Time-Constrained Code Compaction for DSPs

Rainer Leupers, Peter Marwedel

Abstract— **This paper addresses instruction-level parallelism in code generation for DSPs. In presence of potential parallelism, the task of code generation includes code compaction, which parallelizes primitive processor operations under given dependency and resource constraints. Furthermore, DSP algorithms in most cases are required to guarantee real-time response. Since the exact execution speed of a DSP program is only known after compaction, real-time constraints should be taken into account during the compaction phase. While previous DSP code generators rely on rigid heuristics for compaction, we propose a novel approach to exact local code compaction based on an Integer Programming model, which handles time constraints. Due to a general problem formulation, the IP model also captures encoding restrictions and handles instructions having alternative encodings and side effects, and therefore applies to a large class of instruction formats. Capabilities and limitations of our approach are discussed for different DSPs.**

Keywords— **Retargetable compilation, embedded DSPs, code compaction**

I. Introduction

RESEARCH on electronic CAD is currently taking the step towards system-level design automation. For economical reasons, contemporary embedded VLSI systems are of heterogeneous nature, comprising both hardware and software components in the form of ASICs and embedded programmable processors. Consequently, system-level CAD tools need to provide support for integrated hardware and software synthesis. Software synthesis is the task of extracting those pieces of functionality from a system specification, which should be assigned to programmable processors, and mapping these pieces into executable, processor-specific machine code.

The general optimization goal in hardware/software co-synthesis of embedded VLSI systems is to minimize the amount of custom hardware needed to implement a system under given performance constraints. This is due to the fact, that implementation by software provides more flexibility, lower implementation effort, and better opportunities for reuse. On the other hand, software synthesis turns out to be a bottleneck in design of systems comprising programmable digital signal processors (DSPs): Most DSP software is still coded at the assembly-language level [1], in spite of the well-known drawbacks of low-level programming. Although high-level language compilers for off-the-shelf DSPs are available, the execution speed overhead of compiler-generated code (up to several hundred percent compared to hand-crafted code [2]) is mostly unacceptable. The reason for this overhead is, that compilers are hardly capable of exploiting the highly dedicated and irregular architectures of DSPs. Furthermore, there is still no designated standard programming language for DSPs. The situation is even worse for *application-specific* DSPs (ASIPs). Since these are typically low-volume and product-specific designs, high-level language compilers for ASIPs hardly exist. Nevertheless, ASIPs are expected to gain increasing market shares in relation to standard DSPs [1].

Current research efforts to overcome the productivity bottleneck in DSP code generation concentrate on two central issues [3]:

Code quality: In order to enable utilization of high-level language compilers, the code overhead must be reduced by an order of magnitude. This can only be achieved by means of new DSP-specific code optimization techniques, reaching beyond the scope of classical compiler construction. Classical optimization techniques, intended for large programs on general-purpose processors, primarily focus on high compilation speed, and thus have a limited effect. In constrast, generation of efficient DSP code justifies much higher compilation times. Therefore, there are large opportunities for new optimization techniques, aiming at very high quality code generation within any reasonable amount of compilation time.

Retargetability: In order to introduce high-level language compilers into code generation for ASIPs, it is necessary to ensure flexibility of code generation techniques. If a compiler can quickly be retargeted to a new processor architecture, then compiler development will become economically feasible even for low-volume DSPs. In an ideal case, a compiler supports retargeting by reading and analyzing an external, user-editable model of the target processor, for which code is to be generated. Such a way of retargetability would permit ASIP designers to quickly study the mutual dependence between hardware architectures and program execution speed already at the processor level.

The purpose of this paper is to present a code optimization technique, which aims at thoroughly exploiting potential parallelism in DSP machine programs by exact *local code compaction*. Although code compaction is a well-tried concept for VLIW machines, effective compaction techniques for DSPs, which typically show a rather restrictive type of instruction-level parallelism, have hardly been reported. The compaction technique proposed in this paper takes into account the peculiarities of DSP instruction formats as well as *time constraints* imposed on machine programs. Furthermore, it is completely retargetable within a class of instruction formats defined later in this paper.

Since we perform *exact* code compaction, the runtimes are significantly higher than for heuristic approaches. Nevertheless, as will be demonstrated, our compaction technique is capable of solving problems of relevant size within

Authors' affiliation: University of Dortmund, Department of Computer Science 12, 44221 Dortmund, Germany, E-mail: leupers|marwedel@ls12.informatik.uni-dortmund.de

acceptable amounts of computation time. Whenever tight time constraints demand for extremely dense code, exact code compaction thus is a feasible alternative to heuristic compaction.

The remainder of this contribution is organized as follows: Section II gives an overview of the RECORD compiler system, which employs the proposed code compaction technique in order to compile high-quality DSP code in a retargetable manner. In Section III, we provide the background for local code compaction and outline existing techniques. Then, we discuss limitations of previous work, which motivates an extended, DSP-specific definition of the code compaction problem, presented in Section IV. Section V specifies a formal model of code compaction by means of Integer Programming. Real-life examples and experimental results are given in Section VI, and Section VII concludes with a summary of results and hints for further research.

II. System overview

THE RECORD compiler project, currently being carried out at the University of Dortmund, is based on experiences gained with earlier retargetable compilers integrated in the MIMOLA hardware design system [4], [5]. RECORD is a retargetable compiler for DSPs, for which the main objective is to find a reasonable compromise between the antagonistic goals of retargetability and code quality. In the current version, RECORD addresses *fixed-point* DSPs with *single-cycle* instructions, and compiles DSP algorithms written in the DFL language [6] into machine instructions for an externally specified target processor model. The coarse compiler architecture is depicted in figure 1. The code generation process is subdivided into the following phases:

Intermediate code generation: The DFL source program is analyzed and is compiled into a control/dataflow graph (CDFG) representation. The basic CDFG entities are *expression trees* (ETs) of maximum size, which are obtained by data-flow analysis. Common subexpressions in ETs are resolved heuristically by node duplication [7].

Instruction-set extraction: A hardware description language (HDL) model of the target processor is analyzed and is converted into an internal graph representation. Currently, the MIMOLA 4.1 HDL [8] is used for processor modelling, but adaptation towards a VHDL subset would be straightforward. On the internal processor model, *instruction-set extraction* (ISE) is performed in order to determine the complete set of *register transfer (RT) patterns* available on the target processor [9], [10]. Additionally, extracted RT patterns are annotated with (possibly multiple) binary encodings (partial instructions). A partial instruction is a bitstring $I \in \{0, 1, x\}^L$, where L denotes the instruction word-length, and x is a don't care value.

