Time-constrained Code Compaction for DSPs

Rainer Leupers, Peter Marwedel
University of Dortmund, Dept. of Computer Science XII, 44221 Dortmund, Germany
email: *leupers|marwedel@ls12.informatik.uni-dortmund.de*

Abstract–DSP algorithms in most cases are subject to hard real-time constraints. In case of programmable DSP processors, meeting those constraints must be ensured by appropriate code generation techniques. For processors offering instruction-level parallelism, the task of code generation includes code compaction. The exact timing behavior of a DSP program is only known after compaction. Therefore, real-time constraints should be taken into account during the compaction phase. While most known DSP code generators rely on rigid heuristics for that phase, this paper proposes a novel approach to local code compaction based on an Integer Programming model, which obeys exact timing constraints. Due to a general problem formulation, the model also obeys encoding restrictions and possible side effects. [1]

1 Introduction & related work

Design requirements for embedded systems including DSP functionality strongly differ from those for interactive environments such as workstations. While in the latter case an "as fast as possible" behavior is desirable, DSP algorithms (e.g. in audio and video processing) are usually subject to hard real-time constraints, i.e. any speed overhead violating the restrictions is unacceptable for those systems. On the other hand, it is unnecessary to optimize a DSP algorithm beyond the given timing constraint.
This has consequences for code generation in case of programmable DSP processors. Instead of producing highly optimized code, a DSP code generator should basically answer the question whether there exists a machine program whose execution does not take more than T cycles for a given constraint T, and if so, construct that program.

Code generation for DSPs is complicated by the fact that a moderate to high degree of potential parallelism is offered by contemporary DSP instruction sets. The Motorola DSP56156 [1] for instance performs up to three register transfers per cycle. Exploitation of available parallelism during code generation is usually ensured by a *code compaction* phase, in which independent register transfers may be scheduled together into a single control step thus resulting in a lower cycle count. The exact timing behavior of a machine program is *only known after compaction*. This implies that timing constraints should be considered by a compiler *at least* during the compaction phase.

Due to the steadily growing importance of embedded DSP systems based on programmable processors, much research effort has gone into the area of code generation for DSPs during the last years. Compilers for DSPs have to cope with highly irregular datapaths, highly specialized instruction sets, and peculiarities in the instruction formats. Furthermore, a certain degree of *retargetability* is desirable. Recent approaches to DSP code generation include [2, 3, 4, 5, 6]. An overview of the state-of-the-art is to be found in [7].
In those approaches, however, timing constraints are neglected. Instead, focus is on retargetability and code optimization. Due to the problem complexity, heuristics are applied for solving the subtasks of code generation, including code compaction.

While the presence of timing constraints has been subject to research for a long time in the area of *hardware synthesis*, automatic time-constrained software generation is quite a new topic. In contrast to hardware synthesis, time-constrained software generation for predefined processors results in a both time and resource-constrained problem definition. This implies that a given problem does not necessarily have a solution.

The problem of time-constrained software generation has also been addressed in the context of HW/SW codesign. The technique described in [8] uses a very rough estimation of machine program execution times, making simplifying assumptions about the available instruction set. While such an approach makes sense from a "system-level" point of view, it is unlikely to succeed in actual code generation. A constructive system-level technique for software scheduling in presence of real-time constraints has been reported in [9]. In Timmer's approach [10], both time and resource constraints are exploited during code generation. Due to a sophisticated *execution interval analysis*, the technique efficiently produces very high quality code. However, code generation

