	1
x	0	1	x

The value E represents an *error*, that is, the fields cannot be simultaneously valid. In this case, the fields are called *incompatible* or *conflicting*. An example I-tree for a 16-bit instruction word length with its interpretation is given in fig. 6.
I-trees are constructed by means of three operations:

SET: instruction field $\mapsto$ I-tree: SET(B) constructs a single-node I-tree representing version B.

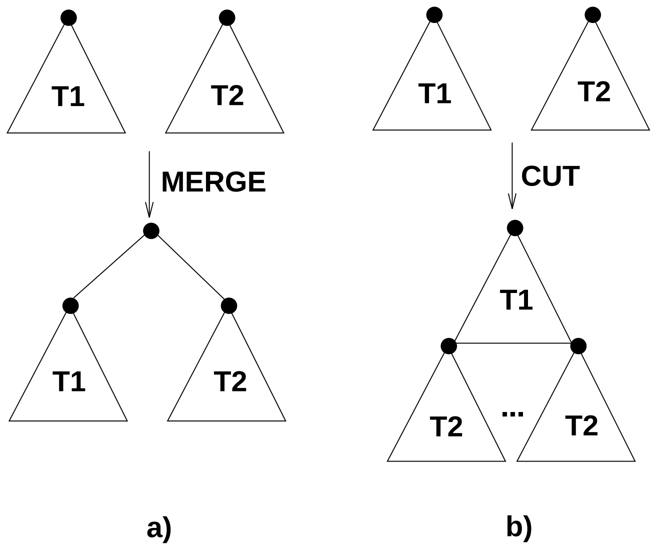


Figure 7. Operations MERGE and CUT on I-trees

MERGE: I-tree $\times$ I-tree $\mapsto$ I-tree: MERGE(T_1, T_2) constructs an I-tree T_3, which represents the union of the version sets of T_1 and T_2. This is performed by attaching T_1 and T_2 to a common root (fig. 7 a). The MERGE operation is used, whenever alternative versions for a register transfer are detected during code generation.

CUT: I-tree $\times$ I-tree $\mapsto$ I-tree: CUT(T_1, T_2) constructs an I-tree T_3, which represents the intersection of the version sets of T_1 and T_2. This is performed by appending T_2 to all leaves of T_1, and checking for conflicts on each path (fig. 7 b). CUT may yield a void I-tree in case that all versions represented by T_1 and T_2 are pairwise conflicting. The CUT operation is used in code generation, whenever different partial instructions must be simultaneously set in order to activate a certain register transfer.

6. Code generation

This section describes how the source algorithm is mapped to the target processor using the above data structures. Code generation proceeds in three sequential phases. The first phase transforms the high-level source program into an RT-level program. The second phase integrates code selection and register allocation, while the third phase heuristically exploits potential instruction-level parallelism.

6.1. Preprocessing

Before code generation takes place, the high-level source program is transformed into an RT-level program. All declared user variables are bound to storage modules, and variable references are substituted by references to the corresponding storages. The process of variable binding may be steered through reservations provided by the user. Otherwise, variables are bound to a arbitrary storage modules of sufficient capacity. Furthermore, all high-level control structures like FOR, WHILE, and REPEAT loops are replaced by IF-constructs, with explicit reference to the labelled program counter register. The following example shows a piece of source code and the corresponding RTL program:

```
source code:

VAR x, y, z: integer;
REPEAT
  y := y + z;
  x := x - 4;
UNTIL x < 0;

RTL code: (let x, y, z be bound to Mem[0], Mem[1], Mem[2])

lab:
     Mem[1] := Mem[1] + Mem[2];
     Mem[0] := Mem[0] - 4;
     PC := (IF Mem[0] >= 0 THEN lab ELSE INCR PC);
```

The REPEAT/UNTIL loop is replaced by a conditional assignment to the program counter `PC`. If `Mem[0] >= 0` is true at the end of the loop body, then the branch to label `lab` is taken. Otherwise, `PC` is incremented so as to point to the next instruction after the loop. Replacement rules for high-level control structures are contained in an external library, which can be edited by the user. In this way, the most appropriate replacements can be selected for each particular target processor. On a DSP for instance, it might be favorable to replace FOR-loops by hardware loops.

After source code has bee transformed down to the RT level, the program consists of a sequence of (possibly conditional) assignments to storage locations, and thereby is prepared to be mapped to the hardware. IF-statements or expressions are the only remaining high-level control structures. In hardware, IF-constructs correspond to multiplexers.

6.2. Code selection and register allocation

The next phase is responsible for selection of instruction patterns which implement the desired behavior. Simultaneously, registers for intermediate results are allocated as required. The basic idea is to perform pattern matching between RTL assignments and COG subgraphs. The necessary control signals are dynamically maintained by means of I-trees. Code selection and register allocation are performed by two interacting modules: the pattern matcher and the temporary cell allocator.

6.2.1. Pattern matching

Single assignments can be represented by trees. For each of those trees, pattern matching is performed separately. MSSQ tries to find a subgraph in the COG, which matches the tree representation of the assignment. This is restricted to single-cycle operations, i.e. read and write nodes in the COG are not crossed during pattern matching. However, pattern matching does exploit transparent modes of combinational modules. Pattern matching stops as soon as the first matching subgraph is found. This subgraph corresponds to one register transfer. Then, MSSQ generates all module control signals required for that RT: The COG provides information about the paths between the designated instruction memory and each RT-level component. For each control port of an RT-level component involved in the selected RT, the annotated setting is adjusted by tracing back the path from that port to the instruction memory, and generating the appropriate partial instructions, say $i_1, \ldots, i_n$. Since $i_1, \ldots, i_n$ are simultaneously required to execute the selected register transfer, the corresponding I-tree is computed by $n-1$ consecutive CUT operations:

$$i \quad = \quad i_1 \quad CUT \quad \ldots \quad CUT \quad i_n$$

If a CUT operation yields a void I-tree due to conflicting settings, pattern matching is iterated, until a subgraph leading to a non-void I-tree is found. If alternative control port settings for a certain subgraph node exist, all alternative versions are kept in the I-tree by using the MERGE operation. Since all alternative versions correspond to a single-cycle operation, there is no cost function for versions, but all versions have unit cost. The most appropriate version is selected only during code compaction, dependent on the context of potentially parallel register transfers. MSSQ also exploits commutativity and neutral elements of arithmetic operations in order to detect more alternative versions. The result of pattern matching for one RTL assignment is an RT, as well as an I-tree representing all its alternative partial instructions. Fig. 8 illustrates pattern matching for an example assignment.

