Graph based Code Selection Techniques for Embedded Processors

Rainer Leupers and Steven Bashford
University of Dortmund, Dept. of Computer Science 12, Dortmund, Germany
email: leupers|bashford@LS12.cs.uni-dortmund.de

Code selection is an important task in code generation for programmable processors, where the goal is to find an efficient mapping of machine-independent intermediate code to processor-specific machine instructions. Traditional approaches to code selection are based on tree parsing, which enables fast and optimal code selection for intermediate code given as a set of data-flow trees. While this approach is generally useful in compilers for general-purpose processors, it may lead to poor code quality in the case of embedded processors. The reason is that the special architectural features of embedded processors require to perform code selection on data-flow graphs, which are a more general representation of intermediate code. In this paper, we present data-flow graph based code selection techniques for two architectural families of embedded processors: media processors with support for SIMD instructions and fixed-point DSPs with irregular data paths. Both techniques exploit the fact that, in the area of embedded systems, high code quality is a much more important goal than high compilation speed. We demonstrate that certain architectural features can only be utilized by graph based code selection, while in other cases this approach leads to a significant increase in code quality as compared to tree based code selection.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: code selection, data-flow graphs, SIMD instructions, irregular data paths, embedded processors

1. INTRODUCTION

Code generation is the problem of mapping an intermediate representation (IR) of a given source program to an equivalent machine program for a given target processor. This involves the subtasks of *code selection*, *register allocation*, and *instruction scheduling*. In this paper, we primarily focus on code selection, where the goal is to find an efficient mapping of an IR to machine instructions without considering detailed register allocation and instruction scheduling.

Code selection can be visualized as a problem of *pattern matching* between a data-flow based IR and available instruction patterns. An example is shown in fig. 1, where part a) shows the data-flow tree (DFT) representation of a computation, part

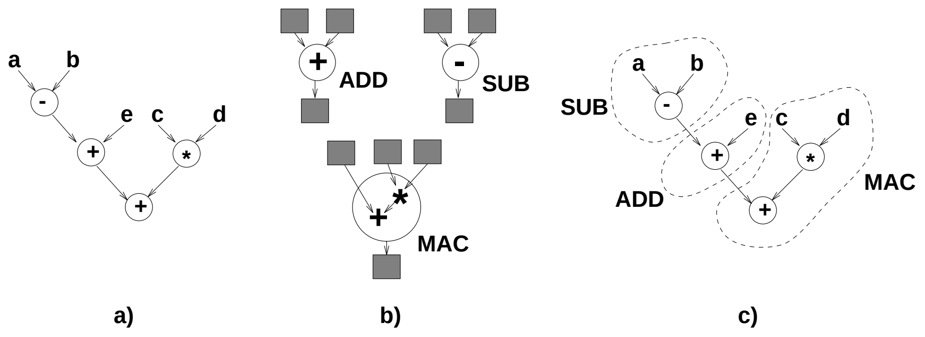

Fig. 1. *Visualization of code selection*

b) shows three sample instruction patterns, and part c) shows a possible covering of the DFT by instruction pattern instances, where each instance matches a certain subtree of the DFT.

There is an efficient and optimal algorithm for code selection [1] which is based on *tree parsing*. Tree parsing can be implemented by tree pattern matching and dynamic programming (see [2] for an overview). Given a certain cost metric for instruction patterns (such as size or execution cycles), tree parsing computes optimal DFT covers by instructions patterns, such as the one shown in fig. 1 c). More formally, tree parsing computes an optimal derivation of a DFT w.r.t. a given tree grammar specification. This algorithm, a variant which is used in many code generators for general-purpose processors, requires only linear time in the DFT size, and its has largely replaced earlier tree-oriented code generation techniques, such as [3; 4].

However, tree parsing also requires that the IR is actually given in the form of DFTs. Furthermore, optimality is only guaranteed, if potential *parallelism at the instruction level* is neglected.

These restrictions lead to two problems when applying standard tree parsing to certain classes of embedded processors.

(1) If the target processor instruction set allows for instruction level parallelism, which is commonly the case for *digital signal processors* (DSPs), then the code selected by tree parsing may strongly deviate from the optimum. An example is given in fig. 2, where the DFT represents a sum-of-products computation. If the target processor offers ADD, MULT, and MAC (multiply-accumulate) instructions, where the latter execute an addition and a multiplication in parallel, then the optimum DFT cover is that one shown in fig. 2 a). However, as tree parsing is not capable of detecting potential parallelism, it generates the cover shown in fig. 2 b), which requires 7 instead of 5 instructions.

(2) If the IR code to be compiled contains *common subexpressions* (CSEs), then its data-flow based representation has the form of a general *data-flow graph* (DFG), whose nodes, in contrast to DFTs, may have a fanout larger than one. Let us assume that re-computation of CSEs on demand is always more expensive that computing the CSE once and keeping it in a register during its lifetime. Then, the common approach to make tree parsing applicable is to break the DFG at

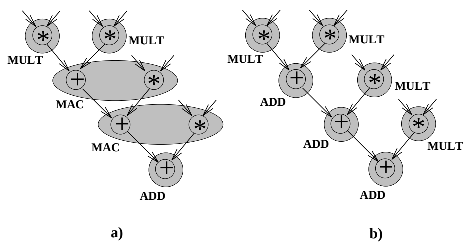

Fig. 2. *Code selection with parallel instructions*

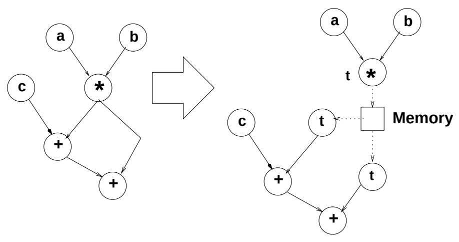

Fig. 3. *Decomposition of a DFG into DFTs*

the CSEs, such that the DFG is transformed into a set of DFTs (fig. 3), and the CSE is communicated between DFTs via register write/read operations. For RISC-like processors with homogeneous register files, this is usually a reasonable approach. However, this is not true for many DSPs which frequently show an irregular data path with *special-purpose registers* (or register files) of very limited storage capacity. For such architectures, the *routing of values* between registers via "move" instructions may significantly contribute to the amount of code generated. The common approach in DSP compilers is to route a CSE through the memory, since in DSPs the memory is the only resource capable of storing values for a relatively long time. However, in certain situations, it might be much more efficient to hold a CSE in a certain special-purpose register. These opportunities cannot be directly detected with tree parsing, since the interface for transporting CSEs between DFTs must be fixed in advance. In addition, the opportunity of using complex instructions for CSEs (such as a MAC for the CSE in fig. 3) is excluded in advance when splitting a DFG into DFTs.

