Instruction Selection for Embedded DSPs with Complex Instructions

Rainer Leupers, Peter Marwedel
University of Dortmund, Department of Computer Science 12
44221 Dortmund, Germany
email: *leupers|marwedel@ls12.informatik.uni-dortmund.de*

Abstract– *We address the problem of instruction selection in code generation for embedded digital signal processors. Recent work has shown that this task can be efficiently solved by tree covering with dynamic programming, even in combination with the task of register allocation. However, performing instruction selection by tree covering only does not exploit available instruction-level parallelism, for instance in form of multiply-accumulate instructions or parallel data moves. In this paper we investigate how such complex instructions may affect detection of optimal tree covers, and we present a two-phase scheme for instruction selection which exploits available instruction-level parallelism. At the expense of higher compilation time, this technique may significantly increase the code quality compared to previous work, which is demonstrated for a widespread DSP.*[1]

1 Introduction

With growing market shares of embedded systems compared to general-purpose computers, code generation for embedded processors has become an important research topic. Although compiler technology has reached a high level of maturity for general-purpose processors, it is widely agreed that code generation for embedded DSPs demands for new techniques [1, 2], because of very high code quality requirements and irregular architectures.

The task of code generation is usually subdivided into phases of instruction selection, register allocation, and scheduling. During instruction selection, pattern matching between dataflow patterns in the source algorithm and instruction patterns is performed, and those instruction patterns are selected which lead to an optimal cover with respect to a certain cost criterion. Although the dataflow in algorithms in general may show a DAG structure, instruction selection is mostly done by tree pattern matching and covering with dynamic programming [3] in order to avoid exhaustive runtimes. Decomposition of DAGs into trees may cause additional load/store instructions, but the penalty is not very high for embedded processors with on-chip memory. Furthermore, versatile tools for automatic generation of tree pattern matchers from an instruction-set model are already available, which facilitates another important issue in code generation for embedded processors: *retargetability*. Tree covering in linear time is also possible with *trellis diagrams*, as shown by Wess [6]. However, construction of trellis diagrams from more general models is more difficult than construction of tree pattern matchers.

As Araujo and Malik [4] point out, the phases of instruction selection and register allocation are not only interdependent, but virtually cannot be separated from each other in presence of irregular processor structures. Their approach integrates both phases into a single tree covering phase, and optimal code generation is reported for the TMS320C25 DSP [7]. This DSP incorporates so many peculiarities that one can expect to have techniques for a large class of embedded processors, once the problems are solved for this one. In [4], however, optimality is only achieved under the assumption that no instruction-level parallelism (ILP) is available, which does not apply to the TMS320C25.

Extension of tree pattern matching towards processors with ILP is straightforward in presence of purely horizontal instruction formats, i.e. parallel execution of register transfers (RTs) is only restricted by data dependencies and resource conflicts. In this case the RTs still can be handled as tree patterns, which may be parallelized later by *local code compaction*, for which good heuristics are known [5]. But, as program ROM size is also a critical resource for embedded processors, horizontal resp. VLIW instruction formats are the exception. In order to reduce ROM size, instructions are often strongly encoded, i.e. available ILP is reduced to a few "critical" cases only. This results in the fact, that there exist *atomic* instructions, which cannot be further subdivided, but which execute a *fixed set* of RTs in parallel. We call these *complex instructions*.

As we point out in section 2, complex instructions do not fit the dynamic programming approach. Furthermore, they exclude the use of classical, heuristic local

[1] Publication: European Design Automation Conference (EURO-DAC), Geneva/Switzerland, Sept. 1996, ©1996 IEEE

compaction algorithms, because the binary encodings must be explicitly taken into account, and generation of incorrect code due to undesired side effects must be avoided. Nevertheless, exploitation of complex instructions might be mandatory for obtaining sufficient code quality. The purpose of this paper is to outline the problems that DSP code generation faces in presence of complex instructions, and to show how a closer coupling between instruction selection and compaction leads to better exploitation of ILP in case of complex instructions. This is subject of section 3. Experimental results in section 4 demonstrate that exploitation of complex instructions can significantly improve code quality, so that the additional time needed for code generation is justified.

