Code Generation for Fixed-Point DSPs

GUIDO ARAUJO
IC-UNICAMP
and
SHARAD MALIK
Princeton University

This paper examines the problem of code-generation for Digital Signal Processors (DSPs). We make two major contributions. First, for an important class of DSP architectures, we propose an optimal $O(n)$ algorithm for the tasks of register allocation and instruction scheduling for expression trees. Optimality is guaranteed by sufficient conditions derived from a structural representation of the processor Instruction Set Architecture (ISA). Second, we develop heuristics for the case when basic blocks are Directed Acyclic Graphs (DAGs).

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

General Terms: Algorithms

Additional Key Words and Phrases: code generation, register allocation, scheduling

1. INTRODUCTION

Digital Signal Processors (DSPs) are receiving increased attention due to their role in the design of modern embedded systems like video cards, cellular telephones, and other multimedia and communication devices. DSPs are for the most part used in systems where general-purpose architectures cannot meet domain-specific constraints. In the case of portable devices, for example, power consumption and cost may make usage of general-purpose processors prohibitive. The same is true when high-performance arithmetic processing is required to implement dedicated functionality at low cost, as in the case of specific communications and computer graphics applications. The increasing use of these processors has revealed a new set of code-generation problems that are not efficiently handled by traditional compiling techniques. These techniques make implicit assumptions about the regular nature of the target architecture and microarchitec-

Authors' addresses: G. Araujo, Institute of Computing, IC-UNICAMP, Campinas, 6176, Brazil; S. Malik, Department of Electrical Engineering, Princeton University, Olden St., Princeton, NJ 08544.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1998 ACM 1084-4309/98/0400–0136 $5.00

ture—which is rarely the case with DSPs, where irregularities in the microarchitecture are the very basis for efficient computation of specialized functions. Due to hard *on-chip* memory constraints and hard real-time performance requirements, the code generated for these processors has to meet very high quality standards. Since existing compilation techniques are not up to this task, the vast majority of the code is written directly in assembly language. This research is part of a project directed towards developing compilation techniques capable of generating quality code for such processors (*http://ee.princeton.edu/spam*). The implementation of these techniques forms the compiling infrastructure used in this work, which is called the SPAM compiler.

There is a large body of work in code generation for general-purpose processors. Code generation is, in general, a hard problem. Instruction selection for expressions subsumes Directed Acyclic Graph (DAG) covering, which is an NP-complete problem [Garey and Johnson 1979]. Bruno and Sethi [1976] and Sethi [1975] showed that the problem of optimal code generation for DAGs is NP-complete, even for a single register machine. It remains NP-complete for expressions in which no shared term is a subexpression of any other shared term [Aho et al. 1977a]. An efficient solution for a restricted class of DAGs has been proposed in Prabhala and Sethi [1980]. Code generation for expression trees has a number of $O(n)$ solutions, where n is the number of nodes in the tree. These algorithms offer solutions for stack machines [Bruno and Sethi 1975]; register machines [Sethi and Ullman 1970; Aho and Johnson 1976; Appel and Supowit 1987]; and machines with specialized instructions [Aho et al. 1977b]. They form the basis of code generation for single issue, in order execution, general-purpose architectures.

The problem of generating code for DSPs and embedded processors has not received much attention, probably due to the small size of the programs running on these architectures (which enabled assembly programming). With the increasing complexity of embedded systems, programming such systems without the support of high-level languages has become impractical. Many of the problems associated with code generation for DSP processors were first brought to light by Lee [1988; 1989] in a comprehensive analysis of the architecture features of these processors. Code generation for DSP processors has been studied in the past, but it is only recently that a number of interesting articles have tackled some of the important problems in this area. Marwedel [1993] proposed a tree-based mapping technique for compiling algorithms into microcode architectures. Liem [1994] uses a tree-based approach for algorithm matching and instruction selection, where registers are organized in classes and register allocation is based on a left-first algorithm. Datapath routing techniques have also been proposed [Lanner et al. 1994] to perform efficient register allocation. Wess [1990] proposed using *normal form schedule* for DSP architectures, and offered a combined approach for register allocation and instruction selection using the concept of *trellis diagrams* [Wess 1992]. Kolson et al. [1996]

recently proposed an interesting exact approach for register allocation in loops. An overview of current research on code generation for DSP processors, and embedded processors in general, can be found in Marwedel and Goosens [1995].

In this article we propose an optimal two-phase algorithm which performs instruction selection, register allocation, and instruction scheduling for an expression tree in polynomial time for a class of DSPs. The architecture model here (described in Section 2) is of a programmable highly encoded Instruction Set Architecture (ISA), fixed-point DSP processor. Formally, this class is an extension of the machine models discussed in Coffman and Sethi [1983]. In the first pass (Section 3), we perform instruction selection and register allocation simultaneously, using the Aho-Johnson algorithm [Aho and Johnson 1976]. The second pass, described in Section 4, is an $O(n)$ algorithm that takes an optimally covered expression tree with n nodes and schedules instructions such that no *memory spills* are required. A *memory spill* is an operation where the content of a particular register is saved in memory, due to a lack of available registers for some operation, and then reloaded from memory after that operation is finished. Note that a memory *store* operation required by the architecture topology is not considered a memory spill. The proposed algorithm uses the concept of the *Register Transfer Graph* (RTG) that is a structural representation of the datapath, annotated with ISA information. We show that if the RTG of a machine is acyclic, then optimal code is guaranteed for any program expression tree written for that machine. In this case the DSP is said to have an *acyclic datapath*. Since DAG code generation is NP-complete, we develop heuristics for the case of acyclic datapaths (Section 5), which again uses the RTG concept. In Section 6 we show the results of applying these ideas to benchmark programs. Section 7 summarizes our major contributions and suggests some open problems.