Since high code quality is of great importance for embedded processors, special code selection techniques, tuned for the specific architectures of embedded processors, are obviously required. This paper presents two such code selection techniques. These techniques are complementary in the sense that they have been designed for different families of embedded processors. However, the common approach is to perform code selection on *complete DFGs* instead of (decomposed) DFTs. In this way, the problems mentioned above can be circumvented, and better code can be generated. A further common characteristic of the two presented techniques is that, in a first phase, *alternative solutions* are generated, while the detailed code selection is performed only later when sufficient global information about an entire DFG is available.

The structure of the paper is as follows. In the next section, we discuss related work in the area of code selection for embedded processors. In section 3, we describe a graph based code selection technique for media processors showing a special kind of instruction level parallelism in the form of so-called SIMD (*single instruction, multiple data*) instructions. We demonstrate that graph based code selection is necessary for exploiting SIMD instructions, and we present results for two existing media processors. Section 4 deals with code selection for DSPs with irregular data paths. We present a code selection technique, based on the paradigm of *constraint logic programming*, which is capable of *optimal* code selection for DFGs. Due to the high runtime requirements, its applicability is restricted to small to medium size DFGs. Therefore, we also present a heuristic variant that produces near-optimal results in comparatively short time, and we quantify the increase in code quality as compared to the tree parsing technique. Finally, conclusions are given.

2. RELATED WORK

As already mentioned, DFT based code selection by the tree parsing algorithm [1] is the technique of choice in many compilers for general-purpose processors, where high compilation speed plays a major role.

Since throughout this paper we will refer to tree parsing several times, we will give a simplified summary of the technique here. The instruction set of the target processor needs to be described as a *tree grammar*

$$G = (\Sigma_T, \Sigma_N, S, R, c)$$

where Σ_T is a set of *terminals*, Σ_N is a set of *nonterminals*, $S \in \Sigma_N$ is the *start symbol*, R is a set of *rules*, and $c : R \rightarrow \mathbb{N}_0$ is a function that assigns a *cost value* to each rule in R. Σ_T is a representation of the nodes occurring in a DFT (variables, constants, and operators), while Σ_N is primarily used to model hardware components that can store data (registers, memories). Additionally, nonterminals are used for factoring common parts of instruction patterns. The instruction patterns themselves are modeled as rules in R. Each such rule describes the behavior of an instruction. For instance, in order to model a MPY instruction which multiplies two register contents and writes the result to a register, the following rule could be used:

```
reg: MULT(reg,reg)
```

Here, `MULT` is a terminal representing the multiply operator, and `reg` is a nonterminal representing a (general-purpose) register. Function c would be used to assign a cost value to that rule. Another example is the rule

```
dft: STORE(reg,PLUS(reg,offs))
```

which describes a store from a register to a memory location addressed by another (pointer) register plus an offset. Here, `offs` is another nonterminal that factors different offset modes (such as long/short constant or register), while in this case `dft` is the grammar start symbol. Using the start symbol at the left hand side of a rule denotes that this rule is expected to match the root node of a DFT, as well as possibly some of its children nodes.

The tree parsing algorithm computes an optimal *derivation* of a given DFT from the start symbol of the tree grammar, while using function c as the cost metric. The main idea is, that an optimal derivation of any subtree ST of a given DFT T can be computed by considering combinations of three (usually quite small) sets: the rules matching the root of ST, as well as the optimal derivations for the left and right subtrees of ST w.r.t. each grammar nonterminal. Therefore, the optimization paradigm of *dynamic programming* is applicable, and this is what makes tree parsing efficient. Another important advantage of tree parsing is its easy *retargetability*: Modifications of the instruction set directly translate into modifications of the tree grammar, and retargeting is supported by tools (e.g. twig [1] and iburg [5]) which automatically generate code selector C source code for a given tree grammar.

Due to the limitations of tree parsing mentioned above, several recent contributions have dealt with the generalization of DFT based code selection towards DFGs. In [6], the tree parsing technique has been generalized for DFGs, but the approach is restricted to regular data path architectures and does not handle instruction-level parallelism. Other DFG based techniques [7; 8] have been specifically designed for DSP processors with irregular architectures. However, they do not adequately solve the problem of efficiently routing CSEs, but assume that CSEs are strictly stored into memory. An approach to code selection with complex instructions (such as the MAC from fig. 2 a) has been described in [9] which, however, is still restricted to DFTs.

The main contributions of this paper, as compared to previous work, are twofold: First, in section 3, we describe a code selection technique, capable of exploiting SIMD instructions. To our knowledge, code selection with SIMD instructions, which is needed for recent families of media processors, has so far not been addressed in previous work. As a consequence, current compilers require the use of assembly libraries and/or compiler intrinsics to exploit SIMD instructions. However, this method results in comparatively high programming effort and low portability of source code. In contrast, the technique proposed in this paper works for plain ANSI C source code without machine-specific language extensions or assembly libraries. Second, in section 4, we provide a technique for exact code selection for DFGs, capable of optimally exploiting special-purpose registers and complex (chained) instructions. This is an extension of work presented in [10], where primarily the integration of code selection with register allocation and instruction scheduling has been described.

3. CODE SELECTION FOR MEDIA PROCESSORS

In order to support the fast execution of computation-intensive multimedia application programs, dedicated *media processors* are available on the semiconductor market. These machines provide architectural support for efficiently processing different data types on the same data path. Examples are the Texas Instruments C6201 [11], the Philips Trimedia [12], and – to a certain extent – Intel's Pentium MMX architecture [13].

Many media processors show a 32-bit data word length. However, applications in the audio or video domain normally require only a precision of 16 or 8 bits, respectively, resulting in a potential waste of computational resources. Therefore, media processors show a special kind of machine instructions, that permit to virtually split each full data register into multiple *sub-registers* and to perform identical computations on the sub-registers in parallel. These instructions are now commonly called *SIMD (single instruction, multiple data) instructions*[1].

A major problem with SIMD instructions is the missing capability of C compilers to exploit such instructions due to the lack of dedicated code selection techniques. As already mentioned, this problem can be circumvented by using compiler intrin-

[1]In the literature, there are sometimes other terms for this feature, such as *split-ALU instructions*, *short vector instructions*, or *sub-word parallelism*.

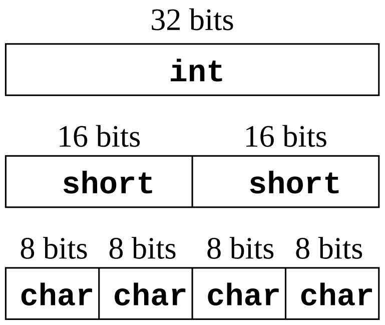

Fig. 4. *Splitting 32-bit registers into sub-registers*

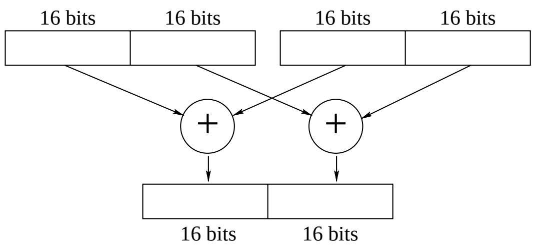

Fig. 5. *TI C6201 instruction "ADD2"*

sics or hand-optimized assembly libraries. Also special language extensions have been proposed [14]. The expense, however, is a high programming effort and low portability of the source code. Our goal, therefore, is to exploit SIMD instructions in code selection for *plain* ANSI C source code.