Compared to related retargetable DSP compiler systems, such as MSSQ [4], CHESS [11], and CodeSyn [12], the concept of ISE is a unique feature of RECORD: It accepts target processor models in a real HDL,

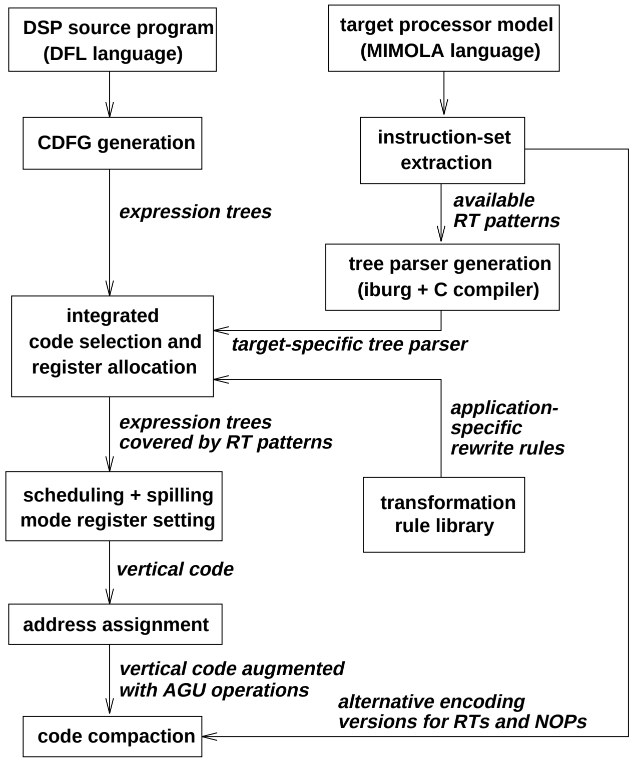
DSP source program (DFL language)
target processor model (MIMOLA language)
CDFG generation
instruction-set extraction
expression trees
available RT patterns
tree parser generation (iburg + C compiler)
integrated code selection and register allocation
target-specific tree parser
application-specific rewrite rules
expression trees covered by RT patterns
scheduling + spilling mode register setting
transformation rule library
vertical code
address assignment
vertical code augmented with AGU operations
alternative encoding versions for RTs and NOPs
code compaction

Fig. 1. Architecture of the RECORD compiler

which provides a convenient interface to CAD frameworks. Furthermore, processors may be modelled at different abstraction levels, ranging from purely behavioral (instruction-set) descriptions down to RT-level netlists, consisting of functional units, registers, busses, and logic gates. Due to usage of *binary decision diagrams* (BDDs) [13] for control signal analysis, ISE can be performed efficiently and also eliminates undesired effects resulting from syntactic variances in processor models. In this way, ISE provides the necessary link between hardware-oriented processor models and advanced code generation techniques. The extracted RT patterns form a set of *tree templates*, each of which represents a primitive, single-cycle processor operation. Such an operation reads values from registers, storages, or input ports, performs a computation, and writes the result into a register, storage cell, or output port.

Tree parser generation: From the extracted RT patterns, a fast processor-specific tree parser is automatically generated by means of the `iburg` code generator generator [14]. The generated tree parser computes optimal implementations of expression trees with respect to the number of selected RT patterns. This includes binding of values to special-purpose registers as well as exploitation of *chained operations*, such as multiply-accumulates. The advantages of tree parsing as a means of integrated code selection and register allocation for irregular processor architectures have already been pointed out in [15] and [16]. ISE and tree parser generation have to be performed only once for each new target processor and can be reused for different source programs.

Code selection and register allocation: ETs in the intermediate format are consecutively mapped into processor-specific RTs by the tree parser. The high speed efficiency of tree parsing permits consideration of different, semantically equivalent alternatives for each ET. Alternative ETs are generated based on a user-editable library of *transformation rules.* Transformation rules are rewrite rules, which are necessary to cope with unforeseen idiosyncrasies in the target processor, and can also increase code quality by exploitation of algebraic rules. Simple algebraic rules, such as commutativity, can already be incorporated into the tree parser at virtually no extra cost in compilation speed (see also [17], [18]). From each set of alternative ETs, the one with the smallest number of required RTs is selected.

Vertical code generation: Code selection and register allocation by tree parsing yield a *register-transfer tree*, which represents the covering of an ET by processor-specific RT patterns. During vertical code generation, register-transfer trees are heuristically sequentialized, so as to minimize the spill requirements for register files with limited capacity. Necessary spill and reload code is inserted, as well as additional RTs adjusting possibly required mode register states (arithmetic modes, shift modes). Mode registers store control signals, which need to be changed only rarely. In the area of microprogramming, mode registers correspond to the concept of *residual control.* Mode register requirements of RTs are determined together with partial instructions during instruction-set extraction. After vertical code generation, the machine program consists of a set of RT-level basic blocks.

Address assignment: Generated RTL basic blocks are augmented with RTs, which ensure effective utilization of parallel *address generation units* (AGUs) for memory addressing. This is accomplished by computing an appropriate memory layout for program variables [19], [20].

Code compaction: After address assignment, all RTs necessary to implement the desired program behavior have been generated, and are one block at a time passed to the code compaction phase, which is the subject of this paper. During code compaction, potential parallelism at the instruction level is exploited, while obeying inter-RT dependencies and conflicts imposed by binary encodings of RT patterns. The result is executable processor-specific machine code.

III. Local code compaction

CODE compaction deals with parallelizing a set of RTs under given dependency relations and constraints by assigning RTs to control steps. The set of all RTs assigned to the same control step form a *machine instruction.* The general optimization goal is to minimize the number of control steps. *Local* compaction starts from an RTL basic block $BB = (r_1, \ldots, r_n)$, i.e. a sequence of RTs generated by previous compilation phases. In contrast, *global* code compaction permits RTs to be moved across basic block boundaries. In this paper, we consider local compaction, because of the following reasons:

1. Effective global compaction techniques need to employ local techniques as subroutines. However, as indicated by experimental surveys [2], even local compaction is not a well-solved problem for DSPs. Therefore, it seems reasonable to first study local techniques in more detail.
2. Popular global techniques, such as Trace Scheduling [21], Percolation Scheduling [22], and Mutation Scheduling [23], have been shown to be effective mainly for highly parallel and regular architectures, in particular VLIWs. Contemporary DSPs, however, are not *highly* parallel and tend to have an irregular architecture.
3. In order to preserve semantical correctness of compacted programs, global techniques need to insert *compensation code* when moving RTs across basic blocks. Compensation code may lead to a significant increase in code size, which contradicts the goal of minimizing on-chip program ROM area.

For local code compaction, it is sufficient to represent an RT by a pair $r_i = (W_i, R_i)$, where W_i is a *write location*, and R_i is a set of *read locations.* Write and read locations are registers, memory cells, and processor I/O ports. Between any pair r_i, r_j of RTs in a basic block $BB = (r_1, \ldots, r_n)$, the following *dependency relations* need to be obeyed in order to preserve semantical correctness:

- r_j is **data-dependent** on r_i ("$r_i \overset{DD}{\Leftrightarrow} r_j$"), if $W_i \in R_j$, and r_i produces a program value consumed by r_j.
- r_j is **output-dependent** on r_i ("$r_i \overset{OD}{\Leftrightarrow} r_j$"),
$$W_i = W_j \quad \text{and} \quad j > i$$
- r_j is **data-anti-dependent** on r_i ("$r_i \overset{DAD}{\Leftrightarrow} r_j$"), if there exists an RT r_k, such that
$$r_k \overset{DD}{\Leftrightarrow} r_i \quad \text{and} \quad r_k \overset{OD}{\Leftrightarrow} r_j.$$

The dependency relations impose a partial ordering on $\{r_1, \ldots, r_n\}$, which can be represented by a DAG:

Definition: For an RTL basic block $BB = (r_1, \ldots, r_n)$, the **RT dependency graph** (RDG) is an edge-labelled directed acyclic graph $G = (V, E, w)$, with $V = \{r_1, \ldots, r_n\}$, $E \subseteq V \times V$, and $w : E \to \{DD, DAD, OD\}$.