2 Complex instructions

The process of tree covering strongly depends on the datapath architecture of the embedded processors for which code is to be generated. We exemplify this using a 4-tap FIR filter computation given by the equation

$$y_n = c_0 \cdot x_n + c_1 \cdot x_{n-1} + c_2 \cdot x_{n-2} + c_3 \cdot x_{n-3}$$

The corresponding expression tree is depicted in fig. 1. For sake of simplicity, only the operators without load and store operations are shown. We assume that multiplications and additions are executed within a single machine cycle, which applies to most fixed-point DSPs. In order to achieve both high throughput and short

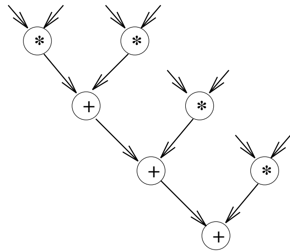

Figure 1: *Expression tree for 4-tap FIR filter*

combinational delay, DSPs often comprise a multiplier and an adder operating in a pipelined fashion (fig. 2 a). In one instruction cycle, a product is computed and stored in a pipeline register (PR), and in the next cycle the product can be accumulated and another multiplication can take place. Besides MULT and ADD instructions, the instruction set then includes a MAC (multiply-accumulate) instruction, which is a non-tree pattern, however (fig. 2 b). Thus, tree covering by dynamic programming cannot detect the optimal cover with 5 instructions shown in fig. 3 a), but yields a suboptimal cover with cost 7 (fig. 3 b). The actual reason why tree covering with dynamic programming fails in case of datapaths containing pipeline registers is that neighboring subtrees are always covered separately. In

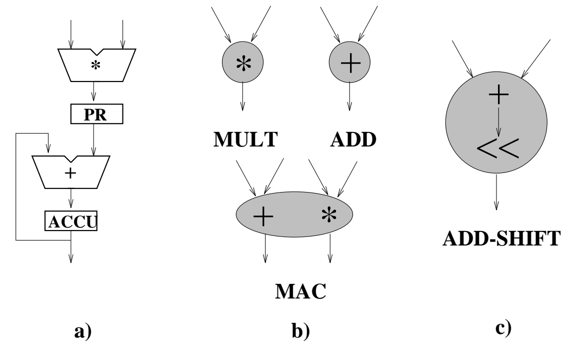

Figure 2: *a) Pipelined multiplier/accumulator, b) resulting instruction patterns, c) chained add-with-shift instruction*

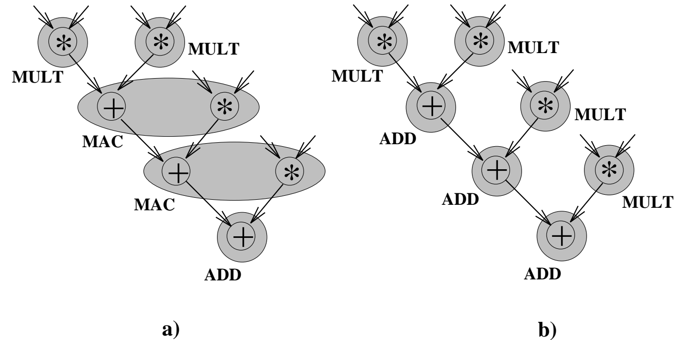

Figure 3: *a) Optimal cover for the tree in fig. 1 with complex MAC (cost: 5), b) suboptimal cover (cost: 7)*

the above example, the complex instruction MAC executes two different RTs in parallel, and therefore spans two different subtrees. It is exactly this property, which makes the difference between complex instructions and *chained instructions*, such as an add-with-shift (fig. 2 c), for which dynamic programming still finds the optimal cover.

Thus, if instruction selection is completed after tree covering, complex instructions are unnecessarily excluded from code generation and inferior code might result. Note that also heuristic local compaction techniques may fail to exploit complex instructions, when applied separately after instruction selection: In order to illustrate the problem, consider the **MPYA** (*multiply and accumulate previous product*) instruction on the TMS320C25, which performs two fixed RTs in parallel:

```
(1) PR := TR * <memory value>;
(2) ACCU := ACCU + PR;
```

MPYA is an "atomic" instruction that cannot be subdivided, so that it cannot be modelled in tree pattern matchers, where each instruction corresponds to exactly one register transfer. The two RTs performed by **MPYA** also exist as separate instructions (**MPY** and **APAC**) on the TMS320C25. While on a horizontal machine the *binary encoding* of **MPYA** would just

be the combination of **MPY** and **APAC**, which can be easily constructed by heuristic compaction, **MPYA** has to be treated as a separate, atomic instruction on the TMS320C25, due to its strongly encoded instruction format: the binary encodings of **MPY**, **APAC**, and **MPYA** are pairwise incompatible. In case instruction selection for a multiplication is required, it cannot be decided whether to use **MPY** or **MPYA**, before *binary* code generaton is performed. As a consequence, both alternatives have to be kept for code compaction in order to permit exploitation of available ILP. We therefore use a closer coupling between instruction selection and code compaction, which is explained in the following section.