3.1 SIMD instructions

We call an instruction a SIMD instruction, if it performs a manipulation (arithmetic or logic operation, load or store) of data stored sub-registers instead of full registers. For the use of SIMD instructions, the 32-bit data registers are considered to be composed of either two 16-bit sub-registers or four 8-bit sub-registers (fig. 4). Thus, in terms of the C programming language, any full register may store either four "char" data, two "short" data, or a single "int" at a time. In the following, for sake of simplicity, we will emphasize SIMD instructions on 16-bit data, although the proposed technique can be easily scaled to 8-bit data as well.

Fig. 5 gives an example of the SIMD instruction "ADD2" of the TI C6201. It performs two 16-bit additions in parallel and writes two results into the two sub-registers of the destination register. While arithmetic SIMD instructions require special hardware support, such as the suppression of carry propagation, there are also "trivial" SIMD instructions like those performing logic operations (AND, OR, XOR, NOT).

In order to take full advantage of SIMD instructions, it is necessary, that the 16-bit or 8-bit data to be manipulated are efficiently loaded from and stored into memory. Under certain conditions, one can use 32-bit instructions to load operands and store results of SIMD instructions. As an example, consider the piece of C code in fig. 6, which describes a vector addition on short data. In this example, the loop body has been unrolled once, so as to reveal the potential parallelism.

Using the above "ADD2" instruction, the two additions in the loop body could be executed in parallel. However, this requires that the operand pairs B[i], C[i] and B[i+1], C[i+1] are loaded into the lower and upper halves of the argument registers, respectively[2]. Therefore, B[i] and B[i+1] must be loaded by a single 32-bit load instruction instead of two separate 16-bit loads, and the same applies to C[i] and

[2]On some processors this requires a memory alignment to word boundaries. We assume that an appropriate alignment can be ensured by compiler or assembler directives.

```
void f(short* A,short* B,short* C)
{ int i;
  for (i = 0; i < N; i += 2)
  { A[i]   = B[i] + C[i];
    A[i+1] = B[i+1] + C[i+1];
  }
}
```

Fig. 6. *Source code for vector addition*

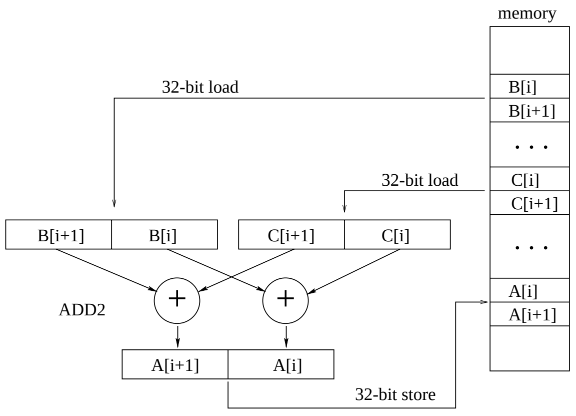

Fig. 7. *Parallelization of vector addition with SIMD instructions*

C[i+1]. Since adjacent array elements are stored in adjacent memory locations, this can be accomplished with two 32-bit load instructions. After execution of "ADD2", the results A[i] and A[i+1] are located in the lower and upper half of the destination register, and a single 32-bit store operation suffices to write the two results back to memory. This is illustrated in fig. 7. In total, the number of instructions required to execute the vector addition can be reduced by 50 % when using SIMD instructions.

There are two major difficulties in exploiting SIMD instructions. Firstly, parallel loading or storing of values located in sub-registers from/to memory requires to establish that the memory address difference is correct. In our C compiler, we apply a standard global data flow analysis technique to pointers in order to determine those sets of operands that qualify for parallel loading and storing with 32-bit instructions.

A more difficult task is to correctly pack potentially parallel instructions together during code generation, so as to form SIMD instructions. Generating SIMD instructions on-the-fly only during the instruction scheduling and register allocation phases, although possible, would be very difficult, because a large number of constraints need to be obeyed. If, for instance, multiple values share a single 32-bit register, then their live ranges are tightly coupled. As a consequence, standard register allocation techniques, such as graph coloring [15], cannot be applied.

Instead, we propose to generate SIMD instructions already early in the code generation process during the code selection phase. The generated code afterwards only operates on (symbolic) full 32-bit registers, so that existing instruction scheduling and register allocation techniques can still be used.

3.2 Generation of alternative DFG covers

Our code selection technique operates on a DFG representation of basic blocks. Using full DFGs (instead of separate DFTs) is necessary, since exploitation of SIMD instructions frequently requires to pack together operations located in different DFTs. The DFGs are generated by means of the LANCE ANSI C frontend [16].

A given DFG is partitioned into multiple DFTs by cutting the DFG at the CSE edges and computing optimal covers for each single DFT. Since media processors tend to show a regular data path architecture, this does not incur significant losses in code quality. However, this standard approach is not directly capable of generating SIMD instructions, since this in general requires the consideration of multiple DFTs at a time.

We solve this problem by permitting the generation of *alternative solutions* during tree pattern matching. Instead of annotating only a single optimal solution (i.e., a grammar rule matching at minimum cost) to each DFG node, we annotate all optimal rules, including those for SIMD instructions, and only later determine the best rules globally for the whole DFG. In order to achieve this, we introduce dedicated nonterminal symbols in the tree grammar, which denote the different possibilities of using a register: either as a full 32-bit register or as two separate 16-bit registers. As an example, consider instructions for addition on a C6201 processor. Instruction "ADD" adds two 32-bit registers and also writes the result to a 32-bit register. This can be expressed by the following tree grammar rule, where nonterminal *reg* denotes a full register.

```
reg: PLUS(reg,reg)
```

The SIMD instruction "ADD2" (fig. 5) simultaneously performs two 16-bit additions. We use two separate rules for modeling the behavior of "ADD2":

```
reg_lo: PLUS(reg_lo,reg_lo)
reg_hi: PLUS(reg_hi,reg_hi)
```

The nonterminals "reg_lo" and "reg_hi" denote the lower and upper 16-bit sub-registers of a full register. Both rules are assigned the same cost value as the 32-bit version. As a consequence, there exist three alternative optimal covers for all DFG nodes representing a PLUS operation. All other instructions that qualify for a SIMD execution mode (arithmetic, logic, load, store) are modeled similarly.