[1]This work has been partially supported by the European Union, ESPRIT project 9138 (CHIPS). Publication: 8th International Symposium on System Synthesis, Cannes/France, Sept. 1995, ©ACM

is currently limited towards restrictive instruction formats. A more versatile code generation system that actually considers exact program execution times on a given processor was presented in [11]. In that system, code generation is based on an Integer Program (IP) formulation of the problem, a technique that has also become quite popular in the area of high-level synthesis (see e.g. [12, 13]). In fact, the approach described in [11] tries to integrate several of the code generation subtasks (including code compaction) into a single Integer Program resulting in optimal programs or time-constrained programs, respectively. However, due to NP-completeness of Integer Programming, one cannot expect to solve complex code generation problems within an acceptable amount of time, although it is reasonable to permit much higher runtimes for a DSP code generator than for a C compiler in a workstation environment.
Therefore, in this contribution we propose an IP model which focuses on the task of code compaction. Inputs to the model are a sequence of register transfers and a maximum time budget T for the compacted code. Running an IP solver on this model either delivers compacted code with a cycle count $\leq T$ or reports infeasibility of the given problem. We do not explicitly consider *minimum* timing constraints, since in code generation those constraints usually can be met by inserting "no-operations".

In extension to the work described in [11] which basically considers resource conflicts and dependencies between microoperations during compaction, our model also handles *encoding restrictions* and operations having *side effects*, i.e. the control code for one register transfer may also trigger other different register transfers. During code compaction it must be ensured that live values in registers are not destroyed by such side effects. Furthermore, we include the concept of *alternative code versions* into compaction. During compaction, an appropriate code version for each operation is selected. These extensions make our approach suitable for a variety of DSP processors. Although the problem to solve remains NP-hard, we believe that an approach which directly incorporates timing constraints into the code generation process is better suited for DSP requirements, since any rigid heuristic trying to optimize the code might fail to find an existing solution.

The next section gives a detailed definition of the code compaction problem. Section 3 shows how one problem instance can be transformed into an IP. Experimental results are given in section 4, and the paper ends with some concluding remarks.

2 Problem definition

As in [11] we concentrate on *local* code compaction, i.e. within basic blocks. While on one hand this restricts the solution space, it can be argued that the possible advantage of tackling the problem in a more global fashion (at the expense of a possibly much larger search space) might not be too high, since DSP algorithms typically show mainly data flow and less control flow behavior.

With the appearance of VLIW machines, local microcode compaction became a popular research topic, and a number of heuristic algorithms have been developed, since the problem was shown to be NP-complete. An extensive experimental study [14] revealed that some of these heuristics very often find solutions close to the optimum within polynomial time. In the area of DSP, however, code optimality is not always necessary. Any machine program that satisfies a given timing constraint is a valid solution. Our problem definition therefore just demands for such a valid solution.

We assume that a sequence of *assignments*

$$SA = (A_1, \ldots, A_n)$$

has been generated by earlier phases of compilation. The goal is to schedule the assignments into a control step list that does not exceed a given length. An assignment A_i is a pair

$$A_i = (W_i, R_i)$$

where W_i is the *write location* and R_i is a set of *read locations* for the assignment. Write and read locations are registers or memory cells. An assignment A_i writes a value to W_i that is a function of R_i. Let $A = \{A_1, \ldots, A_n\}$. Three relations on $A \times A$ are important for preserving correctness in the compacted assignment sequence. For $j > i$ we define:

1. $(A_i, A_j) \in DD$ (data-dependency)
   $:\Longleftrightarrow W_i \in R_j$.
2. $(A_i, A_j) \in OD$ (output-dependency)
   $:\Longleftrightarrow W_i = W_j$.
3. $(A_i, A_j) \in DAD$ (data-anti-dependency)
   $:\Longleftrightarrow W_j \in R_i$.

In case that read or write locations are cells of an addressable memory, the problem of *ambiguous memory references* occurs. Sometimes it is undecidable at compile time whether or not two memory accesses refer to the same address. We therefore assume that the three relations incorporate *potential depencencies* in case of unresolvable ambiguities.

Let $CS(A_i)$ denote the control step to which A_i will be assigned during compaction. Then, the following *dependency constraints* have to be satisfied in any valid schedule:

1. $\forall (A_i, A_j) \in DD \cup OD : CS(A_i) < CS(A_j)$.
2. $\forall (A_i, A_j) \in DAD : CS(A_i) \leq CS(A_j)$.