3 Tree covering with complex instructions

Our approach proceeds in two sequential phases: In phase 1, we allow atomic (possibly complex) instructions to be temporarily split into their single RTs, so that tree covering can be efficiently performed with dynamic programming. In phase 2, the selected RTs are recombined in a way that they only form valid instructions again. We therefore characterize a processor by two different items: Firstly, there is a set of *register transfers* which can be executed on the processor datapath:

$$RTS = \{RT_1, \ldots, RT_n\}$$

A register transfer is a single-cycle tree-pattern operation which computes a value and stores the result into a destination memory or register module. Secondly, the actual *instruction set* is defined as

$$IS = \{I_1, \ldots, I_m\}$$

where each instruction I_k is a set of one or more parallel RTs:

$$I_k = \{RT_{k1}, \ldots, RT_{km_k}\}, \qquad RT_{ki} \in RTS$$

Thereby, available ILP is encoded in the instructions. An instruction I_k is called *complex*, if $|I_k| > 1$. Furthermore, we define a *membership function*

$$M : RTS \rightarrow \mathcal{P}(IS) : RT_i \mapsto \{I_k \quad | \quad RT_i \in I_k\}$$

so that $M(RT_i)$ contains all instructions that comprise RT_i. Using these formulations, tree covers including complex instructions can be obtained as explained in the following. Note that we are only dealing with *restricted* ILP due to encoded instruction formats, while for horizontal formats the size of IS would explode.

3.1 Phase 1: Tree covering by RTs

In phase 1, we temporarily consider the RT set RTS as the instruction set. Tree covering then proceeds by dynamic programming. The instructions are modelled as tree patterns, and an optimal cover (including pure data transport operations) is computed, under the assumption that no ILP is available. In our case we use the `iburg` pattern matcher generator [8]. `iburg` takes as input a *tree grammar* describing the available RT patterns on a processor and (similar to `yacc`) produces C source code for a fast, processor-specific tree pattern matcher. As proposed in [4], we include *register-specific patterns* into the tree grammar, in order to integrate register allocation and instruction selection. In contrast to other approaches, the `iburg` tree grammar itself is not constructed manually, but is derived from a processor model in the MIMOLA hardware description language. In this way, a closer link to ECAD environments is provided. Our retargetable compiler system RECORD uses a BDD-based technique [9] for *extracting* the instruction set from this processor model. Tree covering also includes application of *transformations* (e.g. algebraic rules), which are however beyond the scope of this paper.

In order to enable selection of complex instructions in phase 2, each subtree covered by one RT_i is collapsed to a supernode labelled with $M(RT_i)$, of which one $I \in M(RT_i)$ can be selected later. For instance, suppose a processor with an RT set

$$RTS = \{RT_1, RT_2, RT_3, RT_4\}$$

and an instruction set

$$IS = \{I_1, I_2, I_3, I_4, I_5, I_6\}$$

where

$$\begin{array}{lll} I_1 = \{RT_1\}, & I_2 = \{RT_2\}, & I_3 = \{RT_3\} \\ I_4 = \{RT_4\}, & I_5 = \{RT_2, RT_3\}, & I_6 = \{RT_2, RT_4\} \end{array}$$

Fig. 4 shows a covered expression tree (a) and the resulting tree with labelled supernodes (b). While in phase 1 neighboring subtrees are only considered apart from each other, phase 2 uses a global view of the tree in order to assign supernodes to real (possibly complex) instructions.

3.2 Phase 2: Composition of RTs

According to [4] there exists a criterion (*RTG criterion*) under which an optimal schedule can be derived from a previously covered tree in linear runtime by selecting an appropriate tree evaluation order. For processors fulfilling that criterion, superfluous spill instructions can always be avoided by adding additional resource dependency edges to the tree. After that, scheduling the tree basically requires topological sorting. Optimal schedules are obtained only for expression trees which do not contain potentially parallel operations, i.e. those trees, for which complex instructions are useless.