Note that the rule costs for SIMD instructions are not counted twice. Rule costs are only considered during the DFG covering phase in order to obtain alternative optimal DFT covers. During the subsequent instruction packing phase described below, which aims at maximizing the use of SIMD instructions, the rule costs are no longer required. Additionally, rules representing operations on sub-registers may not be considered as "stand-alone" instructions, since this would result in invalid code. The constraint system presented in the following subsection ensures, that such rules can only be used for covering *pairs* of DFG nodes, in which case only a single SIMD assembly instruction is emitted.

The overall DFG covering process works as follows. The DFG is partitioned into DFTs by assigning each CSE to a symbolic register and replacing all uses of CSEs by read operations on that register. Then, all DFTs are separately covered by means of an extended tree parsing technique (cf. section 2): We use a modification of the tool olive (a variant of twig [1]) to generate the required code selector from an instruction set tree grammar. Since olive in the original version only computes a single optimal solution (with ties broken arbitrarily) for each DFT and thus only annotates a single rule at each DFT node, we have modified the tool in such a way, that alternative optimal covers are retained during DFT covering. Our modified olive version annotates *all minimum cost derivations for each nonterminal* at the DFT nodes. Whether or not SIMD instructions are selected is decided only later globally for the entire DFG.

3.3 Packing of SIMD instructions

After DFG covering we determine the detailed DFG node covers to be selected from the available alternatives. In this phase, the goal is to maximize the use of SIMD instructions across the entire DFG. We solve this problem by transforming the code selection problem into a (quite compact) Integer Linear Program (ILP) formulation.

For each DFG node n_i, the DFG covering phase returns a set of alternative rules $R(n_i)$, which match n_i at minimum costs. We use Boolean variables x_{ij} to express that node n_i is (or is not) covered by rule $r_j \in R(n_i)$. A valid code selection requires that each node is covered by exactly one rule. Therefore, for each node n_i, we impose the constraint

$$\sum_{r_j \in R(n_i)} x_{ij} = 1$$

3.4 Constraints due to DFT edges

Selecting a certain rule r_j for some node n_i has implications on the covering of its children nodes in the DFT. If, for instance, node n_i is covered by rule

```
  reg_lo: PLUS(reg_lo,reg_lo)
```

then it must be ensured that the first and second child of n_i are derived to nonterminal "reg_lo", i.e., the arguments of the PLUS operation reside in lower 16-bit sub-registers. More generally, let $r_j \in R(n_i)$ be the rule selected for node n_i, let n_k be the m-th child (counting left-to-right) of n_i in a DFT, and let $r_l \in R(n_k)$ be the rule selected for n_k. Since n_k is the m-th child of n_i, the nonterminal on the left hand side (LHS) of r_l must be equal to the m-th nonterminal, say nt_m, on the right hand side of r_j. Let $R_m(n_k) \subset R(n_k)$ denote the set of rules r_l for n_k, such that LHS(r_l) $= nt_m$. Then, the following constraint expresses the dependence between n_i and n_k:

$$x_{ij} \leq \sum_{r_l \in R_m(n_k)} x_{kl}$$

3.5 Constraints for common subexpressions

The next class of constraints concerns code selection for common subexpressions (CSEs) in the DFG. As already mentioned, each CSE is strictly assigned to a register, and we insert register read/write nodes (using dedicated grammar terminals) into the DFG so as to replace the CSE edges.

There may exist alternative covers for storing 16-bit "short" CSEs, since these may reside in either full registers or sub-registers. This should not be neglected, since SIMD instructions can sometimes also be exploited for parallel computation of CSEs. A correct code selection requires that the locations (i.e., either upper, lower, or full register) for a CSE definition and its uses are identical across the entire DFG. Thus, for any "short" CSE definition/use pair, the ILP model contains constraints that enforce the assignment of the definition and the use to the same (sub-)register. This is achieved by simply unifying the corresponding x_{ij} variables of the definition and the use.

3.6 Constraints for selecting SIMD instructions

Another class of constraints ensures a valid packing of instructions to SIMD instructions. For this purpose, we introduce the notion of *SIMD pairs.* A pair (n_i, n_j) of DFG nodes is called a SIMD pair, if the following conditions are satisfied:

—There is no scheduling precedence between n_i and n_j

—n_i and n_j have the same operator

—According to the tree grammar rules, n_i may be located in an upper sub-register and n_j may be located in a lower sub-register.

—If n_i and n_j represent load or store operations of 16-bit values, where a_i and a_j are the corresponding memory addresses, then the difference $a_j - a_i$ equals the number of memory words occupied by a 16-bit value (e.g. 2 for a byte-addressable memory).

The latter condition ensures, that parallel loads and stores of sub-registers implemented by SIMD instructions actually refer to adjacent data in memory.

The set P of all SIMD pairs can be computed from the information generated by DFG covering. The required runtime is quadratic in the number of DFG nodes. Any DFG node contained in a SIMD pair can potentially be mapped to a SIMD instruction. However, it must be guaranteed that any selected SIMD instruction actually covers a *pair* of DFG nodes and that any DFG node is covered by *at most* one SIMD instruction.

In order to express these conditions in terms of ILP constraints, we introduce one auxiliary Boolean variable y_{ij} for each SIMD pair (n_i, n_j). The setting of $y_{ij} = 1$ denotes that n_i and n_j are packed into a single SIMD instruction, i.e., n_i operates on the upper sub-register and n_j operates on the lower sub-register of the same full register.

For any n_i let $R_{hi}(n_i) \subset R(n_i)$ and $R_{lo}(n_i) \subset R(n_i)$ denote the sets of rules for n_i operating on an upper or a lower sub-register, respectively. If n_i is covered by some rule in $R_{hi}(n_i)$, then there must be a node n_j, such that $(n_i, n_j) \in P$, and n_j is covered by a rule in $R_{lo}(n_j)$. Conversely, if n_i is covered by some rule in $R_{lo}(n_i)$, then there must be a node n_j, such that $(n_j, n_i) \in P$, and n_j is covered by a rule in $R_{hi}(n_j)$. For any n_i contained in a SIMD pair, this is modeled by two constraints:

$$\sum_{j:(n_i,n_j)\in P} y_{ij} \quad = \sum_{r_k \in R_{hi}(n_i)} x_{ik}$$

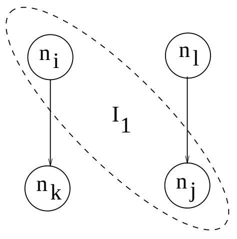

Fig. 8. *Potential scheduling deadlock situation*

$$\sum_{j:(n_j,n_i)\in P} y_{ji} = \sum_{r_k \in R_{lo}(n_i)} x_{ik}$$

Since the right hand sides of the equations are always less or equal to 1, it is also guaranteed, that any node n_i is packed into at most one SIMD instruction.