For each A_i there is a set of *alternative versions*

$$V_i = \{v_{i1}, \dots, v_{im_i}\}$$

A version v_{ij} is a *partial control word setting*, i.e. a bitstring $B \in \{0, 1, x\}^c$, where c is the control word length. One of these versions has to be selected for each A_i. Two assignments A_i, A_j may only be scheduled within the same control step with versions v_{ik}, v_{jl}, if the versions are *bit-compatible*, i.e. there is no position m in the bitstrings such that

$$v_{ik}[m] = 1 \quad \wedge \quad v_{jl}[m] = 0$$

and vice versa. The concept of *versions* is also used in the MSSQ code generator [15]. Considering only control code requirements for an assignment allows for *mapping resource conflicts to instruction conflicts*. As a consequence, no explicit information about resource usage of assignments has to be maintained. Furthermore, versions also account for *encoding restrictions* within the processor controller: In order to keep the instruction word length small, it is often the case that register transfers cannot take place in parallel although being resource-compatible. Like resource conflicts, encoding restrictions are implicitly represented by versions. The presence of *alternative* versions for assignments reflects the fact, that in general there exist several implementations for a given assignment, each having different control code requirements. The example in section 4 demonstrates that taking into account those alternatives during compaction is essential for obtaining acceptable results.

Preserving the semantical correctness of an assignment sequence during compaction in general is further complicated by the presence of *side effects* of register transfers. As an example we consider the TMS320C25 DSP [16]. A possible register transfer in the TMS320C25 is to multiply register `TR` with a data memory value and to store the result in register `PR`. However, there exist different versions to perform this operation (table 1). The multiply (MPY) instruction does just the multiplication. The multiply-accumulate instruction (MAC) additionally accumulates the previous product, causing a side effect. Whenever the accumulator has to retain its value, it must be ensured that during compaction the MPY version is selected, while in other cases it would be favorable to select the MAC version.

Especially in code generation for VLIW-like processors, usually some bits in each control word remain "don't care". However, it must be ensured that a later setting of these don't care bits to either 0 or 1 does not trigger an undesired register transfer that destroys a live value. In order to eliminate such undesired side effects during compaction, we assume that with every write location w (register or memory) there is an associated set of "NOP versions"

$$N_w = \{n_{w1}, \dots, n_{wk_w}\}$$

also being bitstrings in $\{0, 1, x\}^c$. Packing a NOP version n_{wl} into a control step t ensures that location w remains its state during that step. Although not being "real instructions", NOP versions for a location w are quite easy to obtain by inverting the sum of all versions that write to w. Reference [17] describes how all versions can be extracted from a processor model given in an HDL.
For elimination of undesired side effects during compaction, any valid schedule must fulfill the following *state preserving constraints*:

Register states: If an assignment A_i writes a value to a register R, it must be ensured that no other operation destroys that value during its life range. On the other hand, in order to avoid unnecessary restrictions, the register *may* be overwritten as a side effect, whenever it does not contain a live value. Thus, for each pair (A_i, A_j) of data-dependent assignments, which are scheduled in non-subsequent control steps t_i, t_j, a NOP version for the destination of A_i must be scheduled.

Memory states: If an assignment A_i writes a value to a cell of an addressable memory M, the memory must not be disabled during the lifetime of that value, since this would prevent intermediate write accesses to other cells of M. The dependency relations ensure that the live value cannot be overwritten by other assignments. In order to prevent undesired side effects, a NOP version for M must be activated for each control step in which no write access to M takes place.

We can now define the problem of *time-constrained code compaction*:
For an assignment sequence $SA = \{A_1, \dots, A_n\}$ and a timing constraint T find a *valid schedule* of length $\leq T$. A schedule is valid, if:

1. Each assignment is scheduled exactly once.
2. For each control step, the dependency, bit-compatibility, and state preserving constraints are satisfied.

Obviously, time-constrained code compaction is NP-hard. Otherwise, one could solve the problem of optimum code

machine instruction	partial code	functionality
MPY	00111000xxxxxxxx	PR := TR * mem[...]
MAC	01011101xxxxxxxx	PR := TR * mem[...]
		ACCU := ACCU + PR

Table 1: Different versions for multiplication

compaction in polynomial time by time-constrained code compaction combined with a binary search on the possible schedule lengths. Therefore we may map the problem to an Integer Program formulation without loss of efficiency.