In the hardware model used by Record, all RTs are single-cycle operations, all registers permit at most one write access per cycle, and all registers can be written and read within the same cycle. This leads to the following "basic" definition of the code compaction problem:

Definition: A **parallel schedule** for an RDG $G = (V, E, w)$ is a mapping $CS : \{r_1, \ldots, r_n\} \to \mathbf{N}$, from RTs to control steps, so that for all $r_i, r_j \in V$:

$$r_i \overset{DD}{\Leftrightarrow} r_j \quad \Rightarrow \quad CS(r_i) < CS(r_j)$$

$$r_i \overset{QD}{\Leftrightarrow} r_j \quad \Rightarrow \quad CS(r_i) < CS(r_j)$$
$$r_i \overset{DAD}{\Leftrightarrow} r_j \quad \Rightarrow \quad CS(r_i) \leq CS(r_j)$$

Code compaction is the problem of constructing a schedule CS such that

$$\max\{CS(r_1), \ldots, CS(r_n)\} \quad \rightarrow \quad \min$$

The **as-soon-as-possible** time $ASAP(r_i)$ of an RT r_i is defined as:

$$\begin{array}{ll} ASAP(r_i) = & \\ \max\{ & \max\{ASAP(r_j) + 1 \quad | \quad (r_j \overset{DD}{\Leftrightarrow} r_i) \vee (r_j \overset{QD}{\Leftrightarrow} r_i)\}, \\ & \max\{ASAP(r_j) \quad | \quad r_j \overset{DAD}{\Leftrightarrow} r_i\}\} \end{array}$$

with $\max\{\emptyset\} := 1$.
The **critical path length** L_c of an RDG is

$$\max\{ASAP(r_i) \quad | \quad i \in \{1, \ldots, n\}\}$$

which provides a lower bound on the minimum schedule length.
The **as-late-as-possible** time $ALAP(r_i)$ of an RT r_i is defined as:

$$\begin{array}{ll} ALAP(r_i) = & \\ \min\{ & \min\{ALAP(r_j) \Leftrightarrow 1 \quad | \quad (r_i \overset{DD}{\Leftrightarrow} r_j) \vee (r_i \overset{QD}{\Leftrightarrow} r_j)\}, \\ & \min\{ALAP(r_j) \quad | \quad r_i \overset{DAD}{\Leftrightarrow} r_j\}\} \end{array}$$

with $\min\{\emptyset\} := L_c$.
An RT r_i lies on a **critical path** in an RDG, if $ASAP(r_i) = ALAP(r_i)$.

In case of unlimited hardware resources, code compaction can be efficiently solved by topological sorting. Real target architectures however impose resource limitations, which may inhibit parallel execution of pairwise independent RTs. These limitations can be captured by an *incompatibility relation*

$$\not\sim \quad \subseteq \quad V \times V$$

comprising all pairs of RTs, that cannot be executed in parallel due to a resource conflict. Incompatibilities impose the additional constraint

$$\forall r_i, r_j \in V: \quad r_i \not\sim r_j \quad \Rightarrow \quad CS(r_i) \neq CS(r_j)$$

on code compaction, in which case compaction becomes a resource-constrained scheduling problem, known to be NP-hard [24].

Heuristic code compaction techniques became important with appearance of VLIW machines in the early eighties. Popular heuristics include *first-come first-served*, *critical path*, and *list scheduling*. These three $\mathcal{O}(n^2)$ algorithms have been empirically evaluated by Mallett et al. [25]. It was concluded, that each algorithm is capable of producing close-to-optimum results in most cases, while differing in speed, simplicity, and quality in relation to the basic block length n. Nevertheless, the above techniques were essentially developed for horizontal machines with few restrictions imposed by the instruction format, i.e., resource conflicts are mainly caused by restricted *datapath* resources.

IV. Compaction requirements for DSPs

Many DSPs, in particular standard components, do not show horizontal, but *strongly encoded* instruction formats in order to limit the silicon area requirements of on-chip program ROMs. An instruction format is strongly encoded, if the instruction word-length is small compared to the total number of control lines for RT-level processor components. As a consequence, instruction encoding prevents much potential parallelism, that is exposed, if the pure datapath is considered, and most inter-RT conflicts actually arise from *encoding conflicts*. Instruction-level parallelism is restricted to a few special cases, which are assumed to provide the highest performance gains with respect to DSP requirements. A machine instruction of maximum parallelism typically comprises one or two arithmetic operations, data moves, and address register updates. However, there is no full orthogonality between these operations types: Certain arithmetic operation can only be executed in parallel to a data move to a certain special-purpose register, an address register update cannot be executed in parallel to all data moves, and so forth. Thus, compaction algorithms for DSPs have to scan a relatively large search space in order to detect sets of RTs qualified for parallelization. The special demands on code compaction techniques for DSPs are discussed in the following.

A. *Conflict representation*

In presence of datapath resource and encoding conflicts, it is desirable to have a uniform conflict representation. As already observed for earlier MIMOLA-based compilers [5], checking for inter-RT conflicts in case of single-cycle RTs can be performed in a uniform way by checking for conflicts in the partial instructions associated with RTs. Two partial instructions

$$I_1 = (a_1, \ldots, a_L), \quad I_2 = (b_1, \ldots, b_L)$$

with $a_i, b_i \in \{0, 1, x\}$ are conflicting, if there exists an $i \in \{1, \ldots, L\}$, such that

$$(a_i = 1 \quad \wedge \quad b_i = 0) \quad \text{or} \quad (a_i = 0 \quad \wedge \quad b_i = 1)$$

In our approach, partial instructions are automatically derived from the external processor model by instruction-set extraction. Encoding conflicts are obviously represented in the partial instructions. The same holds for datapath resource conflicts, if control code for datapath resources are assumed to be adjusted by the instruction word. This assumption does not hold in two special cases: Firstly, there may be conflicts with respect to the required *mode register states* of RTs. In Record, mode register states are adjusted *before* compaction by inserting additional RTs. Therefore, parallel scheduling of RTs with conflicting mode requirements is prevented by additional inter-RT dependencies.

The second special case occurs in presence of *tristate busses* in the processor model. Unused bus drivers need to be deactivated in each control step, in order to avoid unpredictable machine program behavior due to *bus contentions*.

By deriving the necessary control code settings for all bus drivers already during instruction-set extraction, it is possible to map bus contentions to usual encoding conflicts.

B. Alternative encoding versions

In general, each RT r_i is not associated with a unique partial instruction, but with a *set of alternative encodings* $\{e_{i1}, \ldots, e_{in_i}\}$. Alternative encodings may arise from alternative routes for moving a value through the datapath. In other cases, alternatives are due to instruction format: The TMS320C2x DSP [26], for instance, permits execution of address register updates in parallel to different arithmetic or data move instructions. Each address register update is represented by a different opcode, resulting in a number of alternative encodings to be considered. The same also holds for other operations, for instance, the partial instructions

```
1) 00111100xxxxxxxx (LT)
2) 00111101xxxxxxxx (LTA)
3) 00111111xxxxxxxx (LTD)
4) 00111110xxxxxxxx (LTS)
```

are alternative encodings for the same RT, namely loading the "T" register from memory. Compatibility of RTs strongly depends on the selected encoding versions. Three RTs r_i, r_j, r_k may have pairwise compatible versions, but scheduling r_i, r_j, r_k in parallel may be impossible. Therefore, careful *selection of encoding versions* during compaction is of outstanding importance for DSPs. In [25], *version shuffling* was proposed as a technique for version selection, which can be integrated into heuristic algorithms: Whenever some RT r_i is to be assigned to a control step t, the cross product of all versions for r_i and all versions of RTs already assigned to t are checked for a combination of compatible versions. However, version shuffling does not permit to remove an "obstructive" RT from a control step t, once it has been bound to t, and therefore has a limited optimization effect.