In fact, the problem of optimal scheduling becomes much more difficult for arbitrary trees, where the

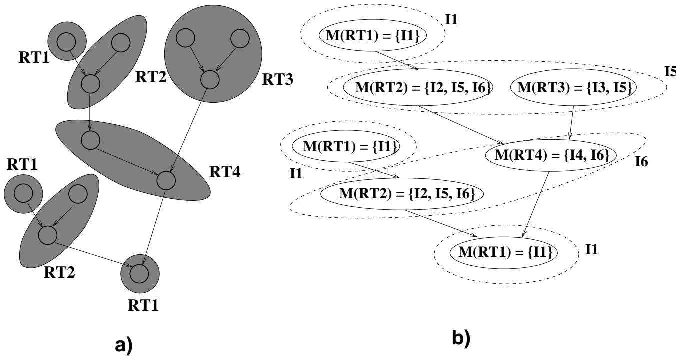

Figure 4: *a) Expression tree covered with RTs, b) corresponding tree with labelled supernodes and its covering by (complex) instructions*

scheduling problem turns into a compaction problem: The dependency relations induced by the tree edges have to be obeyed, and two nodes with labels

$$M(RT_j) = \{I_{j1}, \ldots, I_{jn}\}, \quad M(RT_k) = \{I_{k1}, \ldots, I_{km}\}$$

may only be parallelized if there exist a corresponding complex instruction, i.e.:

$$M(RT_j) \cap M(RT_k) \neq \emptyset$$

Under the assumption that a spill-free schedule exists, and that appropriate resource dependency edges have been added to the tree, this is obviously equivalent to *resource-constrained scheduling*, known to be NP-hard. However, good exploitation of ILP demands for an exact solution, i.e. an optimal selection of instructions has to be made, which minimizes the total code length. Exact scheduling yields the following optimal schedule for the above example, which makes use of two complex instructions: (dashed ellipses in fig. 4 b)

(1) $I_1 = \{RT_1\}$
(2) $I_5 = \{RT_2, RT_3\}$
(3) $I_1 = \{RT_1\}$
(4) $I_6 = \{RT_2, RT_4\}$
(5) $I_1 = \{RT_1\}$

While most previous compaction algorithms rely on heuristics, Timmer [10] recently presented a technique that performs exact code compaction very efficiently using a branch-and-bound search in combination with an a-priori reduction of the search space. This technique can be used, whenever each single RT is also a valid instruction, i.e. in our formulation

$$\forall RT_i \in RTS \quad \exists I_k \in IS : I_k = \{RT_i\}$$

which is often the case. Furthermore, conflicts between RTs must be modelled *statically before scheduling*, which requires for arbitrary RTs $RT_i, RT_j, RT_k \in RTS$:

$$\forall I, I' \in IS :$$

$$I \in M(RT_i) \cap M(RT_j) \quad \wedge \quad I' \in M(RT_i) \cap M(RT_k)$$

$$\Rightarrow \quad \exists I'' \in IS : \{RT_i, RT_j, RT_k\} \subseteq I''$$

i.e. whether or not two RTs can be parallelized must not depend on selection of a certain complex instruction. Unfortunately, this condition does not apply to arbitrary cases, restricting Timmer's technique to code compaction for purely horizontal instruction formats. In case of encoded instruction formats, more versatile methods are required. As an example, consider three RTs on the TMS320C25:

(1)	T register := memory value	(load TR)
(2)	ACCU := ACCU + PR	(accumulate)
(3)	PR := TR * memory value	(multiply)

With respect to these RTs, the following membership relations hold for the complex TMS320C25 instructions **LTA** and **MPYA**:

$$\mathbf{LTA} \in M(\text{accumulate}) \cap M(\text{load TR})$$

$$\mathbf{MPYA} \in M(\text{accumulate}) \cap M(\text{multiply})$$

However, there exists no instruction $I \in IS$, for which

$$\{\text{accumulate}, \text{load TR}, \text{multiply}\} \subseteq I$$

for two different memory values, so that static conflict modelling in advance is impossible for this processor.