The partial structure in fig. 8 b) consists of six modules: instruction memory `I`, memory `M`, decoder `DEC` which computes powers of two, multiplexer `MUX`, an ALU comprising a subtraction mode, and register `R`. Each module has attached the instruction index subrange that controls the module, e.g. `DEC` is steered by instruction bits 5..3, and `M` is addressed by instruction bits 13..6. Pattern matching between the assignment tree in fig. 8 a) with the partial structure delivers two versions for the assignment. The difference lies in the control signal for `MUX` and the way of generating the constant 64. In the first version, `MUX` passes its left input, and the constant is immediately allocated at the instruction bits 13..6. The second version exploits `DEC` for constant generation and `MUX` passes the right input. Assuming that instruction bits 2 and 1 enable register `R` and select subtraction on the ALU, respectively, fig. 8 c) shows the resulting I-tree. However, during I-tree construction, a conflict is detected regarding instruction bits 13..6: this instruction field cannot be simultaneously used for generating the memory address and the constant in this case. Therefore, the first version is discarded.

6.2.2. Temporary cell allocation

Assignments may contain complex expressions which cannot be computed within a single machine cycle. In this case, storage allocation for intermediate results is required. Whereas binding of declared user variables to storage locations in MSSQ is done before code generation (through linkage specification), temporary allocation is done on-the-fly during code generation. Temporary cells may be storage cells or datapath registers. Temporary allocation is activated, if the above pattern matching process fails. In this case, the assignment is split into a sequence of two simpler assignments, for which the pattern matcher is called recursively. Suppose, the assignment

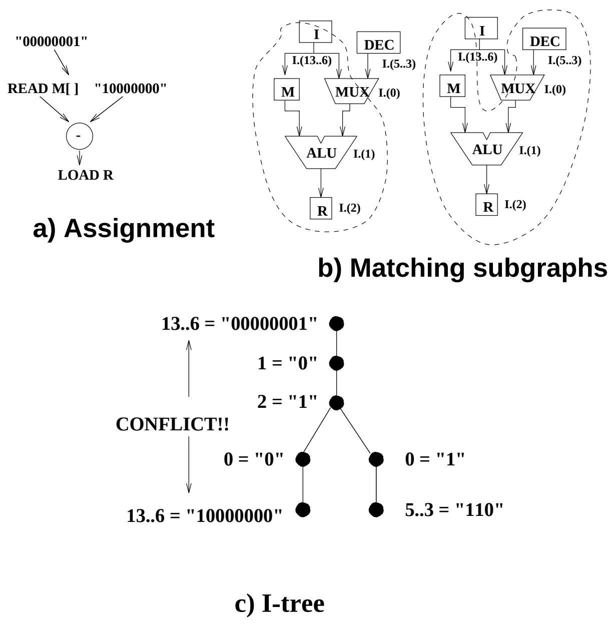


Figure 8. Pattern matching between assignment trees and hardware structure

```
R := RAM[0] - RAM[2];
```

is to be allocated, but the target processor permits only one RAM access per cycle. Then, (using temporary cell TMP) the assignment can be be split into the sequence

```
TMP := RAM[2];
R := RAM[0] - TMP;
```

for each of which the pattern matcher is invoked recursively. Recursion terminates, when a sequence has been successfully allocated, or the allocation process has conclusively failed, e.g. in case of insufficient hardware resources. In the latter case, a detailed error message reporting the failure reason is emitted, so that the user may accordingly correct either the source code or the hardware description.

Basically, MSSQ allocates temporary cells in a greedy fashion. This results in a worst-case runtime exponential in the number of recursion steps during temporary allocation, i.e. the complexity of assignments to be compiled. The possibly vast search space is pruned by obeying the following rules:

- Allocation of temporary cells for intermediate results is goal-directed in the sense, that the information gained during pattern matching is exploited. Temporary cells are generated for that subexpression of a complex expression, for which the pattern matcher reported a failure.
- Only those locations are considered, which have been specified by the user in the linkage section of the processor description. All specified locations are tried before an additional recursion stage in pattern matching is initiated. This prevents temporary allocation from generating costly additional control steps, which could be avoided by a different temporary allocation.
- The bitwidth of possible temporary locations must be equal or greater to the required temporary result bitwidth. In the latter case, MSSQ performs sign, zero, or don't care extension.
- MSSQ has a notion of *distances* between storage locations. Distance is defined as the number of clock cycles required to tranfer data from one location to another, possibly via combinational modules. Only those temporary locations are considered, whose distances to the destination of an assignment do not exceed 2. Fig. 9 shows an example.

Although the latter strategy inhibits successful code generation in some special cases (e.g. if more than three registers need to be passed for a data transfer), it turns out to be a good compromise, since without such a general restriction the search time can become unacceptable.