2. ARCHITECTURAL MODEL

DSP processors are irregular architectures when compared with their general- purpose counterparts. This section analyzes the main architecture features that distinguish DSPs from general-purpose processors with respect to basic block code generation. It is not our purpose in this section to give a detailed and extensive analysis of these features. A comprehensive analysis of DSP architectures can be found in Lee [1988; 1989] and Lapsley et al. [1996].

DSPs can be classified according to the types of data they use as *fixed-point DSPs* and *floating-point DSPs*. In applications running on a fixed-point DSP, users are responsible for scaling the result of integer operations. This is done automatically in floating-point DSPs. Floating point units are extremely costly in silicon area and clock cycles, so a large number of the systems based on DSPs use fixed-point DSPs (from now on DSP means fixed-point DSP).

DSPs have *on-chip* data memory based on fast static RAMs and *on-chip* non-volatile program ROM. Unlike general-purpose architectures, DSPs are not designed with cache or virtual memory systems, since data and program streams usually fit into the available *on-chip* memories. Because *on-chip* memories are fast and cache misses are not an issue, some DSPs are designed as memory-register architectures [Texas Instruments 1990]. In order to achieve the bandwidth required by its applications, other DSPs architectures provide multiple memory banks [Motorola 1990]. Since performance is an important factor for DSP applications, DSP instructions are usually designed to be fetched in a single machine cycle. In order to achieve this, instructions are encoded to minimize the number of bits they require. In some architectures [Texas Instruments 1990] this is done by means of *data memory pages*, where instructions need only to carry the offset of the data within the current page in order to access it.

The goal of the design of a DSP datapath is to implement those functional units that can speed up costly operations occurring frequently in the processor application domain. A common example of such units is Multiply and Accumulate (MAC). Due to design requirements, DSP designers frequently constrain the interconnectivity between registers and functional units. There are two main reasons for this. First, the desired functionality usually requires a particular datapath topology. Second, broad interconnectivity translates into datapath buses and/or muxes, which results in increased cost and instruction performance degradation.

A large number of DSPs are *heterogeneous register architectures*. These architectures contain multiple register files and instructions that require operands and store the resulting computation in different register files (thus the term heterogeneous). In general-purpose architectures, instructions do not usually restrict the registers they use, provided they come from the same register file (hence operand registers are homogeneous). This considerably simplifies the code generation problem, since it decouples the tasks of instruction selection from register allocation. Due to this property, we say that general-purpose architectures are *homogeneous register architectures*.

Example 1. An example of a DSP architecture is the TI TMS320C25 Digital Signal Processor (DSP) [Texas Instruments 1990], which is considered the target architecture in the rest of this paper. This processor is part of the TI TMS320 family of processors, which makes up a large number of all commercial DSP processors in use today. The TMS320 family is composed of fixed-point processors (TMS320C1x/C2x/C5x/C54x) which are heterogeneous architectures, and also by a number of floating-point homogeneous architecture DSPs (TMS320C3x/C4x). The TMS320C25 processor contains an ISA with specialized memory-register and register-register instructions. It has three separate register-files (a, p, and t) containing a single register each.

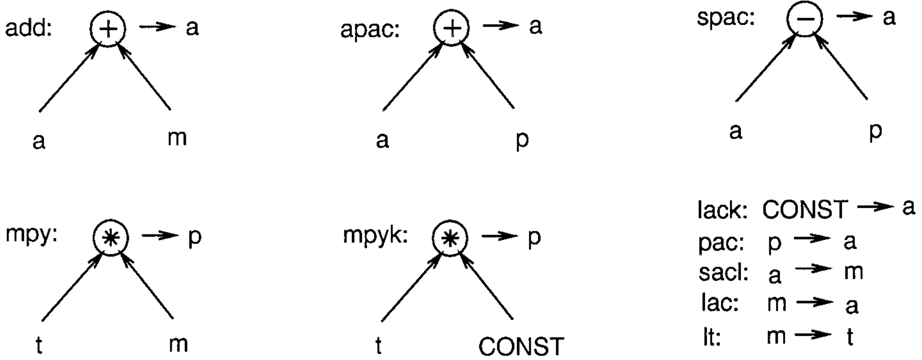


Fig. 1 IR patterns for the TMS320C25 processor.

3. OPTIMAL INSTRUCTION SELECTION AND REGISTER ALLOCATION

In homogeneous register architectures the selection of an instruction has no connection whatsoever with the types of registers that the instruction uses. Selecting instructions for heterogeneous register architectures usually requires allocating registers from specific register-files as operands for particular instructions. The strong binding between instruction selection and register allocation indicates that these tasks must be performed together [Araujo and Malik 1995].

Consider, for example, the *intermediate representation* (IR) patterns in Figure 1, corresponding to a subset of the instructions in the TMS320C25 ISA. In Figure 1, each instruction is associated to a tree-pattern whose node is composed of operations (PLUS,MINUS,MUL), registers (a, p, t), constants (CONST), and memory references (m).

These tree-patterns are represented using the three-address form in Table I. Three-address is a standard compiler representation for instructions, where the destination of the instruction, its two operands (hence the name three address), and the operation it performs are present. Any reference in square brackets is associated to a memory position. Table I also lists the cost associated to each instruction.