3 IP formulation

Given an assignment sequence $A = (A_1, \ldots, A_n)$ and a maximum timing constraint T, the IP model contains two classes of decision variables:

Version variables:
$\forall i \leq n, \quad j \leq |V_i|, \quad t \in R(A_i) : v_{i,j,t} = 1 :\Longleftrightarrow$ version j of assignment i is scheduled in control step t. $R(A_i)$ denotes the *range* of A_i, i.e. the interval

$$[ASAP(A_i), ALAP(A_i)]$$

induced by the dependency relations in accordance with T. We assume that each A_i is executable within a single machine cycle.
Any maximum timing constraint T less than the maximum ASAP value can be rejected in advance, since no valid schedule can have a smaller length than the critical path. Likewise, timing constraints greater than n need not be considered, since for $T \geq n$ the original sequence can simply be kept.

NOP variables:
$n_{w,j,t} = 1 :\Longleftrightarrow$ NOP version j is activated for write location w in control step t. NOP variables are only defined in case of occurence in the state preserving constraints.

The constraints guaranteeing semantical correctness of the compacted sequence are the following:

Each assignment scheduled once:

$$\forall A_i : \sum_{t \in R(A_i)} \sum_{j=1}^{|V_i|} v_{i,j,t} = 1$$

For each A_i one version must be selected and be assigned to one control step within the range of A_i.

Strong dependency constraints:

$$\forall \quad (A_i, A_j) \in DD \cup OD, \quad C := R(A_i) \cap R(A_j) : \forall t \in C :$$

$$\sum_{k=1}^{|V_j|} v_{j,k,t} \leq \sum_{t' < t, t' \in C} \sum_{k=1}^{|V_i|} v_{i,k,t'}$$

Assignments A_i have to be scheduled before their data- and output-dependent assignments A_j.

Weak dependency constraints:

$$\forall \quad (A_i, A_j) \in DAD, \quad C := R(A_i) \cap R(A_j) : \quad \forall t \in C :$$

$$\sum_{k=1}^{|V_j|} v_{j,k,t} \leq \sum_{t' \leq t, t' \in C} \sum_{k=1}^{|V_i|} v_{i,k,t'}$$

Assignments A_i have to be scheduled before or in the same control step as their data-anti-dependent assignments A_j.

Register state preserving constraints:

$$\forall \quad (A_i, A_j) \in DD, \quad W_i = (\text{register})R :$$

$$\forall \quad t \in [ASAP(i) + 1, ALAP(j) - 1] :$$

$$\sum_{t' < t} \sum_{k=1}^{|V_i|} v_{i,k,t'} + \sum_{t' > t} \sum_{k=1}^{|V_j|} v_{j,k,t'} \quad - 1 \quad = \quad \sum_{k=1}^{|N_R|} n_{R,k,t}$$

If A_i and A_j are data-dependent and are not scheduled in subsequent control steps, a NOP version for $W_i = R$ must be activated for each intermediate control step.

Memory state preserving constraints:

$$\forall W_i = (\text{memory})M, \quad A_M := \{A_j : W_j = M\}, \quad \forall t :$$

$$1 - \left(\sum_{A_j \in A_M, t \in R(A_j)} \sum_{k=1}^{|V_j|} v_{j,k,t} \right) = \sum_{k=1}^{|N_M|} n_{M,k,t}$$

For each memory M and each control step t, a NOP version for M is activated in t if and only if no write access to M takes place in t.

Bit-compatibility constraints:
Let "$\not\sim$" denote the bit-incompatibility of two bitstrings.

$$\forall \quad (A_i, A_j) \notin DD \cup OD, \quad C := R(A_i) \cap R(A_j) \neq \emptyset :$$

$$\forall t \in C, \forall k \in \{1, \ldots, |V_i|\}, \quad \forall k' \in \{1, \ldots, |V_j|\},$$

$$v_{i,k,t} \not\sim v_{j,k',t} : \quad v_{i,k,t} + v_{j,k',t} \leq 1$$

Two assignment versions may only be scheduled into the same control step, if they are bit-compatible, i.e. there are neither resource nor encoding conflicts. We omit the complementary constraints that hold for combinations between assignment and NOP versions, as well as between NOP versions only.