C. Side effects

A side effect is an undesired RT, which may cause incorrect behavior of a machine program. Most compaction approaches assume, that the instruction format is such that side effects are excluded in advance. However, if arbitrary instruction formats are to be handled, two kinds of side effects must be considered during code compaction.

Horizontal side effects occur in weakly enocoded instruction formats, where several instruction bits remain don't care for each control step t. Whenever such a don't care bit steers a register or memory, which may contain a live value in CS_t, the register must be explicitly deactivated. This can be accomplished by scheduling of *no-operations* (NOPs) for unused registers. NOPs are special partial instructions, which ensure that certain registers are disabled from loading a new value during a certain control step. Partial instructions for NOPs can be computed as a "by-product" of instruction-set extraction. As for RTs, alternative NOP encoding versions can be present for the same register. However, NOPs do not necessarily exist for all registers, e.g. in architectures with extensive datapath pipelining. In this case, compaction must permit to *tolerate* side effects on registers not containing live values. If this is not taken into account already during compaction, code generation is likely to fail, although a solution might exist.

The second type of side effects, which we call **vertical side effects**, occurs in presence of strongly encoded instruction formats. A vertical side effect is exposed, if an encoding version e_{ik} for an RT r_i is "covered" by a version $e_{jk'}$ of another RT r_j. That is, selection of e_{ik} for r_i implies that r_j will be executed in the same control step. If r_j happens to be an RT ready to be scheduled, this side effect can be exploited. On the other hand, version selection must discard version e_{ik}, whenever this is not the case, and r_j might destroy a live value. Vertical side effects are exemplified in the TMS320C2x instruction set: The partial instructions LTA, LTD, LTS shown above have a side effect on the accumulator register. If version selection is completed *before* NOPs are packed, then vertical side effects can be prevented at most by coincidence.

A special aspect of vertical side effects are multiply-accumulates (MACs) on DSPs. A MAC executes two operations `P = Y * Z` and `A = A + P` within a single cycle. On some DSPs, for instance Motorola DSP56xxx [27], MACs are *data-stationary*, i.e. multiplication and addition are executed in *chained* mode. In contrast, the TMS320C2x incorporates *time-stationary* MACs, in which case value `P` is buffered in a register. From a code generation point of view, there is a strong difference between these MAC types: Data-stationary MACs can already be captured during parsing of expression trees, while generation of time-stationary MACs must be postponed to code compaction. In turn, this demands for compaction methods capable of handling vertical side effects.

D. Time constraints

In most cases, DSP algorithms are subject to real-time constraints. While techniques for hardware synthesis under time constraints are available, incorporation of time constraints into *code generation* has hardly been treated so far. Unfortunate decisions during code selection and register allocation may imply that a given maximum time constraint cannot be met, so that backtracking may be necessary. However, a prerequisite of time-constrained code selection and register allocation is availability of time-constrained code compaction techniques. This is due to the fact, that only compaction makes the critical path and thus the worst-case execution speed of a machine program exactly known. Therefore, compaction techniques are desirable, which parallelize RTs with respect to a given (maximum) time constraint of $T_{\max}$ machine cycles. It might be the case, that a locally suboptimal scheduling decision leads to satisfaction of $T_{\max}$, while a rigid optimization heuristic fails.

E. Approaches to DSP code compaction

Heuristic compaction algorithms have been adopted for several recent DSP code generators. Wess' compiler [15] uses the critical path algorithm, which achieves code size reductions between 30 % and 50 % compared to vertical code for an ADSP210x DSP. The range of possible instruction formats that can be handled is however not reported. For the CodeSyn compiler [12], only compaction for horizontal machines has been described. The CHESS compiler [11] uses a list scheduling algorithm which takes into account encoding conflicts, alternative versions, and vertical side effects. Horizontal side effects and bus conflicts are a priori excluded due to limitations of the processor modelling language. In [28], a Motorola 56xxx specific compaction method is described, however excluding *out-of-order execution*, i.e. the schedule satisfies $i < j \Rightarrow CS(r_i) \leq CS(r_j)$ for any two RTs r_i, r_j, independent of dependency relations.

An exact (non-heuristic) compaction method, which does take into account time constraints, has been reported by Wilson et al. [29]. The proposed Integer Programming (IP) approach integrates code selection, register allocation, and compaction. The IP model comprises alternative versions and vertical side effects, but no bus conflicts and horizontal side effects. Furthermore, the IP model – at least in its entirety – turned out to be too complex for realistic target processors, and requires a large amount of manual description effort.

The graph-based compaction technique presented by Timmer [30] achieves comparatively low runtimes for exact code compaction under time constraints by pruning the search space in advance. The pruning procedure is based on the assumption, that inter-RT conflicts are fixed before compaction. In this case, the RTs can be partitioned into maximum sets of pairwise conflicting RTs, for which separate sequential schedules can be constructed efficiently. Timmer's techniques produced very good results for a family of real-life ASIPs, but has restricted capabilities with respect to alternative versions and side effects. The abovementioned assumption implies, that incompatibility of versions e_{ik} for RT r_i and $e_{jk'}$ for RT r_j implies pairwise incompatibility of *all* versions for r_i and r_j.

The limitations of existing DSP compaction techniques motivate an extended definition of the code compaction problem, which captures alternative versions, side effects, and time constraints:

Definition: Let $BB = (r_1, \ldots, r_n)$ be an RTL basic block, where each r_i has a set $E_i = \{e_{i1}, \ldots, e_{in_i}\}$ of alternative encodings. Furthermore, let $NOP = \{NOP_1, \ldots, NOP_r\}$ denote the set of no-operations for all registers $\{X_1, \ldots, X_r\}$ that appear as destinations of RTs in BB, and let $\{nop_{j1}, \ldots, nop_{jn_j}\}$ be the set of alternative versions for all $NOP_j \in NOP$.

A **parallel schedule** for BB is a sequence $CS = (CS_1, \ldots, CS_n)$, so that for any r_i, r_j in BB the following conditions hold:

- Each $CS_t \in CS$ is a subset of
$$\bigcup_{j=1}^{n} E_j \quad \cup \quad \bigcup_{j=1}^{r} NOP_j$$
- There exists exactly one $CS_t \in CS$, which contains an encoding version of r_i (notation: $cs(r_i) = t$).
- If $r_i \overset{DD}{\Leftrightarrow} r_j$ or $r_i \overset{OD}{\Leftrightarrow} r_j$ then $cs(r_i) < cs(r_j)$.
- If $r_i \overset{DAD}{\Leftrightarrow} r_j$, then $cs(r_i) \leq cs(r_j)$.
- If $r_i \overset{DD}{\Leftrightarrow} r_j$, then all control steps
$$CS_t \in \{CS_{cs(i)+1}, \ldots, CS_{cs(j)-1}\}$$
contain a NOP version for the destination of r_i.
- For any two encoding and NOP versions
$$e_1, e_2 \in \left[\bigcup_{j=1}^{n} E_j \quad \cup \quad \bigcup_{j=1}^{r} NOP_j\right]$$
there is no control step $CS_t \in CS$, for which
$$e_1, e_2 \in CS_t \quad \wedge \quad e_1 \not\sim e_2$$

For an RTL basic block BB whose RT dependency graph has critical path length L_c, **time-constrained code compaction** (TCC) is the problem of computing a schedule CS, such that, for a given $T_{\max} \in \{L_c, \ldots, n\}$, CS satisfies $CS_t = \emptyset$ for all $t \in \{T_{\max}+1, \ldots, n\}$.