Notice that the instructions implicitly define the registers they use. For example, the instruction *apac* can only take its operands from registers a and p, and always computes the result back into a. Observe also that operations that transfer data through datapaths like $lac\ m$ (load register a from memory position m) and pac (move register p into register a) can each be represented as a single node, corresponding to the source register of the transfer operation. The associated cost in this case is only the cost of moving the data from the source register into the destination register. Since registers in DSP architectures are a scarce resource, the final code quality is very sensitive to the cost of routing data through the datapath.

Table I. Partial ISA of the TMS320C25 processor

Instruction	Operands	Destination	Cost	Three Address Form
add m	a,m	a	1	$a \leftarrow a + m$
apac	a,p	a	1	$a \leftarrow a + p$
spac	a,p	a	1	$a \leftarrow a - p$
mpy m	t,m	p	1	$a \leftarrow t^*[m]$
mpyk k	t,k	p	1	$a \leftarrow t^*k$
lack k	k	a	1	$a \leftarrow k$
pac	p	a	1	$a \leftarrow p$
sacl m	a	m	1	$[m] \leftarrow a$
lac m	m	a	1	$a \leftarrow [m]$
lt m	m	t	1	$t \leftarrow [m]$

3.1 Problem Definition

Optimal instruction selection combined with register allocation is the problem of determining the best cover of an expression tree such that the cost of each pattern match depends not only on the number of cycles of the associated instruction, but also on the number of cycles required to move its operands from the location they currently occupy to the location where the instruction requires them to be.

3.2 Problem Solution

A solution for this problem is to use a variation of the Aho-Johnson algorithm [Aho and Johnson 1976] such that at each node we keep not only all possible costs for matches at that node, but also all possible costs resulting from matching the node and moving the result from where it was originally computed into any other reachable location in the datapath.

Tree-grammar parsers have been used as a way to implement code-generators [Aho et al. 1989; Fraser et al. 1993]; Tjiang 1993]. They combine dynamic programming and efficient tree-pattern matching algorithms [Hoffman and O'Donnell 1992] for optimal instruction selection. We have implemented combined instruction selection and register allocation using the OLIVE [Tjiang 1993] code-generator generator, which is based on the techniques proposed in IBURG [Fraser et al. 1993]. OLIVE takes as input a set of grammar rules where tree-patterns are described in a prefixed linearized format. The IR patterns from Table I were converted into the OLIVE description of Figure 2 by rewriting each instruction three-address representation into that format. Notice that the instruction destination registers are now associated to grammar non-terminals and that these are represented by lower-case letters in Figure 2.

Rules 1 to 5 correspond to instructions that take two operands and store the final result in a particular register (a and p, respectively). Rule 6 describes an immediate load into register a. Rules 7 to 10 are associated to data transfer instructions, and bring the cost of moving data through the datapath into the total cost of a match. We point out in Figure 2 that, for

```
a : PLUS(a,m)     {} = {};      (1) add  m
a : PLUS(a,p)     {} = {};      (2) apac
a : MINUS(a,p)    {} = {};      (3) spac
p : MUL(m,t)      {} = {};      (4) mpy  m
p : MUL(t,CONST)  {} = {};      (5) mpyk k
a : CONST         {} = {};      (6) lack k
a : p             {} = {};      (7) pac
m : a             {} = {};      (8) sacl m
a : m             {} = {};      (9) lac  m
t : m             {} = {};     (10) lt   m
```

Fig. 2. Partial OLIVE specification for the TMS320C25 processor (instruction numbers and names on the right are not part of the specification).

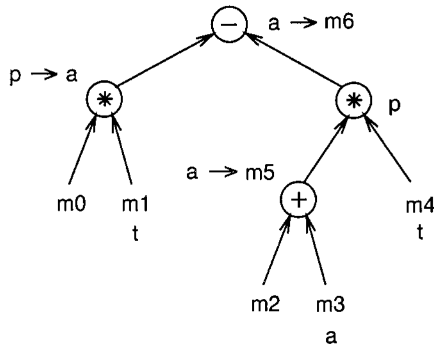
simplicity, we do not represent all patterns corresponding to commutative operations. For example, instruction *add m* can be specified in two ways: PLUS(a,m) and PLUS(m,a). Nevertheless, we consider in the remainder of this paper that all commutative forms of any operation pattern are available whenever required.

If we do not consider instruction scheduling and the associated spills at this point, then the algorithm proposed above is optimal—following from the fact that this algorithm is a variation of the provably optimal Aho-Johnson dynamic programming algorithm [Aho and Johnson 1976].

4. SCHEDULING

Optimal instruction selection and register allocation for an expression tree is not enough to produce optimal code. For optimal code, instructions must be scheduled in such a way that no memory spills are introduced. Note that memory positions allocated in the previous phase are not considered spills. They result from the optimal selection of memory-register instructions in the ISA, and not from the presence of resource conflicts.

Aho and Johnson [1976] showed that, by using dynamic programming, optimal code can be generated in linear time for a wide class of architectures. The schedule they propose is based on their *Strong Normal Form Theorem*, which guarantees that any optimal code schedule for an expression tree, in a homogeneous register architectural model, can always be transformed into *Strong Normal Form* (SNF). A code sequence is in SNF if it is formed by a set of code subsequences separated by memory storage, where each code subsequence is determined by a *Strongly Contiguous* (SC) schedule. A code sequence is a SC schedule if it is formed as follows: at every selected match, m, with child subtrees T_1 and T_2, continuously schedules the instructions corresponding to subtree T_1 followed by the instructions corresponding to T_2, and finally the instruction corresponding to pattern m. Wess [1990] used SNF as a heuristic to schedule instructions for the TMS320C25 DSP.