During temporary allocation MSSQ keeps track of storage contents. Common subexpressions in complex expressions are identified, and allocated only once. However, common subexpression analysis does not cross assignment boundaries. The result of code selection and register allocation in general is a sequence of register

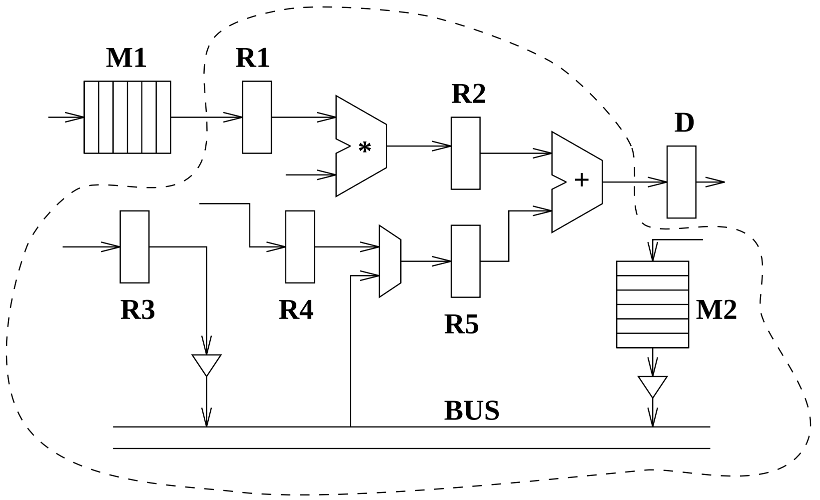


Figure 9. Temporary search space for destination register D

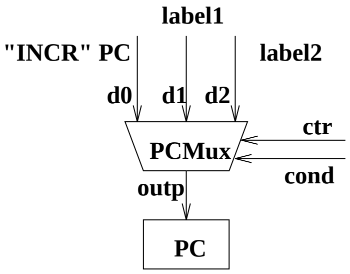
transfers, each executable within a single machine cycle. Each of these RTs has an associated I-tree representing necessary partial instructions and possible alternatives. Note that code selection and register allocation do not imply any decision concerning version selection and control step binding.

6.3. Control flow allocation

So far, only allocation of "dataflow-related" assignments in MSSQ has been described. In contrast to other approaches, MSSQ derives possible control-flow operations, i.e. modifications of the program counter register `PC`, directly from the hardware structure. Using the above pattern matching mechanism, MSSQ tries to generate versions for the following assignments to `PC` in advance:

- `PC := "INCR" PC;` Increment program counter
- `PC := label;` Jump to a symbolic label
- `PC := (IF cond THEN "INCR" PC ELSE label);`
  Conditional ELSE-branch
- `PC := (IF cond THEN label ELSE "INCR" PC);`
  Conditional THEN-branch
- `PC := (IF cond THEN label1 ELSE label2);` Two-way branch

Whether or not versions for all these assignments exist depends on the branch logic of the current target processor. A partial controller structure permitting all five `PC`




```
MODULE PCMux (IN d0,d1,d2: pcwidth;
              IN cond: Bit; ctr:(2:0);
              OUT outp: pcwidth);
CONBEGIN
 outp <- CASE ctr OF
        0: d0;  1: d1;  2: d2;
        3: IF cond THEN d0 ELSE d1;
        4: IF cond THEN d1 ELSE d0;
        5: IF cond THEN d1 ELSE d2;
       END;
CONEND;
```

Figure 10. A "universal" branch logic

assignment types is depicted in fig. 10 together with a possible MIMOLA model of the PC multiplexer `PCMux`.

MSSQ tries the latter three assignment types for implementation of conditional assignments via conditional branch constructs. The symbolic labels are later replaced by instruction memory addresses.

There exist several other methods for implementing conditional assignments in hardware besides conditional branches [57]. One of these (*conditional load*) is implemented in MSSQ. An implementation by "conditional load" is available, when the assignment condition can be routed to the enable input of the destination. In this case, no `PC` modification is necessary. In turn, this increases the freedom for code selection. If available, MSSQ allocates conditional jump as well as conditional load versions for IF-statements, and both are kept for the compaction phase. Like all other RT operations, also assignments to `PC` are associated with a corresponding I-tree.

6.4. Compaction and version selection

The compaction phase aims at exploiting potential instruction-level parallelism, while simultaneously selecting the most appropriate binary encoding versions. Compaction operates on the output from code selection and register allocation, i.e. a sequence of RTs, each with an I-tree representing all alternative versions. Heuristically, these RTs are assigned to control steps while obeying *dependencies* between RTs. If RT_1 is dependent on RT_2, e.g. RT_1 reads a register value by RT_2, then RT_2 must be scheduled earlier than RT_1. Pairwise independent RTs may be scheduled in parallel, if they are compatible, i.e. no conflicts exist with respect to their partial instructions. Compatibility can be efficiently checked by means of the CUT operation on I-trees.

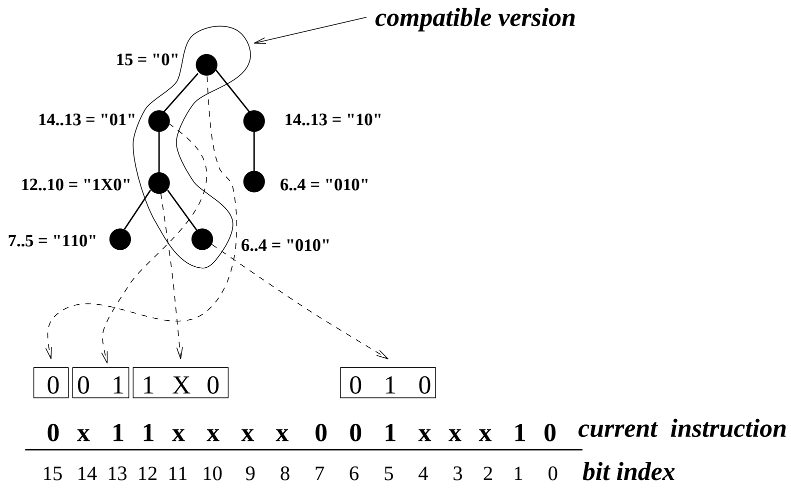


Figure 11. Version selection from I-trees

MSSQ employs a modified version of the "first-come-first-serve" (FCFS) heuristic [26]: The microinstructions are generated step-by-step, starting with the "last" control step. For each control step, first a "ready" RT is arbitrarily selected. An RT is ready, if all its successors with respect to the dependency relations have already been processed. The selected RT is packed into the current microinstruction, and an arbitrary version is selected from its I-tree. Then, all other ready RTs are investigated sequentially. For each of these, MSSQ checks whether a compatible version exists, so that the RT can also be packed into the current microinstruction. Selection of compatible versions from I-trees is illustrated in fig. 11.