TCC is the decision variant of *optimal* code compaction, extended by alternative encodings and side effects, and is thus NP-complete. This poses the question, which problem sizes can be treated within an acceptable amount of computation time. In the next section, we present a solution technique, which permits to compact basic blocks of relevant size in a retargetable manner.

V. Integer Programming formulation

RECENTLY , several approaches have been published, which map NP-complete VLSI-design related problems into an Integer (Linear) Programming model (e.g. [31], [32]), in order to study the potential gains of optimal solution methods compared to heuristics. IP is the problem of computing a setting of n integer *solution variables* $(z_1, \ldots, z_n)$, such that an *objective function* $f(z_1, \ldots, z_n)$ is minimized under the constraint
$$A \cdot (z_1, \ldots, z_n)^T \geq B$$
for an integer matrix A and an integer vector B. Although IP is NP-hard, thus excluding exact solution of large problems, modelling intractable problems by IP can be a promising approach, because of the following reasons:

- Since IP is based on a relatively simple mathematical notation, its is easily verified, that the IP formulation of some problem meets the problem specification.
- IP is a suitable method for formally describing heterogeneously constrained problems, because these constraints often have a straightforward representation in

form of linear inequations. Solving the IP means, that all constraints are *simultaneously* taken into account, which is not easily achieved in a problem-specific solution algorithm.

- Since IP is among the most important optimization problems, commercial tools are available for IP solving. These *IP solvers* rely on theoretical results from operations research, and are therefore considerably fast even for relatively large Integer Programs. Using an appropriate IP formulation thus often permits to optimally solve NP-hard problems of practical relevance.

Our approach to TCC is therefore largely based on IP. In contrast to Wilson's approach [29], the IP instances are not created manually, but are automatically derived from the given compaction problem and target processor model and an externally specified time constraint. Furthermore, it focusses *only* on the problem of code compaction, which extends the size of basic blocks which can be handled in practice.

Given an instance of TCC, first the *mobility range*

$$rng(r_i) := [ASAP(r_i), ALAP(r_i)]$$

is determined for each RT r_i, with $T_{\max}$ being the upper bound of ALAP times for all RTs. The solution variables of the IP model encode the RT versions selected for each control step. Dependencies and incompatibility constraints are represented by linear inequations. Solution variables are only defined for control step numbers up to $T_{\max}$, so that only constraint satisfaction is required. Any IP solution represents a parallel schedule with $T_{\max}$ control steps, possibly padded with NOPs. In turn, non-existence of an IP solution implies non-existence of a schedule meeting the maximum time constraint. The setting of solution variables also accounts for NOPs, which have to be scheduled in order to prevent undesired side effects. We permit arbitrary, multiple-version instruction formats, which meet the following assumptions:

A1: There exists at least one NOP version for all addressable storage elements (register files, memories). However, the NOP sets for single registers may be empty.

A2: For each storage element not written in a certain control step CS_t, a NOP version can be scheduled, *independently* of the RT versions assigned to CS_t.

These assumptions – which are satisfied for realistic processors – permit to insert NOP versions only *after* compaction by means of a version shuffling mechanism, such that the solution space for compaction is not affected.

A. Solution variables

The solution variables are subdivided into two classes of indexed *decision* (0/1) variables:

V-variables (triple-indexed): For each r_i with encoding version set E_i the following *V-variables* (version variables) are used:

$$\{v_{i,m,t} \quad | \quad m \in \{1, \ldots, |E_i|\} \quad \wedge \quad t \in rng(r_i)\}$$

The *interpretation* of V-variables is

$$v_{i,m,t} = 1 \quad :\Leftrightarrow$$

RT r_i is scheduled in control step number t with version $e_{im} \in E_i$.

N-variables (double-indexed): For the set $NOP = \{NOP_1, \ldots, NOP_r\}$ of no-operations, the following *N-variables* (NOP variables) are used:

$$\{n_{s,t} \quad | \quad s \in \{1, \ldots, r\} \quad \wedge \quad t \in [1, T_{\max}]\}$$

The *interpretation* of N-variables is

$$n_{s,t} = 1 \quad :\Leftrightarrow$$

Control step number t contains NOP for destination register X_s.

B. Constraints

The correctness conditions are encoded into IP constraints as follows:

Each RT is scheduled exactly once: This is ensured, if the sum over all V-variables for each RT r_i equals 1.

$$\forall r_i : \quad \sum_{t \in rng(r_i)} \sum_{m=1}^{|E_i|} v_{i,m,t} \quad = \quad 1$$

Data- and output-dependencies are not violated: If r_i, r_j are data- or output-dependent, and r_j is scheduled in control step CS_t, then r_i must be scheduled in an earlier control step, i.e., in the interval

$$[ASAP(r_i), t \Leftrightarrow 1]$$

This is captured as follows:

$$r_i \overset{DD}{\Leftrightarrow\!\!\rightarrow} r_j \quad \vee \quad r_i \overset{OD}{\Leftrightarrow\!\!\rightarrow} r_j \quad \Rightarrow$$

$$\forall t \in rng(r_i) \cap rng(r_j) :$$

$$\sum_{m=1}^{|E_j|} v_{j,m,t} \quad \leq \quad \sum_{t' \in [ASAP(r_i), t-1]} \sum_{m=1}^{|E_i|} v_{i,m,t'}$$

Data-anti-dependencies are not violated: Data-anti-dependencies are treated similarly to the previous case, except that r_i may also be scheduled in parallel to r_j.

$$r_i \overset{DAD}{\Leftrightarrow\!\!\rightarrow} r_j \quad \Rightarrow$$

$$\forall t \in rng(r_i) \cap rng(r_j) :$$

$$\sum_{m=1}^{|E_j|} v_{j,m,t} \quad \leq \quad \sum_{t' \in [ASAP(r_i), t]} \sum_{m=1}^{|E_i|} v_{i,m,t'}$$

Live values are not destroyed by side effects: A value in a single register X_s is live in all control steps between its production and consumption, so that a NOP must be activated for X_s in these control steps. In contrast to more rigid handling of side effects in previous work [5], we permit to *tolerate* side effects, i.e., NOPs for registers are activated only if two data-dependent RTs are not scheduled in consecutive control steps.