(a)

(b)		(c)		(d)	
lt m1	t ← [m1]	lt m4	t ← [m4]	lac m3	a ← [m3]
mpy m0	p ← t * [m0]	lac m3	a ← [m3]	add m2	a ← a + [m2]
pac	a ← p	add m2	a ← a + [m2]	sacl m5	[m5] ← a
sacl m7	[m7] ← a	sacl m5	[m5] ← a	lt m1	t ← [m1]
lac m3	a ← [m3]	mpy m5	p ← t * [m5]	mpy m0	p ← t * [m0]
add m2	a ← a + [m2]	lt m1	t ← [m1]	pac	a ← p
sacl m5	[m5] ← a	pac	a ← p	lt m4	t ← [m4]
lt m4	t ← [m4]	sacl m7	[m7] ← a	mpy m5	p ← t * [m5]
mpy m5	p ← t * [m5]	mpy m0	p ← t * [m0]	spac	a ← a - p
lac m7	a ← [m7]	pac	a ← p	sacl m6	[m6] ← a
spac	a ← a - p	lt m7	t ← [m7]		
sacl m6	[m6] ← a	mpyk 1	p ← t * 1		
		spac	a ← a - p		
		sacl m6	[m6] ← a		

Fig. 3. (a) Matched IR tree for the TMS320C25; (b) SNF left-first schedule; (c) SNF right-first schedule; (d) Optimal schedule.

4.1 Problem Definition

SC schedules are not an efficient way to schedule instructions for heterogeneous register set architectures. They produce code sequences whose quality is extremely dependent on the order in which the subtrees are evaluated. Consider for example the IR tree of Figure 3(a). The expression tree was optimally matched using the approach proposed in Section 3 and the target ISA. It takes variables at memory positions m_0 to m_4 and stores the resulting computation into one variable at memory position m_6, using m_5 as temporary storage.

The code sequences generated for three different schedules and its corresponding three-address representation are showed in Figure 3(b-d). Memory position m_7 was used whenever a spill location was required by the scheduler. For the code of Figure 3(b) the left subtree of each node was

scheduled first, followed by its right subtree and then the instruction corresponding to the node operation. The opposite approach was used to obtain the code of Figure 3(c). Neither the SC schedules in Figure 3(b) and (c), nor any SC schedule, will ever produce optimal code. This is obtained using a non-SC schedule that first schedules the addition $m_2 + m_3$ and then the rest of the tree, as in Figure 3(d). Notice that this schedule is indeed an SNF schedule, since first the subtree corresponding to $m_2 + m_3$ is contiguously scheduled followed by a storage operation into memory position m_5, and by another code sequence resulting from a SC schedule of the rest of the tree.

From Figure 3 we can verify how the appropriate SNF schedule minimizes spilling. For example, if the tree of Figure 3(a) is scheduled using left-first, the result of operation $m_0 \times m_1$ is first stored in p and then moved into a. Just after that, register a has to be used to route the result of $m_2 + m_3$ into memory position m_5. But a still contains a live result (the result of $m_0 \times m_1$). In this case, the code generator has to emit code to spill the value of a into memory and recover it later. This would not be required if the scheduler had first stored $m_2 + m_3$ into m_5, before loading a with the result of $m_0 \times m_1$.

Problems like the one above are very common in DSP architectures. The obvious question then is: Does there exist a guaranteed SNF schedule such that no spilling is required ? We prove that this schedule exists under conditions that depend exclusively on the ISA of the target processor. But before doing so, let us define the problem formally: Given an optimally covered expression tree for an heterogeneous register architecture, determine an instruction schedule that does not introduce any spill code.

4.2 Problem Solution

This section is divided as follows. In Section 4.2.1 we state and prove a sufficient condition that a heterogeneous register architecture has to satisfy in order to enable spill-free schedules. In Section 4.2.2 we introduce the concept of a *Register Transfer Graph* (RTG) and show how it impacts the code generation task. Finally, we prove the existence of an optimal linear time scheduling algorithm for a class of DSP architectures that have acyclic RTGs.

Let T be an expression tree with unary and binary operations. Let $L : T \rightarrow R \cup M$ be a function that maps nodes in T to the set $R \cup M$, where $R = \{r_i, 1 \leq i \leq N\}$ is a set of N registers, and M the set of memory locations. Let u be the root of an expression tree, with v_1 and v_2 the children of u. Consider that after allocation is performed, registers $L(v_1) = r_1$, $L(v_2) = r_2$ are assigned to v_1 and v_2, respectively. Let T_1 and T_2 be the subtrees rooted at v_1 and v_2, as in Figure 4. From now on, the terms expression tree and allocated expression tree are used interchangeably, with the context distinguishing whether the tree is allocated or not.

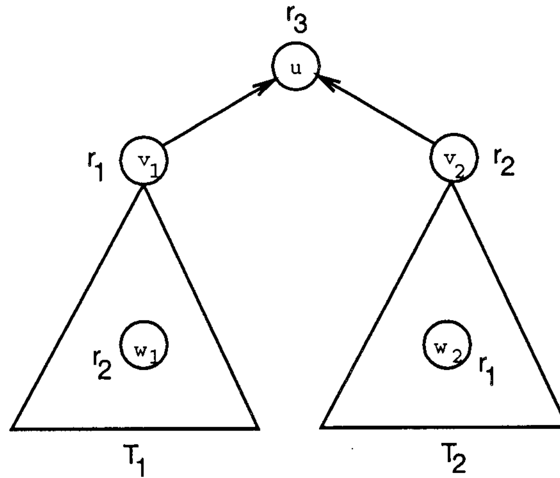


Fig. 4. Allocation deadlock in an expression tree.