3.7 Constraints for preserving schedulability

The last class of constraints is required for avoiding code selection decisions leading to scheduling deadlocks. For any DFG node n_i, let $pred(n_i)$ denote the set of nodes that must be scheduled before n_i (e.g. due to data or output dependence), and let $succ(n_i)$ be the set of nodes to be scheduled after n_i. Whenever a SIMD pair (n_i, n_j) is covered by a SIMD instruction I_1, and there is a SIMD pair $(n_k, n_l) \in P$ with $n_k \in succ(n_i)$ and $n_l \in pred(n_j)$, or vice versa, then it must be ensured, that n_k and n_l are *not* packed into another SIMD instruction I_2 (fig. 8). Otherwise, the resulting code could not be scheduled, since I_2 would need to be executed both before and after I_1. For any SIMD pair (n_i, n_j), let $X_{ij} \subset P$ denote the set of SIMD pairs (n_k, n_l), such that $n_k \in succ(n_i)$ and $n_l \in pred(n_j)$, or $n_k \in pred(n_i)$ and $n_l \in succ(n_j)$. Then, for (n_i, n_j) and any $(n_k, n_l) \in X_{ij}$, we specify the following constraint to avoid scheduling deadlocks:

$$y_{ij} + y_{kl} \leq 1$$

3.8 Objective function

For an optimized code selection under the above correctness constraints, the number of selected SIMD instructions must be maximized across the entire DFG G. For any node n_i, let $S(n_i) = R_{hi}(n_i) \cup R_{lo}(n_i) \subset R(n_i)$ denote the subset of rules for n_i operating on a sub-register. Then, we maximize the following objective function:

$$f = \sum_{n_i \in G} \sum_{r_j \in S(n_i)} x_{ij}$$

This task can be performed with any ILP solver. For our experiments we have used "lp_solve" [17]. The 0/1 binding of the x_{ij} solution variables accounts for the detailed code selection and thus allows to emit assembly code for the DFG.

3.9 Experimental results

For an experimental evaluation, we have implemented code selectors for the Texas Instruments C6201 [11] and the Philips Trimedia TM1000 [12]. We have compiled ANSI C source codes into assembly code for several signal processing kernel routines, which mainly consist of one finite loop. "vector add" is the example from fig. 6, "image compositing" is taken from [18], and the remaining sources are from the DSPStone suite [19].

Note that it was not the purpose of our experimentation to compare the code quality gain achieved by a certain SIMD instruction set. Instead, we mainly wanted to show that code selection with SIMD instructions frequently does result in higher code quality, and more important, that exploitation of SIMD instructions in compilers is possible without compiler intrinsics and assembly libraries. This is reflected by the fact, that we compiled *identical* sets of *plain* ANSI C source codes to the two different target processors.

source	data type	unroll	without SIMD	with SIMD	CPU
TI C6201					
vector add	short	1	8	4	0.7
IIR filter	short	0	21	17	2.9
convolution	short	1	8	6	0.6
FIR filter	short	1	15	11	0.9
N complex updates	short	1	20	16	3.0
image compositing	short	1	14	11	3.1
Trimedia TM1000					
vector add	short	1	8	4	0.7
IIR filter	short	0	22	22	5.1
convolution	short	1	8	8	0.9
FIR filter	short	1	15	9	0.9
N complex updates	short	1	20	20	4.7
image compositing	short	1	14	7	3.2
vector add	char	3	16	4	5.0
FIR filter	char	3	36	18	26.5

Table I. Experimental results: code selection with SIMD instruction

The experimental results for the TI C6201 are listed in the upper part of table I. The unrolling factor specifies the number of duplications of the loop body, which is necessary to exhibit enough parallelism for exploitation of SIMD instructions. Columns 4 and 5 give the number of generated machine instructions for the loop body without and with exploitation of SIMD instructions. Column 6 mentions the required CPU seconds (Sun Ultra-1, including both DFG covering and ILP solving) when using SIMD instructions.

The TI C6201 shows a comparatively limited support for SIMD instructions, essentially parallel additions and subtractions on 16-bit sub-registers. Therefore, all experiments have been carried out with 16-bit "short" data types. The maximum reduction in instruction count (50 %) was obtained for the "vector add" example, since using the C6201 SIMD instructions permits to unroll the loop once without increasing the instruction count.

The lower part of table I shows the corresponding results for the Trimedia architecture. While for some source codes, such as the "IIR filter" and "convolution", SIMD instructions were not applicable, the code quality gains for "FIR filter" and "image compositing" were more significant as compared to the C6201. This is due to the more powerful SIMD capabilities of the Trimedia (e.g. special instructions for FIR computations), which become particularly obvious for certain algorithms on 8-bit char data. As shown for the "vector add" and "FIR filter" examples, the use of SIMD instructions for char data results in a reduction of instruction count of 75 % and 50 %, respectively.

Even though we use ILP for a part of code selection, the runtime consumed by our approach is moderate if the DFGs to be compiled are not too large. This is a consequence of the fact, that most decisions concerning code selection are already made during the DFG covering phase, which only takes polynomial time in the DFG size. The largest example (FIR filter on char data), whose DFG comprises 95 nodes, took 26.5 CPU seconds. We believe that this is acceptable for embedded applications and systems-on-a-chip, where code quality is of much higher concern than compilation speed. Current limitations of the code selection technique concern the required memory alignment and determination of the optimum loop unrolling factor, which so far have to be performed manually.

4. CODE SELECTION FOR IRREGULAR DATA PATHS

In this section we consider code selection approaches for DFGs mapped to irregular data paths, such as commonly found in fixed-point DSPs. The approach is based on *constraint logic programming* (CLP [20]). CLP foundations have already been described in [10]. There we focused on a specific phase coupling approach of code selection with register allocation and instruction scheduling. In this section we give a detailed description of extended features and opportunities of the code selection approach. It provides a framework to easily and quickly derive new code selection techniques, comprising phase integration of code selection with other sub-tasks of code generation. For embedded processors, development of new code generation techniques is mandatory, because new processors have special features which also require special code generation techniques, in order to generate high quality code. Different processors may require different degrees of phase coupling, but also stand-alone code selection techniques may be required. The approach we present here provides a unified model for developing exact and heuristic techniques, while handling features like: irregular data transfer paths, chained (complex) operations, graph-shaped instruction patterns, and restricted instruction level parallelism. We first describe a concept for modeling alternative DFG covers, which is the basis of our approach. We then demonstrate how this concept is used to easily define several strategies for optimal and heuristic DFG code selection.