This is iterated until no further RT can be packed. Then, the microinstruction is finished, and the next one is generated in the same way. Compaction terminates when all RTs have been packed into a microinstruction. The complexity of this compaction heuristic is $O(n^2)$ for n register transfers.

When "finishing" a microinstruction, MSSQ inserts additional partial instructions in order to avoid *undesired side effects.* Such side effects potentially arise from two sources:

Unused storages: In each microinstruction, a certain set of sequential modules (registers and memories) are not written. As shown for the example module `ACCU` (see section 4.2), sequential modules typically use distinguished control ports for enabling write operations on their variables. Whenever such a module variable contains a live value, but is not written in a certain control step, it must be ensured that its value is retained, which corresponds to activation of the predefined MIMOLA operation `NOLOAD`. Similar to other hardware operations, `NOLOAD`s are associated with partial instructions. Since MSSQ does not perform book-keeping of live registers across RTs, it uses a conservative

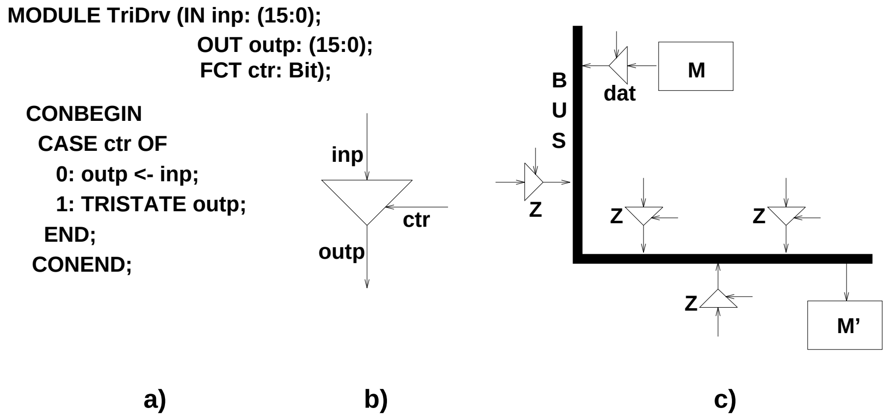


Figure 12. Tristate buses: a) tristate driver in MIMOLA, b) driver symbol, c) tristate bus with multiple drivers

approach: A `NOLOAD` operation is "scheduled" for each unused module variable in each microinstruction.

Tristate buses: All modules that can write to a bidirectional bus must provide a `TRISTATE` operation. In the simplest case, such modules are *tristate drivers* as shown in fig. 12 a). These modules ensure, that bus conflicts can be avoided by disconnecting the driver from the bus. If data is to be transmitted from module M to module M' via a bus (fig. 12 c), then exactly one driver performs a `dat` operation, while all unused drivers perform a `TRISTATE` (denoted by "Z"). Similar to `NOLOAD`s, MSSQ packs `TRISTATE` operations for all unused bus drivers into each microinstruction.

The integrated compaction and version selection phase concludes code generation in MSSQ. Compaction is called for each RT sequence generated by code selection and register allocation. Concatenating all machine instructions generated by compaction yields the complete machine program for the source algorithm. Binding of instructions to instruction memory addresses and insertion of jump addresses into the binary code are performed during a postpass phase. Fig. 13 illustrates the overall structure of the code generation process.

7. Results

Besides requiring a microprogrammable controller, MSSQ is not dedicated to a certain processor family. Therefore, MSSQ could be successfully retargeted to a variety of different machines. Table 1 lists target processors, for which MSSQ has generated code so far, and the corresponding MIMOLA model sizes.