4.2.1 *Allocation Deadlock.*

Definition 1. An expression tree contains an allocation deadlock iff the following conditions are true: (a) $L(v_1) \notin M$, $L(v_2) \notin M$, (b) $L(v_1) \neq L(v_2)$, and (c) there exist nodes w_1 and w_2, and $w_2 \in T_2$ such that $L(w_1) = L(v_2)$ and $L(w_2) = L(v_1)$.

The above definition can be seen in Figure 4. This is the situation when two sibling subtrees T_1 and T_2 each contain at least one node allocated to the same register as the register assigned to the root of the other sibling tree. Using this definition it is possible to propose the following result.

THEOREM 1. *Let T be an expression tree. If T does not have a spill–free schedule, then it contains at least one subtree that has an allocation deadlock.*

PROOF. Assume that all nodes u in T are such that T_u is free of allocation deadlocks and that no valid schedule exist for T. According to Definition 1, T_u does not have an allocation deadlock when

- (a) $L(v_1) = M$ ($L(v_2) = M$). In this case, a SNF schedule exists if subtree T_1 (T_2) is scheduled first, followed by subtree T_2 (T_1).
- (b) $L(v_1) = L(v_2)$. This cannot happen because no non-unary operator of an expression tree takes its two operands simultaneously from the same location.
- (c) $L(v_1) \neq L(v_2)$, $L(w_1) = L(v_2)$, but no node w_2 exists for which $L(w_2) = L(v_1)$. In this case it is possible to schedule T_1 first, followed by T_2 and the instruction corresponding to node u. This is a valid schedule

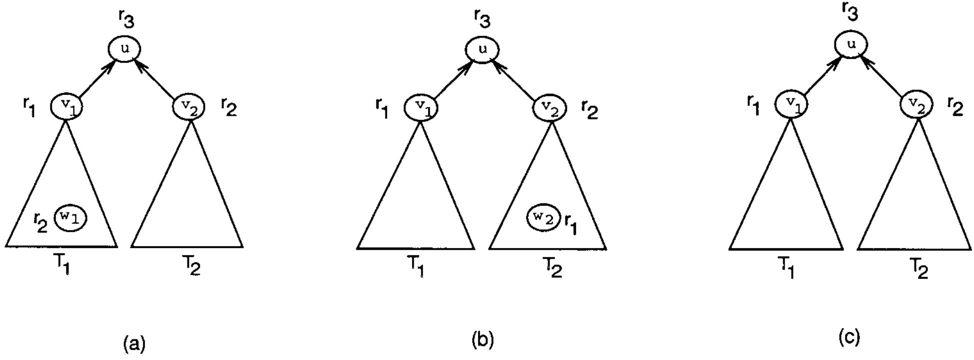


Fig. 5. Trees without allocation deadlock.

because just after the schedule of T_1 is finished, only register r_1 is live, and since no register r_1 exists in T_2, no resource conflict will occur when this subtree is scheduled (Figure 5(a)).

- (d) $L(v_1) \neq L(v_2)$, $L(w_2) = L(v_1)$, but no node w_1 exists for which $L(w_1) = L(v_2)$. This is symmetric to the previous case. Schedule T_2 first, followed by T_1 and the instruction corresponding to u (Figure 5(b)).
- (e) $L(v_1) \neq L(v_2)$, but no nodes w_1 and w_2 exist. This case is trivial, any *SC* schedule results in a spill–free schedule (Figure 5(c)).

Since the above conditions can be applied to any node u, T will have a valid schedule that is free of memory–spilling code. This contradicts the initial assumption. □

Corollary 1. Let T be an expression tree. If T has no subtree containing an allocation deadlock, then it must have a spill–free schedule. Moreover, this schedule can be computed using the proof of Theorem 1.

PROOF. Directly from the theorem above. □

4.2.2 *The RTG Model and Theorem.*

Definition 2. The RTG is a directed labeled graph where each node represents a location in the datapath architecture where data can be stored. Each edge in the RTG from node r_i to node r_j is labeled after those instructions in the ISA that take operands from location r_i and store the result in location r_j.

The nodes in the RTG represent two types of storage: *register files* and *single registers*. Register file nodes represent a set of locations of the same type that can store multiple operands. A datapath single register (or simply single register) is a register file of unitary capacity. Register file nodes are distinguished from single register nodes by means of a double circle. Because of its uniqueness, memory is not described in the RTG. Instead

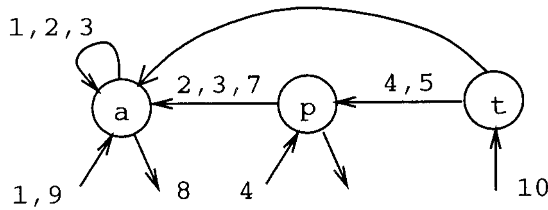


Fig. 6. TMS320C25 architecture has an acyclic RTG.

arrows are used to represent memory operations. An incoming (outgoing) arrow pointing to (from) an RTG node r is associated to a *load* (*store*) operation from (into) memory. Notice that the RTG is a labeled graph where each edge has labels corresponding to the instructions that require that operation. In other words, if both instructions p and q take one operand in r_i and store its result into r_j, then the edge from r_i to r_j will have at least two labels, p and q. We say that an architecture RTG is *acyclic* if it contains no cycles. As a consequence, any register transfer cycle in an acyclic RTG has to go through memory.[1]