4.1 Constraint Logic Programming

In contrast to the DFT based approach using tree parsing, optimal code selection for DFGs is NP-complete. In the last years, constraint logic programming has been successfully applied to solving instances of quite large NP-complete problems, where other approaches failed [23]. A basic concept is to model problems as *constraint satisfaction problems* (CSPs). CSPs are represented by a set of variables and a set of constraints which define mutual dependencies between the variables. Each variable is associated with a certain domain (a set of values) containing the possible candidates for the variable, that can occur in a solution of the problem. The variables are therefore called *domain variables* but we simply use the notion *variable* throughout the paper. We will denote a domain member of a domain variable V as an element of V. A simple CSP is given by variables X and Y, both associated with the domain $\{1,..,4\}$, and the constraint set $\{Y < 4, X < Y\}$. A solution for a CSP is a mapping of the variables to domain elements, such that all constraints are met. One approach for finding a solution is to assign elements to variables in a certain order, which is called *labeling*. Throughout labeling, the constraints ensure feasibility and trigger backtracking in case of constraint violations. Effective search is supported by applying good pruning techniques throughout labeling (e.g. *constraint propagation*). In our implementation we use ECLiPSe [22], which is an extension of the logic programming language PROLOG. ECLiPSe provides a library for finite domains, comprising predefined constraints and labeling strategies. Furthermore, there are generic optimization procedures, expecting a labeling strategy $l(V)$ over variable set V together with an objective function $cost(V)$ as an input (e.g. $minimize(l(V), cost(V))$). The user may define custom constraints and problem specific labeling and optimization strategies. In ECLiPSe, constraints are handled in the background, i.e. consistency checking and backtracking are performed automatically by the runtime system.

4.2 Representing alternative covers

An essential point in our code selection methodology is a new representation of all alternative machine operation covers of a DFG via mapping the covering problem into a CSP. We make use of a compact representation capable of representing a set of machine operations within a single representation, by means of a *factored machine operation* (FMO[3]). A FMO is given by the tuple $(Op, R, [O_1, \ldots, O_n], ERI, Cons)$: In a first step Op denotes an operation available on the processor (we will extend this view to sets later). R and each O_i are variables whose elements are the available alternative storage resource locations (SRs[4]) for the result and the operands, to which the functional units providing operation Op have access. ERI is a set of entities specifying further resources used by machine operations, e.g. the variable FU specifies the alternative functional units on which Op can be executed. Generally, not all assignments of the variables of a FMO to elements will yield machine operations actually available on the target processor. Therefore, we introduce a set of constraints $Cons$ for each FMO, describing mutual dependencies between the variables, actually reflecting the machine constraints for resource usage in legal machine operations of the processor. For a certain target processor the initial domains and the constraints for FMOs can be derived statically and independently from any DFG. FMO templates are provided, from which instances of the constraints and initial domains of variables are generated[5].

A first step in code generation is to associate each node $n_k \in DFG$ with a corresponding FMO_k. Operations of a DFG node generally need not to be operations

[3]In [10] we used the notion factored register transfer operation (FRT) but we think that the notion FMO is more adequate.

[4]SRs comprise the set of register files and memories of a processor.

[5]In [10] the details of how FMOs are used to specify target processors are described. The mechanisms for specifying constraints in ECLiPSe allow an elegant and concise methodology for specifying the instruction set of a processor.

available on the target machine. Furthermore, there often is a set of matching FMOs at a DFG node, obtained when algebraic transformations are applied. Therefore, the FMO templates are extended by a variable SOp denoting the operations that can be found in a DFG. Op is extended to a set of operations available on the processor, either matching SOp immediately or after applying algebraic transformations to SOp. These extensions impose additional constraints relating the elements of Op to elements of the other variables of the FMO[6].

To instantiate an FMO_k, the FMO templates are accessed via a interface procedure $fmo(SOp_k, Op_k, R_k, [O_{k,1}, .., O_{k,m}], ERI_k)$ ($O_{k,j}$ denotes the j-th operand of node n_k). The constraint set $Cons_k$ is handled implicitly and all further constraint handling of the constraints in $Cons_k$ is performed automatically by the ECLiPSe system in background. We now consider the data flow between node n_d and node n_u, such that there exists an edge $(n_d, n_u) \in edges(DFG)$. The function $def(n_u, i)$ yields n_d, such that n_d produces operand i of n_u. For a DFG it now has to be ensured, that there exists a transfer path from R_d to $O_{u,i}$ if $def(n_u, i) = n_d$. This is ensured by transfer constraints which are also specified as templates and are accessed via the interface function $\rightarrow^* (R_d, O_{u,i})$. Note that it only has to be ensured that for each element $rf_1 \in R_d$ there is at least one element in $rf_2 \in O_{u,i}$ with either $rf_1 = rf_2$ or there exists a sequence of data transfers so that values can be moved from rf_1 to rf_2.

A DFG covered with FMOs reflects the available alternative machine operations at each node. The constraints ensure the selection of legal machine operations at each node so that only resources are selected that meet the machine constraints. The constraints also ensure the existence of at least one legal transfer path between the definition of values and their uses. Furthermore, the following features are supported by the model:

—Representation and handling of chained operations, e.g. the MAC operation, which also allows CSEs to be a sub-operation of a chained operation. Chained operations are modeled by introducing virtual SRs (VSRs) for result locations of FMOs which may be a sub-operation of other operations. These virtual locations can be interpreted e.g. as wires or latches (for details see [10]).

—A large class of restrictions on instruction level parallelism can already be modeled by variables $FU, IT \in ERI$ defining the alternative functional units and instruction word types available for an FMO.

—Modeling of graph patterns, like the square function, which is enabled by including constraints over the graph structure, that impose relations between different FMOs. Therefore the actual DFG node is also passed to the interface function.

—From a result location to a certain operand location, the set of all alternative transfer paths of bounded length can be represented by a single transfer path of FMOs. All transfer paths of length i $(0 \leq i \leq l)$ can be modeled with a single FMO sequence of length l.

[6]The specification methodologies of ECLiPSe allow, that these extensions are specified without any modification to the existing FMO templates. The extended FMO templates are specified as a second layer over the existing ones.

source	run-time	nodes	edges	CSEs
iir filter (iir)	0.62	17	19	2
complex update (cu)	0.96	17	18	4
complec multiply (cm)	0.85	13	14	4
lattice filter (lf)	1.52	23	27	8
t1	2.41	24	38	8
t2	3.04	29	47	7
t3	2.15	21	33	9
t4	7.99	82	153	15

Table II. Runtimes for FRT covering

The set of FMO templates for a processor are denoted as the *FMO model* for the processor. The process of associating a DFG with FMOs and the according constraints is denoted as *FMO covering*. In table II the runtimes for FMO covering for several benchmark DFGs for the Analog Devices ADSP-210x target processor are shown. These comprise some benchmarks of the DSPStone suite [19] (iir,cu, and cm) and some internal benchmarks with large basic blocks (lf and t1-t4). All runtimes are given in UltraSparc CPU seconds. The runtime data indicate that FMO covering is efficient. The table also shows some characteristics of the benchmarks: the number of DFG nodes, edges, and CSEs. One can show that the worst case complexity of FMO covering is $O(N * D^2)$, where N is the number of DFG nodes, and D is the maximum number of SR locations. An important feature of our approach is, that a generally exponential number of alternative covers are stored in a representation of linear size (w.r.t. to the number N of DFG nodes).