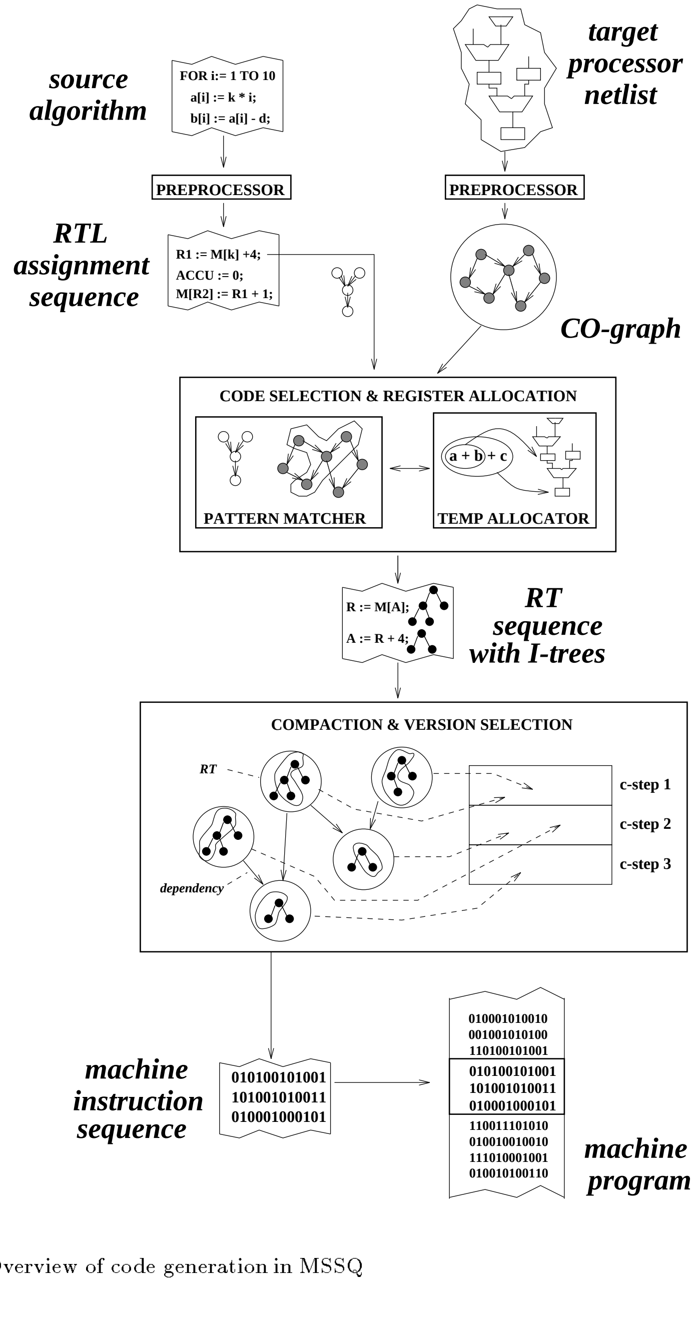


Figure 13. Overview of code generation in MSSQ

Table 1. Target machines for MSSQ and MIMOLA model sizes

machine	type	# RTL modules	model size (lines)
AMD 29203	bit-slice processor	11	533
SAMP	self-timed VLIW machine	36	624
PRIPS	VLIW PROLOG processor	38	1255
BassBoost	ASIP	20	773
TMS320C25	digital signal processor	94	1889
DSP56156	digital signal processor	116	1439

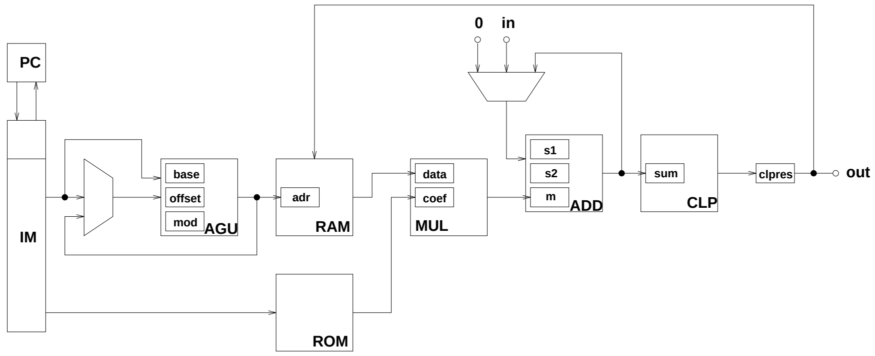


Figure 14. Industrial ASIP for digital audio signal processing

Detailed descriptions of the AMD 29203, SAMP, and PRIPS architectures, as well as code generation results for these machines are given in [58, 59, 60, 61, 62]. Here, we focus on the latter three architectures in table 1, which are more representative for embedded processors.

7.1. Bass boost ASIP

The architecture depicted in fig. 14 is an industrial ASIP, which is used in the area of digital audio signal processing. The ASIP has a moderately encoded 41-bit microinstruction format, a dedicated address generation unit (AGU) supporting ring buffers, and a 120×18 bit RAM. Arithmetic computations are performed on a 24-bit multiply-accumulate section. Filter coefficients are stored in a ROM. All functional units can work in parallel in order to guarantee high throughput.

We used MSSQ to map a *digital bass boost* (DBB) algorithm into machine code for the ASIP. The DBB consists of two identical stereo channels, each realizing a *low-pass filter*. Since MSSQ does not provide particular support for ring buffers, we traded manual description effort against code quality by using three different MIMOLA descriptions of the DBB source algorithm:

Table 2. Results for digital bass boost ASIP

programming style	# source code statements	# machine instructions	CPU seconds (SPARC-20)
high level	84	197	602
medium level	98	98	200
low level	329	64	7

High level: In the first program, the DBB algorithm is described at the pure PASCAL level, without any particular reference to the underlying hardware. Ring buffers are replaced by successive data moves at the end of each sample period. Code selection, register allocation, and compaction are completely left to the compiler.

Medium level: In the second program, storage layout is manually predefined by storage binding of variables. This also includes explicit storage allocation for ring buffers and access to ring buffer elements by means of the modulo addressing capabilities of the AGU in order to avoid data moves. Code selection, register allocation, and compaction are left to the compiler.

Low level: In the third program, very few freedom is left to the compiler. The MIMOLA program already closely reflects the machine program by predefining a sequence of control steps. Variables are completely replaced by storage and register references. The compiler is only responsible for binding operators to hardware resources and translating the RTs into binary machine instructions.

Each of these programs were separately compiled by MSSQ. The results are shown in table 2. The high-level program required few description effort, but led to inferior code quality, essentially due to insufficient AGU utilization. This problem was avoided in the medium-level program, for which a 50 % reduction in the number of generated instructions was obtained. The highest code quality was achieved for the low-level program. Through extensive manual analysis of hardware capabilities, a very high resource utilization is ensured, resulting in only 64 machine instructions. This is the same code quality as it was obtained originally by completely manual code generation. The effort for writing the low-level program was, however, comparatively high.

For sake of completeness, table 2 also shows the compilation times, which are relatively high for the first two program versions. Nevertheless, in contrast to compilers for general-purpose systems, *compilation speed* plays only a secondary role in embedded code generation. While the user of a C compiler on a workstation is hardly willing to spend more than a few minutes in compiling thousands of source code lines, even compilation times of several hours may be regarded as being acceptable for embedded code generation, because the demands on compilers are much higher in this area. In fact, most publications on embedded code generation

Table 3. Results for TMS320C25

program	# source code statements	# machine instructions	CPU seconds (SPARC-20)
test1	7	22	53
test2	17	49	77
elliptical wave filter	74	184	186
greatest common divisor	8	18	20
PID control	34	75	74
differential equation solver	59	99	117

do not explicitly mention compilation speed, as long as an acceptable amount of CPU time is not exceeded.