Therefore, our approach is based on more general definition of the compaction problem by means of Integer Programming (IP), which requires exponential runtime in the worst case, but exactly solves the scheduling problem for arbitrary cases: It neither requires that each RT is a valid instruction on its own, nor that all RT conflicts can be modelled statically before scheduling. The details of the IP model itself are subject of another paper [11]. Given a tree with labelled supernodes as in fig. 4 b), it yields the optimal assignment of RTs to instructions, i.e. for each supernode labelled with possible instructions

$$M(RT_j) = \{I_{j1}, \ldots, I_{jn_j}\}$$

exactly one $I_{jk} \in M(RT_j)$ is selected, such that the total schedule length is minimized. Code compaction including complex instructions can be regarded as a generalization of the classical compaction problem, due to the following reasons: Firstly, compaction in presence of complex instructions necessarily must employ a mechanism for *version shuffling*, i.e. for each RT a set of alternative binary encodings must be maintained, from which the most appropriate one is selected for each control step. Some classical compaction algorithms do include version shuffling [5], but do not take into account possible side effects: On the TMS320C25, for instance, both the **MPY** (multiply) and **MPYA** (multiply-accumulate) instructions represent are valid encodings versions for a multiplication. However, in case that code for a multiplication is to be generated,

but no accumulate operation is ready to be scheduled in parallel, the compaction mechanism must ensure, that **MPY** is selected rather than **MPYA**. Otherwise, the accumulator register, possibly containing a live value, would be overwritten as a side effect. Nowak's MSSQ code generator [12] uses the concept of "no-operations" (NOPs) in order to avoid undesired side effects during compaction. While other approaches assume that appropriate NOPs are implicit in the binary encoding of each instruction, MSSQ explicitly packs NOPs into each control step. NOPs ensure that unused sequential components (registers and memories) retain their state during one control step. MSSQ however performs heuristic packing of NOPs as a postpass phase, so that decisions concerning version selection are never revised. As a consequence, code compaction in MSSQ might fail, because an undesired side effect cannot be avoided, although a solution exists. In contrast, our IP-based technique permits to revise compaction decisions at any point in time, so that optimal solutions are guaranteed to be found.

It should be noted, that although our approach provides a close coupling of instruction selection and compaction, and each of both phases is solved optimally, the overall result is not necessarily *globally* optimal for an expression tree. This is due to the fact, that instruction selection by tree covering cannot take into account whether or not two RTs can be parallized by means of a complex instruction. One can construct examples, where optimal tree covering in terms of the smallest number of RTs does not lead to the most compact code.

The next section discusses the practical application of the two-phase instruction selection technique for the TMS320C25 processor.

4 Experimental results

As a first example, consider the expression tree in fig. 5. Six TMS320C25 instruction patterns are necessary for covering the tree:

LT:	(T register := memory value)
MPY:	(P register := T register * memory value)
PAC:	(ACCU := P register)
APAC:	(ACCU := ACCU + P register)
SPAC:	(ACCU := ACCU - P register)
SACL:	(memory cell := ACCU)

Code generation according to [4] leads to the schedule of length 16 in fig. 6 a). Exploiting the complex TMS320C25 instructions

LTP = {**LT**, **PAC**}
LTA = {**LT**, **APAC**}
MPYA = {**MPY**, **APAC**}

however leads to the code of length 12 shown in fig. 6 b). Four pairs of instructions have been replaced by one complex instruction each. In case of the **MPYA** instruction, the execution order needed to be changed: In order to enable exploitation of **MPYA**, operation **LT m3** is scheduled before the originally preceeding **APAC** operation.

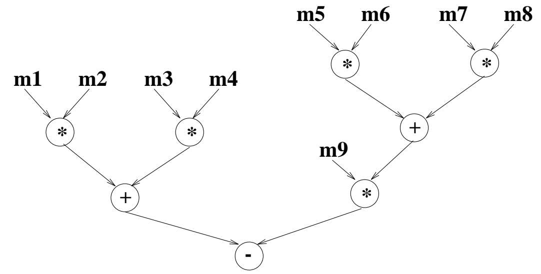

Figure 5: *Expression tree with potential parallelism. m1 to m9 refer to values in memory.*

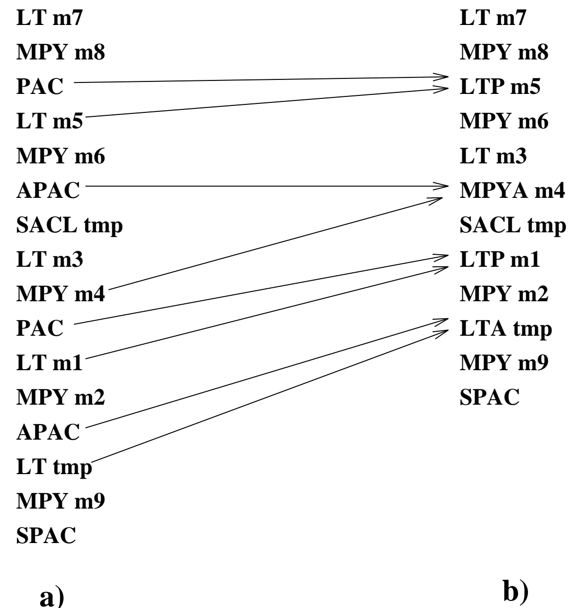

Figure 6: *Generated code sequences for the expression tree in fig. 5: a) without complex instructions, b) compact code with complex instructions LTP, LTA, and MPYA (*`tmp` *refers to a temporary cell in memory).*