Example 2. Consider, for example, the partial OLIVE description in Figure 2. The RTG of Figure 6 was formed from that description. The numbers in parenthesis on the right side of Figure 2 are used to label each edge of the graph. Not all ISA instructions of the target processor are represented in the description of Figure 2, and therefore not all edges in the RTG of Figure 6 are labeled. Notice that the RTG of the TMS320C25 architecture is acyclic. Other DSP processors also have acyclic RTGs, like the processors TMS320C1X/C2X/C5X and the Fujitsu FDSP-4. This paper proposes a solution for code generation for acyclic RTG architectures. Unfortunately, other known DSPs like the ADSP-2100 and the Motorola 56000 have cyclic RTGs. Nevertheless, as shown later, code generation for these processors can also benefit from the results of this work.

THEOREM 2. *If an architecture RTG is acyclic, then for any expression tree there exists a schedule that is free of memory spills.*

PROOF. Let T be an expression tree rooted at u, and v_1 and v_2 be its children such that $L(u) = r_3$, $L(v_1) = r_1$ and $L(v_2) = r_2$. Let T_1 and T_2 be the subtrees rooted at nodes v_1 and v_2. Let P_k, ($k = 1,2, \ldots$) be subtrees of T with root p_k for which the result of operation p_k is stored into memory (i.e., $L(p_k) = M$). Define Q_i ($i = 1,2$) (dark areas in Figure 7) as the subtrees formed in T_i after removing all nodes from subtrees P_k. We show that if the RTG is acyclic, an optimal schedule can always be determined by properly ordering the schedules for P_k (e.g., P_1, P_2, P_3, P_4) and Q_1, Q_2. Here we have to address two cases: (a) Assume that T has no allocation deadlock,

[1]Observe that a self-loop is not considered an RTG cycle.

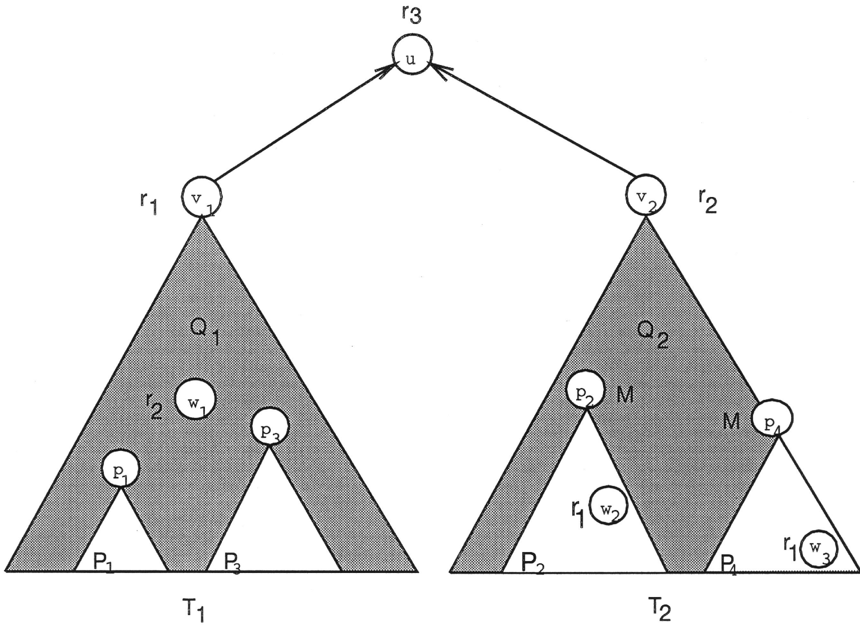


Fig. 7. The RTG theorem.

therefore from Corollary 1, T has an optimal schedule; (b) now consider that an allocation deadlock is present in T and that it is caused by registers r_1 and r_2, shown in Figure 7. Assume also that there exist paths from r_2 to r_1 in the processor RTG. Observe that for each node in T_2 (Figure 7) allocated to r_1, e.g., w_2, the path that goes from w_2 to its ancestor v_2 (allocated to r_2) will necessarily pass by a node allocated to memory, e.g., p_2. This results from the fact that any path from r_1 to r_2 has to traverse memory, given that the RTG is acyclic and that it contains paths from r_2 to r_1. Notice that one can recursively schedule subtrees P_2 and P_4 in T_2 for which the root was allocated to memory, and that this corresponds to emitting in advance all instructions that store results in r_1. Once this is done, only memory locations are live, and the remaining subtree Q_2 contains no instruction that uses r_1. The nodes that remain to be scheduled are those in subtrees T_1 and Q_2. Therefore, the tree $T_1 \cup Q_2 \cup \{u\}$ can now be scheduled using Corollary 1 and no spill is required. Note that the same result is obtained if one first recursively schedules all subtrees P_1, P_2, P_3, P_4 (white areas in Figure 7), followed by applying Corollary 1 to schedule subtree $Q_1 \cup Q_2 \cup \{u\}$. □

Based on the proof of Theorem 2 above, an algorithm can be designed that computes the best schedule for an expression tree in any acyclic RTG architecture. We have designed such an algorithm and named it *OptSchedule*.