7.2. TMS320C25

The Texas Instruments TMS320C25 is a popular standard DSP [63] with a strongly encoded 16-bit instruction format and a moderate amount of instruction-level parallelism. We have used MSSQ to compile six high-level programs into TMS320C25 code. These include arithmetic test programs as well as realistic DSP algorithms. The results are listed in table 3.

Analysis of the generated code yields an estimated overhead of 50 - 100 % compared to hand-crafted code. This overhead is mainly due to the fact, that MSSQ maps only statement-by-statement, but does not perform dataflow analysis between statements. This sometimes results in redundant data move instructions. The statement-wise mapping mechanism also prevents MSSQ from sufficiently exploiting instruction-level parallelism on the TMS320C25, e.g. in form of multiply-accumulate instructions. The quality of the generated code is however not too bad, if compared to results obtained by processor-specific compilers. A recent experimental study [10] revealed that current commercial compilers for standard DSPs yield code with similar overheads in quality.

7.3. DSP56156

The DSP56156 [64] is a member of the widespread Motorola 56000 family of digital signal processors. In contrast to the TMS320C25, it as a moderately encoded 24-bit instruction word, with a significant amount of instruction-level parallelism. We have used MSSQ to compile the same set of benchmark programs onto this processors as for the TMS320C25. Table 4 shows the results.

Except for the *elliptical wave filter* program, the number of generated instructions is comparable to the results achieved for the TMS320C25. The code quality is however worse, because the parallel memory access capabilities of the DSP56156 are not exploited. This is due to the fact, that binding of variables to memory

Table 4. Results for DSP56156

program	# source code statements	# machine instructions	CPU seconds (SPARC-20)	# instructions GNU gcc
test1	7	24	48	45
test2	17	44	59	55
elliptical wave filter	74	133	224	290
greatest common divisor	8	20	36	30
PID control	34	90	67	113
differential equation solver	59	93	58	202

banks in MSSQ takes place before code generation. For architectures with parallel memory banks, such as the DSP56156, delayed binding of variables, as proposed in [50] yields much higher utilization of parallelism.

For comparison purposes, we have also applied the DSP56000 version of the GNU C compiler `gcc` to the above benchmark programs. The results (optimizations enabled) are shown in the rightmost column of table 4. Naturally, `gcc` compiles much faster and only needs fractions of a CPU second for each program. The code quality is however very poor. Instruction-level parallelism is not exploited at all, and insufficient utilization of special-purpose registers leads to overheads of more than 100 % compared to MSSQ-generated code. Furthermore, MSSQ requires much smaller effort for processor modelling: The MIMOLA model of the DSP56156 consists of approximately 1400 lines, while the GNU model comprises more than 7700 lines.

8. Conclusions

Design automation for HW/SW codesign of embedded systems demands for flexible code generators as an interface between software synthesis and embedded processors. Retargetable compilers provide a promising solution, if different target processors need to be investigated during HW/SW partitioning. Furthermore, retargetable compilers are necessary for single-chip designs comprising ASIPs, for which compiler support is hardly available so far.

In this contribution, we have presented the MSSQ compiler. In contrast to other approaches to retargetable code generation for embedded processors, MSSQ operates on purely structural RT-level processor descriptions. Probably the most important result of our work with MSSQ is the fact, that the compiler could be successfully retargeted to a large variety of different real-life machines. In contrast to related work, for which practical results are reported only for a single target or at most a narrow class of targets, MSSQ has been applied to general-purpose processors, standard DSPs, and ASIPs, and it was able to generate code. This is essentially due to the fact, that MSSQ uses very detailed processor models, and derives all required information automatically from these models, instead of making assumptions about the target architecture in advance. Furthermore, the hardware description language MIMOLA provides a well-defined, flexible, and easy-to-learn interface to MSSQ. In this way, a close link to hardware synthesis tools and simulators is provided. Alternatively, a corresponding subset of VHDL could be used, which would provide compliance with existing standards.

Limitations of the current MSSQ version mainly concern code quality. Although the quality of MSSQ-generated code is comparable with the quality achieved by contemporary processor-specific compilers, more advanced code optimization techniques are definitely necessary, in particular for DSPs. By considering larger entities than single source code statements, more potential parallelism can be revealed for code compaction. Furthermore, MSSQ will benefit from faster register allocation techniques, as for instance presented in [46]. Recent research results show, that the structural modelling approach does not contradict such advanced code generation techniques based on behavioral and mixed models, since these can be constructed from structural model by means of *instruction-set extraction* [65, 66]. Work is in progress to overcome the limitations of MSSQ. We are currently considering special enhancements tailored to RISCs [52] and DSPs [67, 68] within the context of the MAPS and RECORD compiler projects. We believe that usage of advanced code optimization techniques, yet retaining the flexibility achieved with structural processor models, is possible and bears the potential of significant progress in code generation for embedded processors.

Acknowledgments

The authors would like to thank Ralf Niemann for performing some of the experimental work. Material on the digital bass boost ASIP was provided by Jef van Meerbergen, Philips Research Labs, Eindhoven (The Netherlands). The financial support by the European Union through ESPRIT project 9138 (CHIPS) is gratefully acknowledged.