Further experiments also indicate that complex instructions should be exploited whenever very high code quality must be ensured. Table 1 lists results for expression trees which have been extracted from DSP algorithms. Columns 2 and 3 show the number of instructions generated by tree covering without consideration of instruction-level parallelism (ILP), and the number achieved by our approach, respectively. The CPU seconds (measured on a SUN SparcStation 20, Integer Programs solved with Optimization Subroutine Library (OSL) V 1.2 from IBM) are given in column 4, while column 5 shows the relative improvement. The experiments demonstrate that up to one third of the instructions can be saved by optimal exploitation of avail-

exp. tree	no ILP	ILP	CPU	% gain
fig. 5	16	12	2	25
adaptive1	20	17	8	15
adaptive2	19	14	6	26
pidctrl	13	9	30	30
lattice2	18	17	35	5
fir12	33	24	4	27
equalize	39	27	3	31
dct	23	20	6	15
bassboost	31	23	18	25

Table 1: *Code quality improvements by exploitation of complex instructions*

able parallelism after tree covering. Time for dynamic programming was always less than 1 second and can be neglected. Due to the usage of an IP solver for scheduling, the CPU times are relatively high in some cases, and are not proportional to the problem size. As mentioned in section 1, however, there may be good reasons to spend more time for code optimization, whenever significant improvements are the result. This is obviously the case here. Beyond tree sizes of 40 however, the required runtime often exceeds the acceptable amount. We therefore consider our technique to be complementary to heuristic compaction methods. Its main application area can be identified as code optimization for very time-critical pieces of DSP algorithms, such as loop bodies.

5 Conclusions and future work

It was shown that instruction selection in code generation for embedded processors is strongly affected by the presence of instruction-level parallelism in form of complex instructions. Complex instructions prohibit one-phase optimal tree covering by dynamic programming because they result in non-tree patterns. By partially postponing the instruction selection phase to the compaction phase, tree covers can be generated which also exploit complex instructions. This is achieved by a combination of dynamic programming and exact code compaction. Examples for a popular DSP, the TMS320C25, showed that this technique significantly improves earlier work in terms of code quality. To our knowledge, this is the first time, that systematic exploitation of complex instructions is reported for that processor.

Similar to the optimality criterion provided in [4], deeper theoretical investigations are necessary to identify those processors, for which tree covering followed by exact compaction actually leads to optimal code in presence of complex instructions. Future work is also necessary to combine the presented technique with other compilation phases. Furthermore, as already indicated by Timmer's results [10], we expect that much of the time needed for compaction could be saved at the expense of lower retargetability, when only certain instruction formats need to be considered.

Acknowledgements
The authors would like to thank Steven Bashford for helpful discussions on tree pattern matching. This work has been partially supported by the European Union under ESPRIT contract 9138 (CHIPS).

References

[1] P. Marwedel: *Code Generation for Embedded Processors: An Introduction*, chapter 1 in: P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[2] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang, A. Wang: *Challenges in Code Generation for Embedded Processors*, chapter 3 in : P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[3] A. V. Aho, M. Ganapathi, S. W. K. Tjiang: *Code Generation using Tree Pattern Matching and Dynamic Programming*, ACM Trans. on Programming Languages and Systems, Vol. 11, No. 4, Oct. 1989, pp. 491-516

[4] 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

[5] 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

[6] B. Wess: *Code Generation based on Trellis Diagrams*, IEEE Int. Symp. on Circuits and Systems, 1992, pp. 645-648

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

[8] C. W. Fraser, D. R. Hanson, T. A. Proebsting: *Engineering a Simple, Efficient Code Generator Generator*, Journal of the ACM, 22(12), 1993, pp. 248-262

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

[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 (DAC), 1995, pp. 593-598

[11] R. Leupers, P. Marwedel: *Time-constrained Code Compaction for DSPs*, 8th Int. Symp. on System Synthesis (ISSS), 1995, pp. 54-59

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