Conversely, we *enforce* to schedule these RTs consecutively, if no NOP for the corresponding destination register exists. This is modelled by the following constraints:

$$W_i = X_s \quad \wedge \quad r_i \overset{DD}{\Leftrightarrow} r_j \quad \Rightarrow$$

$$\forall t \in \underbrace{[ASAP(r_i)+1, ALAP(r_j) - 1]}_{=:R(i,j)} :$$

$$\sum_{t' \in R(i,j) | t' < t} \sum_{m=1}^{|E_i|} v_{i,m,t'} \quad +$$

$$\sum_{t' \in R(i,j) | t' > t} \sum_{m=1}^{|E_j|} v_{j,m,t'} \quad -1 \quad \leq \quad n_{s,t}$$

The left hand side of the inequation becomes 1, exactly if r_i is scheduled before t, and r_j is scheduled after t. In this case, a NOP version for X_s must be activated in CS_t. If no NOP is present for register X_s, then $n_{s,t}$ is replaced by zero. This mechanism is useful for *single* registers. Tolerating side effects for (addressable) storage elements is only possible, if N-variables are introduced for *each element* of the file, because the different elements must be distinguished. However, this would imply an intolerable explosion of the number of IP solution variables. Instead, as mentioned earlier, we assume that a NOP is present for each addressable storage element.

Compatibility restrictions are not violated: Two RTs r_i, r_j have a potential conflict, if they have at least one pair of conflicting versions, they have non-disjoint mobility ranges, and they are neither data-dependent nor output-dependent. The following constraints ensure, that at most one of two conflicting versions is scheduled in each control step CS_t.

$$\forall r_i, r_j, \quad (r_i, r_j) \notin DD \cup OD : \quad \forall t \in rng(r_i) \cap rng(r_j) :$$

$$\forall b_{im} \in E_i : \quad \forall b_{jm'} \in E_j :$$

$$b_{im} \not\sim b_{jm'} \quad \Rightarrow \quad v_{i,m,t} + v_{j,m',t} \leq 1$$

C. Search space reduction

The IP model of a given compaction problem can be easily constructed from the corresponding RT dependency graph and the set of partial instructions. If the IP has a solution, then the actual schedule can be immediately derived from the V-variables, which are set to 1. These settings account for the detailed control step assignment and selected encoding version for each RT. Based on this scheduling information, NOP versions are packed into each control step by means of version shuffling: If a control step CS_t demands for a NOP on register X_s, as indicated by the setting of N-variables, then a NOP version $nop_{js} \in NOP_s$ is determined, which is compatible to all RTs assigned to CS_t. Existence of this version is guaranteed by the above assumptions A1 and A2. If X_s is an addressable storage, then a NOP version is scheduled in each control step, in which X_s is not written. This is done independently of the setting of N-variables.

The computation times both for IP generation and NOP version shuffling are negligible. However, it is important to keep the number of solution variables as low as possible in order to reduce the runtime requirements for IP solving. The number of solution variables can be reduced by discarding *redundant* variables, which do not contribute to the solution space. Obviously, N-variables not occurring in live value constraints are superfluous and can be eliminated. V-variables are redundant, if they do not *potentially* increase parallelism, which can be efficiently checked in advance: Selecting encoding version e_{im} of some RT r_i for control step CS_t is useful, only if there exists an RT r_j, which could be scheduled in parallel, i.e., r_j meets the following conditions:

1. $t \in rng(r_j)$
2. $(r_i, r_j) \notin DD \cup OD$
3. r_j has a version $e_{jm'}$ compatible to e_{im}

If no such r_j exists, then all variables $v_{i,m,t}$ are, for all m, equivalent in terms of parallelism, and it is sufficient to keep only single, arbitrary representative. Further advance pruning of the search space can be achieved by computing tighter mobility ranges through application of some ad hoc rules. For instance, two RTs r_i, r_j, with $r_i \overset{DAD}{\Leftrightarrow} r_j$, cannot be scheduled in parallel, if all encoding versions for r_i and r_j are pairwise conflicting. The efficacy of such ad hoc rules is however strongly processor-dependent.

VI. Examples and results

A. TMS320C25

As a first example, we consider code generation for the TMS320C25 DSP, while also focussing on the interaction of code compaction and preceding code generation phases. The TMS320C25 shows a very restrictive type of instruction-level parallelism, making compaction a non-trivial task even for small programs. We demonstrate exploitation of potential parallelism for the `complex_multiply` program taken from the DSPStone benchmark suite [2], which computes the product of two complex numbers and consists of two lines of code:

```
cr  = ar * br - ai * bi ;
ci  = ar * bi + ai * br ;
```

The vertical code generated by code selection, register allocation, and scheduling is shown in figure 2. The real and imaginary parts are computed sequentially, employing registers **TR**, **PR**, and **ACCU**, and the results are stored in memory. The next step is insertion of RTs for *memory addressing*. Record makes use of indirect addressing capabilities of DSPs, based on a generic model of *address generation units* (AGUs). Based on the variables access sequence in the basic block, a permutation of variables to memory cells is computed, which maximizes AGU utilization in form of auto-increment/decrement operation of address registers [20]. For `complex_multiply`, the computed

```
(1)  TR = MEM[ar]
 // TR = ar
(2)  PR = TR * MEM[br]
 // PR = ar * br
(3)  ACCU = PR
 // ACCU = ar* br
(4)  TR = MEM[ai]
 // TR = ai
(5)  PR = TR * MEM[bi]
 // PR = ai * bi
(6)  ACCU = ACCU - PR
 // ACCU = ar * br - ai * bi
(7)  MEM[cr] = ACCU
 // cr = ar * br - ai * bi
(8)  TR = MEM[ar]
 // TR = ar
(9)  PR = TR * MEM[bi]
 // PR = ar * bi
(10) ACCU = PR
 // ACCU = ar * bi
(11) TR = MEM[ai]
 // TR = ai
(12) PR = TR * MEM[br]
 // PR = ai * br
(13) ACCU = ACCU + PR
 // ACCU = ar * bi + ai * br
(14) MEM[ci] = ACCU
 // ci = ar * bi + ai * br
```

Fig. 2. Vertical code for `complex_multiply` program

address assignment is

$$\texttt{MEM[0]} \leftrightarrow \texttt{ci},\quad \texttt{MEM[1]} \leftrightarrow \texttt{br},\quad \texttt{MEM[2]} \leftrightarrow \texttt{ai},$$

$$\texttt{MEM[3]} \leftrightarrow \texttt{bi},\quad \texttt{MEM[4]} \leftrightarrow \texttt{cr},\quad \texttt{MEM[5]} \leftrightarrow \texttt{ar}$$

After insertion of AGU operations, the vertical code consists of 25 RTs, as shown in figure 3. The TMS320C25 has eight address registers, which in turn are addressed by address register pointer `ARP`. In this case, only address register `AR[0]` is used. The optimized address assignment ensures, that most address register updates are realized by auto-increment/decrement operations on `AR[0]`.