4.3 Optimal code selection for DFGs

A DFG covered with FMOs reflects all alternative potential covers of a DFG by instruction patterns. The task of code selection is to select machine operations and data transfer paths, so that a certain cost function is minimized. This is performed by assigning certain elements to the variables of the FOMs. In this section we consider optimal code selection for DFGs of each basic block of a program with respect to a sequential instruction execution model, i.e., neglecting instruction-level parallelism (we have also extended this model to take into account instruction level parallelism - see section 4.5), and we show the improvements in code quality as compared to the DFT based code selection approach. Costs are given in instruction cycle counts. Each FMO_k is associated with variable C_k and one variable $TC_{k,i}$ for each $O_{k,i}$. C_k is initialized with the sum of the costs of each legal machine operation of FMO_k. $TC_{k,i}$ reflects the transfer costs associated with $\rightarrow^* (R_d, O_{k,i})$. According to the set of possible transfer paths from elements of R_d to elements of $O_{k,i}$ we initialize $TC_{k,i}$ with the sum of minimal transfer cost to move each element of R_d to an element of $O_{k,i}$.

For all leaf nodes n_v representing a program variable, R_v is set to the possible initial locations at the beginning of a basic block. With each node n_c representing a CSE we associate extra variables R'_c and TC'_c. CSEs rooted at n_c are handled by additionally defining transfer constraints between R_c, R'_c, and between R'_c and the uses from n_c (fig. 9). This allows to define extra locations for CSEs and to combine common prefixes of transfer paths in the cost model. The set CV_{DFG} denotes

source	cost CS1	t_{CS1}	cost CS2	t_{CS2}	cost CS3	t_{CS3}	cost CS4	t_{CS4}
iir	16	0.4	13	1.69	13	1.97	13	0.58
cu	16	0.3	12	4.80	12	5.30	12	0.95
cm	14	0.17	10	6.90	10	4.32	10	0.46
lf	39	0.7	24	235.37	24	687.40	25	4.94
t1	44	0.69	29	27107.61	28	24555.29	28	2.49
t2	42	0.9	28	timeout	26	53296.18	27	2.20
t3	47	0.55	31	1929.09	28	1490.33	30	3.51
t4	148	2.65	96	timeout	88	timeout	92	216.51

Table III. Results of different code selection methods

the set of all cost variables associated with a DFG. The set SR_{DFG} is the set of all variables denoting SRs. Intuitively, optimal DFG covering can be specified by finding a labeling of the variables in $CV_{DFG} \cup SR_{DFG}$, w.r.t. minimizing the costs $Cost(CV_{DFG}) = \sum_{c_i \in CV_{DFG}} c_i$. It can be shown, that it is sufficient to label the set of variables $V_{DFG} = (\bigcup CV_{DFG} \cup \{R'_c | cse(n_c), n_c \in DFG\}$, which drastically reduces the search space. Optimal covering can now be specified, making use of ECLiPSe's optimization procedures: $minimize(l(V_{DFG}), Cost(CV_{DFG}))$. The labeling strategy $l(V_{DFG})$ selects the most constrained variable in each labeling step. We compare three code selection techniques (denoted as CS1, CS2, CS3) for DFGs and provide experimental results, showing how the proposed techniques reduce the code overhead as compared to DFT based covering.

CS1: Traditional DFT based covering, i.e. DFGs are split into DFTs at CSE nodes. Each DFT_i is covered optimally but separately. A constraint $R'_c \in Mem$ ensures that all CSE nodes n_c are located in memory. Covering is performed by applying $minimize(l(V_{DFT_i}), Cost(CV_{DFT_i}))$. Effectively, here we simulate the DFT based approach with the CLP technique. **CS2:** Takes into account arbitrary transfer paths of CSEs to their uses. As an example, consider fig. 9, where the insertion of the extra result location R' of CSEs is advantageous: The transfer paths from node 0 to nodes 1 and 2 both require a move from ar to my. Both transfer paths are combined into a single transfer path from R to R', thus reducing the transfer cost from 2 to 1. In CS2 we do not allow CSEs to be mapped to sub-operations of chained operations. Thus, covering is specified by imposing a constraint $cse(n_c) \Rightarrow R'_c \notin VSR$ and applying $minimize(l(V_{DFG}), Cost(CV_{DFG}))$. **CS3:** CSEs are allowed to be mapped to chained operations without extra constraints.

Table III shows the resulting costs (number of RTs) and runtimes t of the DFG code selection methods CS1 to CS3. The improvements from strategy CS1 to CS3 range within 18% – 50% (on average 35%). The costs of benchmarks where runtimes are given as "timeout" are costs computed within a runtime limit of 24 hours, while the optimization did not terminate within 24 hours. Thus, those results are not proven to be optimal. **Note:** The approach yields a set of alternative optimal covers: generally not all variables are strictly bound to elements (e.g. the SR variables) and there are still alternative transfer paths also in the context of bounded transfer costs. Mutual dependencies between the variables with remaining alternative elements are still present in the form of their constraints which are propagated into subsequent tasks.

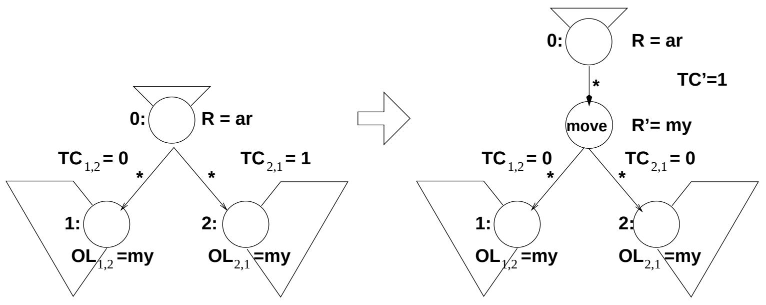

Fig. 9. *Effect of CSE Routing*

4.4 Partitioned DFG code selection

Since optimal DFG code selection is an exponential problem, it is obviously too runtime intensive for large DFGs. We have therefore designed an additional code selection method, where a DFG is split into a set of smaller, manageable DFGs. This method (denoted as **CS4**) leads to much better runtimes than strategies CS2 and CS3, while coming close to the optimal results. In CS4, the strategy for partitioning the DFG is to split the DFG at its CSEs. However, unlike in CS1, the data routes are not constrained to pass a certain location (e.g., the memory in CS1). The basic idea is to postpone the labeling of the data route variables of CSEs to the labeling of those DFGs, which use the corresponding values. We assume a certain labeling order $[DFG_1, ..., DFG_n]$ for the partitioned DFGs. The results of the method CS4 are also shown in table III. The partitioning of DFGs makes it possible to cover large DFGs in acceptable time[7] with results close to the optimal results (only 2% average overhead compared to the optimal method CS3). Further code quality improvements can be achieved by considering different permutations for ordering the DFGs. For instance, labeling the reversed sequence $[DFG_n, ..., DFG_1]$ reduced the average overhead to 1.5% for the considered benchmarks.