References

1. R.K. Gupta, G. De Micheli: *System-Level Synthesis Using Re-Programmable Components*, European Conference on Design Automation (EDAC), 1992, pp. 2-8
2. P. Chou, G. Boriello: *Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems*, 31st Design Automation Conference (DAC), 1994, pp. 1-4
3. R. Ernst, J. Henkel, T. Benner: *Hardware-Software Cosynthesis for Microcontrollers*, IEEE Design & Test Magazine, no. 12, 1993, pp. 64-75
4. K. Buchenrieder, A. Sedlmeier, C. Veith: *Design of HW/SW Systems with VLSI Subsystems Using CODES*, 6th IEEE Workshop on VLSI Signal Processing, 1993, pp. 233-239
5. A. Kalavade, E.A. Lee: *A Hardware-Software Codesign Methodology for DSP Applications*, IEEE Design & Test Magazine, no. 9, 1993, pp. 16-28
6. R. Camposano, J. Wilberg: *Embedded System Design*, Design Automation for Embedded Systems, vol. 1, nos. 1-2, 1996
7. G. Goossens, F. Catthoor, D. Lanneer, H. De Man: *Integration of Signal Processing Systems on Heterogeneous IC Architectures*, 5th High-Level Synthesis Workshop (HLSW), 1992, pp. 16-26

8. P. Paulin, C. Liem, T. May, S. Sutarwala: *DSP Design Tool Requirements for the Nineties: An Industrial Perspective*, Technical Report, Bell Northern Research, 1992
9. M. Strik, J. van Meerbergen, A. Timmer, J. Jess, S. Note: *Efficient Code Generation for In-House DSP Cores*, European Design and Test Conference (ED & TC), 1995, pp. 244-249
10. V. Zivojnovic, J.M. Velarde, C. Schläger: *DSPStone – A DSP-Oriented Benchmarking Methodology*, Technical Report, Dept. of Electrical Engineering, Institute for Integrated Systems for Signal Processing, University of Aachen, Germany, 1994
11. P. Marwedel: *A new Synthesis Algorithm for the MIMOLA Software System*, 23rd Design Automation Conference (DAC), 1986, pp. 271-277
12. P. Marwedel: *Matching System and Component Behaviour in the MIMOLA Synthesis Tools*, European Conference on Design Automation (EDAC), 1990, pp. 146-156
13. P. Marwedel: *Tree-based Mapping of Algorithms to Predefined Structures*, Int. Conf. on Computer-Aided Design (ICCAD), 1993, pp. 586-993
14. G. Krüger: *A Tool for Hierarchical Test Generation*, IEEE Trans. on CAD, vol. 10, no. 4, 1991, pp. 519-524
15. U. Bieker, P. Marwedel: *Retargetable Self-Test Program Generation Using Constraint Logic Programming*, 32nd Design Automation Conference (DAC), 1995, pp. 605-611
16. P. Marwedel, W. Schenk: *Cooperation of Synthesis, Retargetable Code Generation and Test Generation in the MIMOLA Software System*, European Conference on Design Automation (EDAC), 1993, pp. 63-69
17. M.E. Conway: *Proposal for an* UNCOL, Comm. of the ACM, vol. 1, 1958
18. R.S. Glanville: *A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers*, Doctoral thesis, University of California at Berkeley, 1977
19. R.G.G. Cattell: *Formalization and Automatic Derivation of Code Generators*, Doctoral thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, 1978
20. M. Ganapathi, C.N. Fischer, J.L. Hennessy: *Retargetable Compiler Code Generation*, ACM Computing Surveys, vol. 14, 1982, pp.573-592
21. R.M. Stallmann: *Using and Porting GNU CC* V2.4, Free Software Foundation, Cambridge/Massachusetts, 1993
22. H. Emmelmann, F.W. Schröer, R. Landwehr: *BEG – A Generator for Efficient Backends*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), SIGPLAN Notices 24, no. 7, 1989, pp. 227-237
23. A.V. Aho, M. Ganapathi, S.W.K Tjiang: *Code Generation Using Tree Matching and Dynamic Programming*, ACM Trans. on Programming Languages and Systems 11, no. 4, 1989, pp. 491-516
24. C.W. Fraser, D.R. Hanson, T.A. Proebsting: *Engineering a Simple, Efficient Code Generator Generator*, ACM Letters on Programming Languages and Systems, vol. 1, no. 3, 1992, pp. 213-226
25. D.J. DeWitt: *A Machine Independent Approach to the Production of Optimal Horizontal Microcode*, Doctoral thesis, Technical Report 76 DT 4, University of Michigan, 1976
26. S. Davidson, D. Landskov, B.D. Shriver, P.W. Mallet: *Some Experiments in Local Microcode Compaction for Horizontal Machines*, IEEE Trans. on Computers, vol. 30, no. 7, 1981, pp. 460-477
27. J.A. Fisher: *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Trans. on Computers, vol. 30, no. 7, 1981, pp. 478-490
28. A. Aiken, A. Nicolau: *A Development Environment for Horizontal Microcode*, IEEE Trans. on Software Engineering, no. 14, 1988, pp.584-594
29. R. Potasman, J. Lis, A. Nicolau, D. Gajski: *Percolation Based Synthesis*, 27th Design Automation Conference (DAC), 1990, pp. 444-449
30. T. Baba, H. Hagiwara: *The MPG System: A Machine Independent Efficient Microprogram Generator*, IEEE Trans. on Computers, vol. 30, no. 6, 1981, pp. 373-395
31. S.R. Vegdahl: *Local Code Generation and Compaction in Optimizing Microcode Compilers*, Doctoral thesis, Dept. of Computer Science, Carnegie-Mellon University, 1982
32. S.R. Vegdahl: *Phase Coupling and Constant Generation in an Optimizing Microcode Compiler*, 15th Ann. Workshop on Microprogramming (MICRO-15), 1982, pp. 125-133