The critical path length L_c imposed by inter-RT dependencies is 15. Table I shows experimental data (CPU seconds[1], number of V- and N-variables) for IP-based compaction of the `complex_multiply` code for $T_{\max}$ values in $[15, 21]$. For the theoretical lower bound $T_{\max} = 15$, no solution exists, while for $T_{\max} = 16$ (the actual lower bound) a schedule is constructed in less that 1 CPU second. Beyond $T_{\max} = 18$, the CPU time rises to minutes, due to the large search space that has to be investigated by the IP solver. This infavorable effect is inherent to any IP-based formulation of a time-constrained scheduling problem: The computation time may dramatically grow with

[1]Integer Programs have been solved with IBM's *Optimization Subroutine Library* (OSL) V1.2 on an IBM RISC System 6000.

```
(1)   ARP = 0                 // init AR pointer
(2)   AR[0] = 5               // point to 5 (ar)
(3)   TR = MEM[AR[ARP]]
(4)   AR[ARP] -= 4            // point to 1 (br)
(5)   PR = TR * MEM[AR[ARP]]
(6)   ACCU = PR
(7)   AR[ARP] ++              // point to 2 (ai)
(8)   TR = MEM[AR[ARP]]
(9)   AR[ARP] ++              // point to 3 (bi)
(10)  PR = TR * MEM[AR[ARP]]
(11)  ACCU = ACCU - PR
(12)  AR[ARP] ++              // point to 4 (cr)
(13)  MEM[AR[ARP]] = ACCU
(14)  AR[ARP] ++              // point to 5 (ar)
(15)  TR = MEM[AR[ARP]]
(16)  AR[ARP] -= 2            // point to 3 (bi)
(17)  PR = TR * MEM[AR[ARP]]
(18)  ACCU = PR
(19)  AR[ARP] --              // point to 2 (ai)
(20)  TR = MEM[AR[ARP]]
(21)  AR[ARP] --              // point to 1 (br)
(22)  PR = TR * MEM[AR[ARP]]
(23)  ACCU = ACCU + PR
(24)  AR[ARP] --              // point to 0 (ci)
(25)  MEM[AR[ARP]] = ACCU
```

Fig. 3. Vertical code for `complex_multiply` program after address assignment

TABLE I
EXPERIMENTAL RESULTS FOR IP-BASED COMPACTION OF `complex_multiply` TMS320C25 CODE

$T_{\max}$	CPU	solution	# V-vars	# N-vars
15	8.39	no	71	23
16	0.75	yes	141	44
17	1.26	yes	211	56
18	22	yes	281	64
19	119	yes	351	72
20	164	yes	421	79
21	417	yes	491	84

the number of control steps, even though the scheduling problem intuitively gets easier. Therefore, it is favorable to choose relatively tight time constraints, i.e. close to the actual lower bound. For tight time constraints, IP-based compaction produces extremely compact code within acceptable amounts of computation time: Figure 4 shows the parallel schedule constructed for $T_{\max} = 16$. Both compilation by the TMS320C25-specific C compiler and manual assembly programming did not yield higher code quality in the DSPStone project [2].

B. Motorola DSP56k

IP-based code compaction is not specific to code generation techniques used in RECORD, but can essentially

```
(1)         ARP = 0                    // LARP 0
(2)         AR[0] = 5                  // LARK AR0,5
(3)         TR = MEM[AR[ARP]]          // LT *
(4)         AR[ARP] -= 4               // SBRK 4
(5,7)       PR = TR * MEM[AR[ARP]] // MPY *+
            || AR[ARP] ++
(6,8,9)     ACCU = PR                  // LTP *+
            || TR = MEM[AR[ARP]]
            || AR[ARP] ++
(10,12)     PR = TR * MEM[AR[ARP]] // MPY *+
            || AR[ARP] ++
(11)        ACCU = ACCU - PR           // SPAC
(13,14)     MEM[AR[ARP]] = ACCU        // SACL *+
            || AR[ARP] ++
(15)        TR = MEM[AR[ARP]]          // LT *
(16)        AR[ARP] -= 2               // SBRK 2
(17,19)     PR = TR * MEM[AR[ARP]] // MPY *-
            || AR[ARP] --
(18,20,21) ACCU = PR                   // LTP *-
            || TR = MEM[AR[ARP]]
            || AR[ARP] --
(22,24)     PR = TR * MEM[AR[ARP]] // MPY *-
            || AR[ARP] --
(23)        ACCU = ACCU + PR           // APAC
(25)        MEM[AR[ARP]] = ACCU        // SACL *
```

Fig. 4. Parallel TMS320C25 code for `complex_multiply`

be applied to any piece of vertical machine code for processors satisfying our instruction-set model. As a second example, we consider compaction of Motorola 56000 code generated by the GNU C compiler `gcc` [33]. Compared to the TMS320C2x, the M56000 has a more regular instruction set, and parallelization of arithmetic operations and data moves is hardly restricted by encoding conflicts. As a consequence, the number of IP solution variables and the required computation time grow less strongly with the number of control steps. In table II, this is exemplified for an RTL basic block of length 23, extracted from an MPEG audio decoder program. The critical path length in this case is 14, with an actual lower bound of 19 control steps. Experimental results for three further blocks are given in table III, which indicate that exact compaction may save a significant percentage of instructions compared to purely vertical compiler-generated code. Note that in case of the M56000 a higher exploitation of parallelism can be achieved by late assignment of variables to different memory banks during compaction [28], which is not yet included in our approach.

C. Horizontal instruction formats

The permissible basic block length for IP-based compaction is inversely related to the degree of instruction encoding: For weakly encoded formats, compaction constraints are mainly induced by inter-RT dependencies, so that the critical path length is close to the actual lower

TABLE II
COMPACTION RUNTIMES FOR `gcc`-GENERATED M56000 MACHINE CODE IN RELATION TO $T_{\max}$

$T_{\max}$	CPU	solution	# V-vars	# N-vars
14	0.28	no	84	38
15	0.36	no	107	56
16	0.50	no	130	68
17	1.40	no	153	74
18	3.75	no	176	80
19	20	yes	199	86
20	22	yes	222	92
21	33	yes	245	98
22	67	yes	268	104
23	107	yes	291	110

TABLE III
EXPERIMENTAL RESULTS FOR COMPACTION OF M56000 MACHINE CODE

BB	# RTs	L_c	opt.	# V-vars	# N-vars	CPU
1	8	5	7	32	9	0.33
2	23	14	17	131	41	6.63
3	37	20	25	412	110	130

bound for the schedule length. In this case, runtime requirements are low even for larger blocks, if tight time constraints are chosen. On the other hand, weakly encoded instruction formats represent the worst case in presence of loose time constraints. This is due to the fact, that encoding constraints permit early pruning of the search space explored by the IP solver, whenever RTs have large mobility ranges. In table IV, this is demonstrated for an audio signal processing ASIP with a purely horizontal instruction format. Experimental results are given for a sum-of-products computation consisting of 45 RTs, including AGU operations, with a critical path length of 14. The ASIP executes up to 4 RTs per machine cycle, so that the actual lower bound meets the theoretical limit. For $T_{\max} \in [14, 17]$, schedules are computed very fast. Beyond $T_{\max} = 17$, runtimes are much higher and also less predictable than in the previous experiments.

In summary, our results indicate, that complex standard DSPs, such as TMS320C2x and M56000, represent the upper bound of processor complexity, for which IP-based compaction is reasonable. For these processors, blocks of small to medium size can be compacted within amounts of computation time, which may be often acceptable in the area of code generation for embedded DSPs. For ASIPs, which (because of lower combinational delay and design effort) tend to have a weakly encoded instruction format, also larger blocks can be compacted, however with higher sensitivity towards specification of time constraints. In the context of retargetable code generation, these limitations are compensated by high flexibility of our approach: Due

TABLE IV
Compaction runtimes for an ASIP with horizontal instruction format in relation to $T_{\max}$

$T_{\max}$	CPU	# V-variables	# N-variables
14	0.14	46	1
15	0.32	89	44
16	0.53	132	48
17	2.15	175	52
18	120	218	56
19	11	261	60
20	75	304	64
21	26	347	68
22	85	390	72
23	76	433	76
24	20	476	80

to a general definition of the compaction problem, our IP formulation immediately applies to complex DSP instruction sets, for which exact compaction techniques have not been reported so far. As indicated by previous work [30], significant runtime reductions can be expected for more restricted classes of instruction formats.