Running an IP solver without an objective function on an instance of this model either yields a compacted schedule with length $\leq T$, or proves that no such schedule exists. This implies that we solve the IP as a *decision problem* instead of an optimization problem. Although both variants show same computational complexity, the decision problem is solved faster, since only one solution has to be found.
Since the number of assignments and versions is fixed, the number of variables in the IP model is mainly influenced by the ASAP/ALAP ranges of the assignments. The assignment versions may be exploited for further reduction of those ranges without restricting the solution space. One means of such a reduction is the following rule: Whenever two assignments A_i, A_j are data-anti-dependent, and all versions for A_i, A_j are pairwise bit-incompatible, they can be treated like being strongly dependent, i.e. the data-anti-dependency induces a $<$ relation on $CS(A_i)$ and $CS(A_j)$. Apparently, the effectiveness of such reduction rules strongly depends on the instruction set of the target processor and is not further discussed here.

4 Example

Due to the NP-completeness of time-constrained code compaction and IP, respectively, the model in general cannot be applied for very large assignment sequences. Like IP models in the area of high-level synthesis, it is intended to work on small to medium size subproblems, whose solutions can be combined to a global one as proposed in [11]. For instance, a large DSP algorithm with a timing constraint T could be subdivided into smaller blocks for each of which an exact solution for a "small" timing constraint is determined. Obviously, it is still a difficult problem how to obtain an appropriate block structure, and several iterations will be required in general.

On the other hand it can be stated that especially for DSP processors focus must be on careful *local* compaction of register transfers. This is due to the application area, where many multiply-accumulate and data move operations occur. Most DSP instruction sets permit to parallelize such operations, and meeting real-time constraints in most cases demands for exploitation of potential parallelism. Again, we consider an example for the TMS320C25. The equation

$$u(n) = u(n-1) + K0 \cdot e(n) + K1 \cdot e(n-1) + K2 \cdot e(n-2)$$

is needed in a PID control loop. The following assignment sequence computes this equation (all signals are assumed to reside in data memory):

```
 (1) ACCU := u(n-1)
 (2) TR := e(n-2)
 (3) PR := TR * K2
 (4) TR := e(n-1)
 (5) e(n-2) := e(n-1)
 (6) ACCU := ACCU + PR
 (7) PR := TR * K1
 (8) TR := e(n)
 (9) e(n-1) := e(n)
(10) ACCU := ACCU + PR
(11) PR := TR * K0
(12) ACCU := ACCU + PR
(13) u(n) := ACCU
```

The cycle count without any compaction is 13. Suppose, a hard timing constraint of 9 cycles is given. The TMS320C25 instruction set comprises a number of different versions for each of the assignments (e.g. MPY and MAC for the multiplications). The compactor has to look for common versions for independent assignments in order to achieve a tighter schedule. Running the IP solver[2] on the corresponding IP formulation of the problem (containing 157 decision variables) yields the following compacted schedule within 16 seconds on a SPARC-20 (|| denotes parallel execution, NOPs not shown):

```
(1) ACCU := u(n)
(2) TR := e(n-2)
(3) PR := TR * K2
(4) e(n-2) := e(n-1) ||
    TR := e(n-1) ||
    ACCU := ACCU + PR
(5) PR := TR * K1
(6) e(n) := e(n-1) ||
    ACCU := ACCU + PR ||
    TR := e(n)
(7) PR := TR * K0
(8) ACCU := ACCU + PR
(9) u(n) := ACCU
```

In control steps (4) and (6) the compactor exploited the TMS320C25 "LTD" instruction for meeting the constraint, which loads TR, accumulates a previous product in PR, and in parallel performs a data memory move.