THEOREM 3. *Algorithm OptSchedule is optimal and has running time $O(n)$, where n is the number of nodes in the subject tree T.*

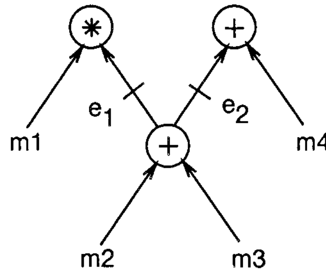
PROOF. The first part is trivial, since *OptSchedule* implements the proof of Theorem 2. Also from Theorem 2, the algorithm divides T into a set of disjoint subtrees ($P_1 - P_4$, Q_1, Q_2) and recursively schedules each of them. Therefore, every node in T is visited only once. Hence, the algorithm running time is $O(n)$. □

If the RTG is acyclic for a particular architecture, then optimal sequential code is guaranteed for any expression tree compiled from programs running on that architecture. Unfortunately, this is not true for those architectures that do not have acyclic RTGs. Nevertheless, expression trees in those architectures can also benefit from this work. Observe from Corollary 1 that if an expression tree is free of allocation deadlocks, then it can be optimally scheduled. This is valid for any expression tree generated from any architecture, no matter whether this architecture has an acyclic RTG or not. Consider for example that a path is added from p to t in the RTG of Figure 6. This creates a cycle in the architecture RTG, which does not go through memory. On the other hand, any expression tree that does not use this new path is free of allocation deadlocks, and so can still be optimally scheduled. Such expression trees could be identified by a simple modification of the instruction selection algorithm. The question of how many of these trees exist in a typical program is still open.

5. HEURISTIC FOR DAGS

Instruction selection for an expression DAG requires DAG covering, which is known to be NP-complete [Garey and Johnson 1979]. In practical solutions to this problem, heuristics have been proposed that divide the DAG into its component trees by selecting an appropriate set of trees. However, dismantling the DAG into component trees is not unique, and there are several ways in which it can be done. Traditionally, in homogeneous register architectures, the heuristic is to disconnect multiple fanout nodes of the DAG [Aho et al. 1988].

Dividing a DAG into its component trees requires disconnecting (or *breaking*) edges in the DAG. For code generation, breaking a DAG edge between nodes u and v implies the allocation of temporary storage to save the result of operation u, while it is not consumed by operation v. Storage is usually in memory, but it can be in any location in the datapath. Our key idea is a heuristic that uses architectural information from the RTG in the selection of component trees of a DAG such that the resulting code has minimal spills. Consider for example the DAG of Figure 8. Notice that two different approaches can be used to decompose this DAG into its component trees, depending on which edge (e_1 or e_2) is selected for breaking. From now on, we show a broken edge by a line segment traversal to the subject edge. As shown in Figure 8(b), one extra instruction is generated when the



(a)		(b)	
lac m2	a ← [m2]	lac m2	a ← [m2]
add m3	a ← a + [m3]	add m3	a ← a + [m3]
sacl m5	[m5] ← a	sacl m5	[m5] ← a
lt m1	t ← [m1]	lac m5	a ← [m5]
mpy m5	p ← t * [m5]	add m4	a ← a + [m4]
add m4	a ← a + [m4]	lt m1	t ← [m1]
		mpy m5	p ← t * [m5]

Fig. 8. (a) Breaking edge e_1; (b) breaking edge e_2.

dismanteling heuristic is based on breaking edge e_2 instead of e_1. Incidentally, the code in Figure 8(a) is also the best sequential code that can be generated from the subject DAG. From the architectural description in Table I, note that the multiplication operation requests its operands from memory (m) and t, and that the result of the addition operation always produces its result in the accumulator a.

See also in Figure 6 that to bring any data from a to register t we have to go through memory. From Figure 8 we can see that the result of the addition operation $m_2 + m_3$ has to be stored in a and must be moved to memory or t in order to be used as an operand of the multiplication operation. But to move data from a to t, we have to go through memory. Suppose the memory position selected to store this temporary result is m_5. Hence, by breaking DAG edge e_1, we are just assigning in advance a memory operation that will appear on that edge during the instruction selection phase of code generation. Note that the existence of a register-transfer path that always goes through memory whenever data is moved from a to t is a property of the target datapath. Similarly, the register-transfer path from a to p must also pass through memory.

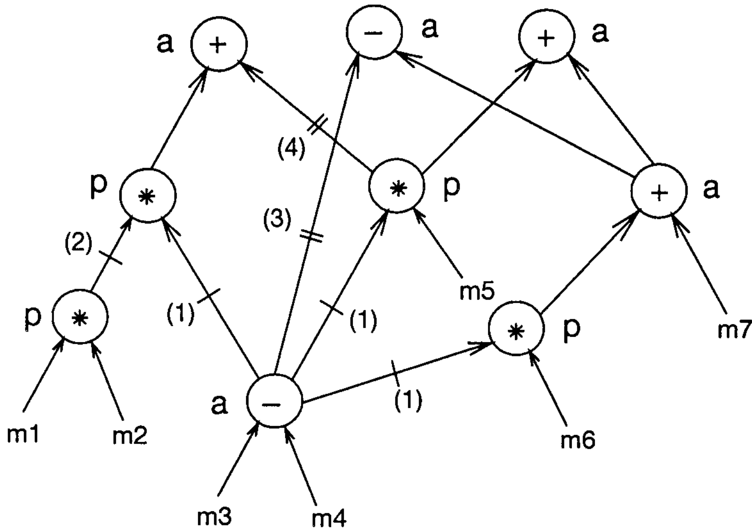


Fig. 9. Expression DAG after partial register allocation was performed and natural and pseudo-natural edges identified by its corresponding lemma.