VII. Conclusions

Future system-level CAD environments will need to incorporate code generation tools for embedded processors, including DSPs, in order to support hardware/software codesign of VLSI systems. While satisfactory compilers are available for general-purpose processors, this is not yet the case for DSPs. Partially, this is due to missing dedicated DSP programming languages, which causes a mismatch between high-level language programs and DSP architectures. In contrast to general-purpose computing, compilation speed is no longer a primary goal in DSP code generation. Therefore, the largest boost in DSP compiler technology can be expected from new code optimization techniques, which – at the expense of high compilation times – explore comparatively vast search spaces during code generation. Also retargetability will become an increasingly important issue, because the diversity of application-specific processors creates a strong demand for flexible code generation techniques, which can be quickly adapted to new target architectures.

In this contribution, we have motivated and described a novel approach to thorough exploitation of potential parallelism in DSP programs. The proposed IP formulation of local code compaction as a time-constrained problem is based on a problem definition designated to DSPs, which removes several limitations of previous work. Our approach applies to very high quality code generation for standard DSPs and ASIPs, and is capable of optimally compacting basic blocks of relevant size. Due to a general problem definition, peculiarities such as alternative encoding versions and side effects are captured, which provides retargetability within a large class of instruction formats. Since existing solutions are guaranteed to be found, we believe that exact code compaction is a feasible alternative to heuristic techniques in presence of very high code quality requirements.

Further research is necessary to extend time-constrained code generation towards *global* constraints, for which local techniques may serve as subroutines. In turn, this demands for closer *coupling* of code compaction and the preceding code selection, register allocation, and scheduling phases. Also the mutual dependence between retargetability, code quality, and compilation speed should be studied in more detail in order to identify feasible compromises.

Acknowledgments

The authors would like to thank Birger Landwehr for helpful comments on Integer Programming issues and Steven Bashford for careful reading of the manuscript. This work has been partially supported by the European Union through ESPRIT project 9138 (CHIPS).

References

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

[2] 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.

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

[4] L. Nowak, P. Marwedel: *Verification of Hardware Descriptions by Retargetable Code Generation*, 26th Design Automation Conference (DAC), 1989, pp. 441-447.

[5] P. Marwedel, *Tree-based Mapping of Algorithms to Predefined Structures*, Int. Conf. on Computer-Aided Design (ICCAD), 1993, pp. 586-993.

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

[7] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers - Priciples, Techniques, and Tools*, Addison-Wesley, 1986.

[8] S. Bashford, U. Bieker, et al., *The MIMOLA Language V 4.1*, Technical Report, University of Dortmund, Dept. of Computer Science, September 1994.

[9] R. Leupers, P. Marwedel: *Instruction Set Extraction from Programmable Structures*, European Design Automation Conference (EURO-DAC), 1994, pp. 156-161.

[10] R. Leupers, P. Marwedel, *A BDD-based Frontend for Retargetable Compilers*, European Design & Test Conference (ED & TC), 1995, pp. 239-243.

[11] D. Lanneer, J. Van Praet, et al., *CHESS: Retargetable Code Generation for Embedded DSP Processors*, chapter 5 in [3].

[12] P. Paulin, C. Liem, et al., *FlexWare: A Flexible Firmware Development Environment for Embedded Systems*, chapter 4 in [3].

[13] R.E. Bryant, *Symbolic Manipulation of Boolean Functions Using a Graphical Representation*, 22nd Design Automation Conference (DAC), 1985, pp. 688-694.

[14] 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.

[15] B. Wess, *Automatic Instruction Code Generation based on Trellis Diagrams*, IEEE Int. Symp. on Circuits and Systems (ISCAS), 1992, pp. 645-648.

[16] 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.

[17] H. Emmelmann, *Code Selection by Regular Controlled Term Rewriting*, in: R. Giegerich, S. Graham, *Code Generation: Concepts, Tools, Techniques*, Springer, 1992, pp. 3-29.

[18] E. Pelegri-Llopart, S. Graham, *Optimal Code Generation for Expression Trees*, 15th Ann. ACM Symp. on Priciples of Programming Languages, 1988, pp. 294-308.

[19] S. Liao, S. Devadas, K. Keutzer, et al., *Storage Assignment to Decrease Code Size*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995.

[20] R. Leupers, P. Marwedel, *Algorithms for Address Assignment in DSP Code Generation*, Int. Conf. on Computer-Aided Design (ICCAD), 1996.

[21] J.A. Fisher, *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Trans. on Computers, vol. 30, no. 7, 1981, pp. 478-490.

[22] A. Aiken, A. Nicolau, *A Development Environment for Horizontal Microcode*, IEEE Trans. on Software Engineering, no. 14, 1988, pp. 584-594.

[23] S. Novack, A. Nicolau, N. Dutt, *A Unified Code Generation Approach using Mutation Scheduling*, chapter 12 in [3].

[24] M.R. Gary, D.S. Johnson, *Computers and Intractability – A Guide to the Theory of NP-Completeness*, Freemann, 1979.

[25] S. Mallett, D. Landskov, B.D. Shriver, P.W. Mallett, *Some Experiments in Local Microcode Compaction for Horizontal Machines*, IEEE Trans. on Computers, vol. 30, no. 7, 1981, pp. 460-477.

[26] Texas Instruments, *TMS320C2x User's Guide*, rev. B, Texas Instruments, 1990.

[27] Motorola Inc., *DSP 56156 Digital Signal Processor User's Manual*, Motorola, 1992.

[28] 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.

[29] 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.

[30] 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.

[31] C. Gebotys, M. Elmasry, *Optimal VLSI Architectural Synthesis*, Kluwer Academic Publishers, 1992.

[32] B. Landwehr, P. Marwedel, R. Dömer, *OSCAR: Optimum Simultaneous Scheduling, Allocation, and Resource Binding based on Integer Programming*, European Design Automation Conference (EURO-DAC), 1994.

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

Rainer Leupers, born in 1967, holds a Diploma degree (with distinction) in Computer Science from the University of Dortmund, Germany. He received a scholarship from Siemens AG and the Hans Uhde award for an outstanding Diploma thesis. Since 1993, he is with Prof. Peter Marwedel's VLSI CAD group at the Computer Science Department of the University of Dortmund, where he is currently working towards a Ph.D. degree. His focus of research interest is on system-level design automation for embedded systems, in particular modelling of embedded programmable components and code generation for digital signal processors.

Peter Marwedel (M'79) received his Ph.D. in Physics from the University of Kiel (Germany) in 1974. He worked at the Computer Science Department of that University from 1974 until 1989. In 1987, he received the Dr. habil. degree (a degree required for becoming a professor) for his work on high-level synthesis and retargetable code generation based on the hardware description language MIMOLA. Since 1989 he is a professor at the Computer Science Department of the University of Dortmund (Germany). He served as the Dean of that Department between 1992 and 1995. His current research areas include hardware/software codesign, high-level test generation, high-level synthesis and code generation for embedded processors. Dr. Marwedel is a member of the IEEE Computer society, the ACM, and the Gesellschaft für Informatik (GI).