The relatively high runtime for this example indicates

[2] "lp_solve V1.5" by Michel Berkelaar, Eindhoven University of Technology, The Netherlands

that the IP solver has to scan a large search space in order to find the schedule. In turn, this implies that any heuristic compaction algorithm is likely to fail in this case. Therefore, in presence of timing constraints, a higher runtime is acceptable for compaction. Furthermore, the TMS320C25 is a fairly complex example. For processors with a less restrictive instruction set, solutions for blocks with more than 100 assignments have been found within a second of CPU time. Thus the acceptable assignment sequence length clearly depends on the target processor, and no quantitative remarks about the runtime behavior are possible. Due to the fact that DSP algorithms typically are short and computation-intensive, we however expect that our approach, which allows to handle sequence lengths between 10 and 100 within a reasonable amount of time, is sufficient in most cases.

5 Conclusions

With growing importance of programmable DSP processors integrated in embedded systems, time-constrained code generation has become a complementary research topic to time-constrained hardware synthesis. In this contribution we introduced a method that exactly solves a subproblem of time-constrained code generation, namely code compaction. This method is implemented in Record, a retargetable compiler for DSPs.
Due to the NP-completeness of the problem, we formulated an Integer Programming version. The IP formulation permits finding solutions for small to medium size problems within an acceptable amount of time. For complex DSP processors and tight timing constraints such an approach seems favorable, since heuristic compaction algorithms are likely to fail in those cases. Especially, this holds in presence of encoding restrictions and operations with side effects, which occur in contemporary DSPs. The presented IP model provides means of handling those peculiarities.

Further research is necessary on integration of code compaction and other phases of retargetable code generation. Furthermore, an a priori reduction of the search space as used by Timmer [10] would be favorable in order to decrease run time.

Acknowledgement
The authors would like to thank Birger Landwehr for his helpful comments on Integer Programming issues.

References

[1] DSP56156 User's Manual, Motorola Inc. 1992

[2] C. Liem, T. May, P. Paulin: Instruction-set matching and selection for DSP and ASIP code generation, European Design & Test Conference (ED & TC), 1994

[3] A. Fauth, A. Knoll: Translating signal flowcharts into microcode for custom digital signal processors, Proc. ICSP, 1993

[4] B. Wess: Optimizing signal flow graph compilers for digital signal processors, Proc. ICSPAT 1994

[5] 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, 1994

[6] R. Leupers, R. Niemann, P. Marwedel: Methods for retargetable DSP code generation, IEEE Workshop on VLSI Signal Processing, 1994

[7] P. Marwedel, G. Goossens (eds.): Code generation for embedded processors, Kluwer Academic Publishers, to appear: June 1995

[8] R. K. Gupta, G. De Micheli: Constrained software generation for hardware-software systems, 3rd Int. Workshop on Hardware/Software Codesign, 1994

[9] P. Chou, G. Borriello: Software scheduling in the co-synthesis of reactive real-time systems, 31st Design Automation Conference, 1994, pp. 1-4

[10] A. H. Timmer, M. T. J. Strik, J. L. van Meerbergen, J. A. G. Jess: Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores, 32nd Design Automation Conference, 1995

[11] T. Wilson, G. Grewal, D. K. Banerji: An integrated approach to retargetable code generation, 7th Int. Symp. on High-Level Synthesis, 1994

[12] C. H. Gebotys, M. I. Elmasry: Global optimization approach for architectural synthesis, IEEE Trans. on CAD, Vol. 12, No. 9, 1993

[13] B. Landwehr, P. Marwedel, R. Doemer: Optimum simultaneous scheduling, allocation, and resource binding based on integer programming, Euro-DAC, 1994

[14] 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

[15] L. Nowak: Graph Based Retargetable Microcode Compilation in the MIMOLA Design System, 20th Annual Microprogramming Workshop (MICRO-20), 1987, pp. 126-132

[16] TMS320C2x User's Guide, Rev. B, Texas Instruments, 1990

[17] R. Leupers, P. Marwedel: A BDD-based frontend for retargetable compilers, European Design & Test Conference (ED & TC), 1995