4.5 Further Applications of the FMO Model

New constraints can be easily added to the model. Therefore, new problem aspects can be added in a very modular way without having to change the rest of the FMO model. The different code selection strategies (C1-C4) of sections 4.3 and 4.4 were implemented by simply selecting different sets of variables as input for labeling and for the cost function, while the initial FMO model for the processor remained unchanged. In the same way other strategies can be adapted: For instance, in CodeSyn, CBC, and CHESS (see [24]), in a first step of code generation a mapping of source operations to available machine operations is performed, and in a second step chained operations are computed in order to globally minimize the amount of operations. There, heuristics are used instead of considering transfer costs. These techniques are easily adapted in our approach and can even be performed optimally

[7]Runtime can be further reduced as follows: Analyzing the optimization process showed, that results very close to the optimum are generally computed very fast. The rest of the time is only spent for verifying that there is no better solution. This could be exploited by defining appropriate timeouts for the optimization process, and taking the best result computed within this time.

for DFGs of limited size. Other optimization criteria can be accommodated by redefining the cost function (e.g. exchanging the cost variables for instruction cycle counts by code size). Mapping of DFG nodes to sequences of FMOs is also possible. This can be implemented by associating a sequence of FMOs with the FMOs resulting after covering and code selection. Thereby, the result and operand locations must be related to corresponding locations in the sequence of FMOs.

In order to take into account register pressure and restricted instruction level parallelism it is also possible to integrate code selection with register allocation and instruction scheduling. This was implemented in two approaches, both based on the presented FMO model: In [10] code generation phases were still performed in a sequential order, but phase coupling was achieved by delaying several decisions in one task (basically the binding of resources) and to propagate these decisions to the subsequent tasks by means of constraints. In the second approach - for the TI TMS320C5x (not published so far) - constraints for register allocation and instruction scheduling were added to the initial FMO model and solved simultaneously. The optimization goal was to minimize instruction cycles. Both approaches generated code coming close/equal to hand crafted code and showed drastic improvements compared to the code generated by the native commercial compilers.

5. CONCLUSIONS

This paper has concentrated on code selection for certain classes of embedded processors: media processors with SIMD instructions and DSP processors with irregular data paths. We have pointed out that in both cases the traditional DFT based code selection approach is insufficient, and we have described techniques that generalize code selection from DFTs to DFGs in order to generate more efficient code. Code efficiency is an extremely important goal in code generation for embedded processors, which justifies to spend more time for optimization than in the case of general-purpose processors. Taking this into account, we have used unconventional optimization techniques: Integer Linear Programming and Constraint Logic Programming. By a careful modeling of the underlying optimization problems it has been possible to obtain good solutions in a reasonable amount of compilation time. As a net result, we were able to present techniques for two problems that, to our knowledge, have not been solved in previous work: code selection with SIMD instructions and optimal (or near-optimal) DFG code selection for irregular data paths. We believe that such time-intensive, architecture-specific code optimization techniques will be the key to ensure sufficient quality of compiler-generated code for embedded processors. Eventually, this will allow us to take the step from assembly programming to the use of high-level language compilers for embedded processors, which means a productivity boost in the design of software-dominated embedded systems.

ACKNOWLEDGMENTS

This work has been supported by Agilent Technologies, California, and the German Research Foundation DFG.

References

[1] A.V. Aho, M. Ganapathi, S.W.K Tjiang: *Code Generation Using Tree Matching and Dynamic Programming*, ACM Trans. on Programming Languages and Systems 11, no. 4, 1989, pp. 491-516

[2] R. Wilhelm, D. Maurer: *Compiler Design*, Addison-Wesley, 1995

[3] A.V. Aho, S.C. Johnson: *Optimal Code Generation for Expression Trees*, Journal of the ACM, vol. 23, no. 3, 1976

[4] R.S. Glanville: *A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers*, Doctoral thesis, University of California at Berkeley, 1977

[5] 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

[6] M.A. Ertl: *Optimal Code Selection in DAGs*, ACM Symp. on Principles of Programming Languages (POPL), 1999

[7] S. Liao, S. Devadas, K. Keutzer, S. Tjiang: *Instruction Selection Using Binate Covering for Code Size Optimization*, Int. Conf. on Computer-Aided Design (ICCAD), 1995, pp. 393-399

[8] G. Araujo, S. Malik, M. Lee: *Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures*, 33rd Design Automation Conference (DAC), 1996

[9] R. Leupers, P. Marwedel: *Instruction Selection for Embedded DSPs with Complex Instructions*, European Design Automation Conference (EURO-DAC), 1996

[10] S. Bashford, R. Leupers: *Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths*, Design Automation for Embedded Systems, vol. 4, no. 2/3, Kluwer Academic Publishers, 1999

[11] Texas Instruments: TMS320C62xx CPU and Instruction Set Reference Guide, URL http://www.ti.com/sc/c6x, 1998

[12] Philips: URL http://www.trimedia.philips.com, 1998

[13] Intel: MMX Technology Application Notes, URL http://developer.intel.com/drg/mmx/appnotes, 1998

[14] R.J. Fisher, H.G. Dietz: *Compiling for SIMD Within a Register*, 11th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC98), 1998

[15] G.J. Chaitin: *Register Allocation and Spilling via Graph Coloring*, ACM SIGPLAN Symp. on Compiler Construction, 1982

[16] LANCE V1.0 Software and User's Guide, University of Dortmund, Dept. of Computer Science 12, ls12-www.cs.uni-dortmund.de/~leupers, 2000

[17] Eindhoven University of Technology: ftp.es.ele.tue.nl/pub/lp_solve/

[18] A. Peleg, S. Wilkie, U. Weiser: *Intel MMX for Multimedia PCs*, Comm. of the ACM, vol. 40, no. 1, 1997

[19] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[20] K. Marriott and P.J. Stuckey: *Programming with Constraints: An Introduction*, The MIT Press, 1998

[21] M. Wallace: *Constraint Programming*, Contact address: IC-Parc, William Penney Laboratory, Imperial College, London SW7 2AZ, email:mgw@doc.ic.ac.uk, 1995 Publications at http://www.icparc.ic.ac.uk/.

[22] M. Wallace, S. Novello, and J. Schimpf: *ECL^iPS^e: A Platform for Constraint Logic Programming*. Contact address: IC-Parc, William Penney Laboratory, Imperial College, London SW7 2AZ, email:mgw@doc.ic.ac.uk, 1997,

[23] P. Van Hentenryck: *Constraint Satisfaction in Logic Programming*, The MIT Press, 1989

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