33. R.A. Mueller, J. Varghese: *Flow Graph Machine Models in Microcode Synthesis*, 16th Ann. Workshop on Microprogramming (MICRO-16), 1983, pp.159-167
34. K. Rimey, P.N. Hilfinger: *Lazy Data Routing and Greedy Scheduling for Application-Specific Signal Processors*, 21st Annual Workshop on Microprogramming and Microarchitecture (MICRO-21), 1988, pp. 111-115
35. R. Hartmann: *Combined Scheduling and Data Routing for Programmable ASIC Systems*, European Conference on Design Automation (EDAC), 1992, pp. 486-490
36. B. Wess: *On the Optimal Generation for Signal Flow Graph Computation*, IEEE Int. Symp. on Circuits and Systems (ISCAS), 1990, pp. 444-447
37. A.V. Aho, S.C. Johnson: *Optimal Code Generation for Expression Trees*, Journal of the ACM, vol. 23, no. 3, 1976, pp. 488-501
38. B. Wess: *Automatic Code Generation for Integrated Digital Signal Processors*, IEEE Int. Symp. on Circuits and Systems (ISCAS), 1991, pp. 33-36
39. A. Fauth, A. Knoll: *Translating Signal Flowcharts into Microcode for Custom Digital Signal Processors*, Int. Conf. on Signal Processing (ICSP), 1993, pp. 65-68
40. A. Fauth, G. Hommel, A. Knoll, C. Müller: *Global Code Selection for Directed Acyclic Graphs*, in: P.A. Fritzson (ed.): 5th Int. Conference on Compiler Construction, 1994
41. D. Lanneer, M. Cornero, G. Goossens, H. De Man: *Data Routing: A Paradignm for Efficient Data-Path Synthesis and Code Generation*, 7th Int. Symp. on High-Level Synthesis (HLSS), 1994, pp. 17-21
42. J. Van Praet, G. Goossens, D. Lanneer, H. De Man: *Instruction Set Definition and Instruction Selection for ASIPs*, 7th Int. Symp. on High-Level Synthesis (HLSS), 1994, pp. 11-16
43. A. Fauth, J. Van Praet, M. Freericks: *Describing Instruction-Set Processors in nML*, European Design and Test Conference (ED & TC), 1995, pp. 503-507
44. C. Liem, T. May, P. Paulin: *Instruction-Set Matching and Selection for DSP and ASIP Code Generation*, European Design and Test Conference (ED & TC), 1994, pp. 31-37
45. C. Liem, T. May, P. Paulin: *Register Assignment through Resource Classification for ASIP Microcode Generation*, Int. Conf. on Computer-Aided Design (ICCAD), 1994, pp. 397-402
46. G. Araujo, S. Malik: *Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures*, 8th Int. Symp. on System Synthesis (ISSS), 1995, pp. 36-41
47. S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Storage Assignment to Decrease Code Size*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995
48. D.H. Bartley: *Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes*, Software – Practice and Experience, vol. 22(2), 1992, pp. 101-110
49. S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Code Optimization Techniques for Embedded DSP Microprocessors*, 32nd Design Automation Conference (DAC), 1995, pp. 599-604
50. A. Sudarsanam, S. Malik: *Memory Bank and Register Allocation in Software Synthesis for ASIPs*, Int. Conf. on Computer-Aided Design (ICCAD), 1995, pp. 388-392
51. S. Novack, A. Nicolau, N. Dutt: *A Unified Code Generation Approach using Mutation Scheduling*, chapter 12 in [55]
52. W. Schenk: *Retargetable Code Generation for Parallel, Pipelined Processor Structures*, chapter 7 in [55]
53. T. Wilson, G. Grewal, B. Halley, D. Banerji: *An Integrated Approach to Retargetable Code Generation*, 7th Int. Symp. on High-Level Synthesis (HLSS), 1994, pp. 70-75
54. A. Timmer, M. Strik, J. van Meerbergen, J. Jess: *Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores*, 32nd Design Automation Conference (DAC), 1995, pp. 593-598
55. P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995
56. S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer: *The MIMOLA Language V4.1*, Technical Report, University of Dortmund, Dept. of Computer Science, September 1994

57. P. Marwedel: *Implementation of IF-statements in the TODOS microarchitecture synthesis system*, in: G. Saucier, J. Trilhe (eds.): *Synthesis for Control Dominated Circuits*, IFIP Trans. A-22, North-Holland, 1993, pp. 249-262
58. Advanced Micro Devices: *Bipolar Microprocessor Logic and Interface Data Book*, Sunnyvale, 1983
59. L. Nowak: *SAMP: A General Purpose Processor Based on a Self-Timed VLIW Structure*, ACM Comp. Arch. News, vol. 15, no. 4, 1987, pp. 32-39
60. W. Schenk: *A High Speed PROLOG Implementation on a VLIW Processor*, Microprocessing and Microprogramming, vol. 27, nos. 1-5, 1989, pp. 601-606
61. C. Albrecht, S. Bashford, P. Marwedel, A. Neumann, W. Schenk: *The Design of the PRIPS Microprocessor*, 4th EUROCHIP Workshop on VLSI Design Training, 1993, pp. 254-259
62. L. Nowak: *Graph Based Retargetable Microcode Compilation in the MIMOLA Design System*, 20th Ann. Workshop on Microprogramming (MICRO-20), 1987, pp. 126 - 132
63. Texas Instruments: *TMS320C2x User's Guide*, rev. B, 1990
64. Motorola Inc.: *DSP 56156 Digital Signal Processor User's Manual*, 1992
65. R. Leupers, P. Marwedel: *A BDD-based Frontend for Retargetable Compilers*, European Design & Test Conference (ED & TC), 1995, pp. 239-243
66. R. Leupers, P. Marwedel: *Retargetable Generation of Code Selectors from HDL Processor Models* European Design & Test Conference (ED & TC), 1997
67. R. Leupers, P. Marwedel: *Time-constrained Code Compaction for DSPs*, 8th Int. System Synthesis Symposium (ISSS), 1995, pp. 54-59
68. R. Leupers, P. Marwedel: *Algorithms for Address Assignment in DSP Code Generation*, Int. Conference on Computer-Aided Design (ICCAD), 1996

Received Date
Accepted Date
Final Manuscript Date