Notice also that when edge e_2 is broken, pattern PLUS(a,m) (instruction *add* m_4) cannot be used to match the addition of m_4 with the result of $m_2 + m_3$ in the accumulator a. In this case, instruction *lac* m_5 in Figure 8(b) has to be issued in order to bring the data from m_5 back to the accumulator, adding a new instruction to the final code.

5.1 Problem Solution

The heuristic we propose to address the problem just described is divided into four phases. In the first phase (Section 5.1.1), partial register allocation is done for those datapath operations that can be clearly allocated before any code generation task is performed in the DAG. During the second phase (Section 5.1.2), architectural information is employed to identify special edges in the DAG that can be broken without introducing any loss of optimality for the subsequent tree mapping stages. In the third phase (Section 5.2), edges are marked and disconnected from the DAG. Finally, component trees are scheduled and optimal code generated for each component tree (Section 5.2).

5.1.1 *Partial Register Allocation.* A general property of heterogeneous register architectures is that the results of specific operations are always stored in well–defined datapath locations. This does not imply total register allocation because data has to be routed through the datapath to locations required by other instructions. Take, for example, operations *add* and *mul* in the target processor. Note that they implicitly define the primary storage resources used for the operation result. In this case (see

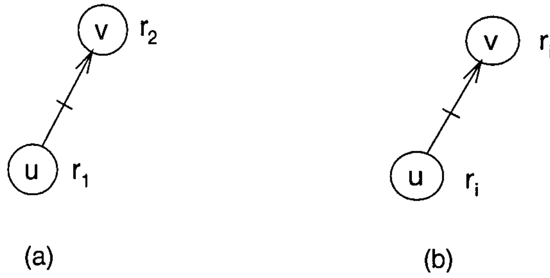


Fig. 10. Natural edges are identified by a single line segment: (a) (u, v) is natural; (b) (u, v) is natural if r_i has no self-loop in the RTG.

Table I), no register allocation task is required to determine that registers a and p are used, respectively, to store the immediate result of operations add and mul. Thus partial allocation can be performed well in advance, even before breaking the edges of the expression DAG takes place. Again, this is only possible when an operation uses the same register file to store its immediate result. In the expression DAG of Figure 9, partial register allocation can be performed immediately for registers a and p.

5.1.2 *Natural Edges.* We see in Figure 8 that some edges have specific properties originating in the target architecture, which allows us to disconnect them from the DAG without compromising optimality. These edges, called *natural edges*, are defined as follows.

Definition 3. If the matching of edge (u, v) always produces a sequence of data transfer operations (in the datapath) that passes through memory, edge (u, v) is a natural edge.

Now given an expression DAG D and a target architecture with an acyclic RTG, it can be shown that a number of edges in D are natural edges. In order to do so, we state a set of lemmas.

Let r_1 and r_2 be a pair of registers in the datapath of an acyclic RTG architecture. And let $L : D \rightarrow R \cup M$ be a function that maps nodes in D into the set of datapath locations $R \cup M$, where R is the set of registers in the datapath and M the set of memory positions.

LEMMA 1. *Let r_1 and r_2 be registers in the architecture RTG such that there exists no path from r_1 to r_2. Therefore, any edge (u, v) in D for which $L(u) = r_1$ and $L(v) = r_2$ is a natural edge.*

PROOF. Given that a path from registers r_1 to r_2 will be traversed whenever instruction selection is performed on edge (u, v), a memory operation will always be selected during instruction selection on (u, v), and therefore (u, v) is a natural edge (Figure 10(a)). □

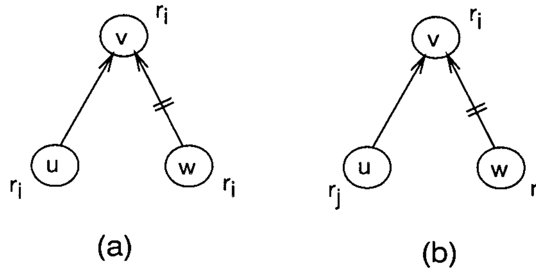


Fig. 11. The selected pseudo-natural edges are identified by a double line segment: (a) one of the edges uses a loop in the RTG; (b) one of the edges goes through memory.

LEMMA 2. *Edges (u, v), for which $L(u) = L(v) = r$ $i = 1,2, \ldots |R|$, are natural edges only if no self-loop exists on register node r_i in the RTG representation of the target architecture (Figure 10(b)).*

PROOF. If an architecture has an acyclic RTG, then any loop in the RTG (which is not a self-loop) will traverse memory. Thus, if register r_i has no self-loop in the RTG, then any loop starting at r_i will go through memory. Therefore, a memory operation will be selected whenever instruction selection is performed on edge (u, v). Hence (u, v) is a natural edge. □

Note that the task of breaking natural edges does not introduce any new operations into the DAG because, during the instruction selection phase, a memory operation is selected naturally, owing to constraints in the architecture datapath topology. As a result, no potential optimality is lost by breaking natural edges.

Example 3. Consider each one of the lemmas above and the RTG of Figure 6. Observe the expression DAG of Figure 9 after natural edges have been identified.

(1) From Lemma 1 we can see that when $r_1 = a$ and $r_2 = p$, every edge (u, v) such that $L(u) = a$ and $L(v) = p$ is a natural edge.

(2) Now consider Lemma 2. First take the situation where $r_i = p$. From the RTG of Figure 6, observe that register p has no self-loop. Since the RTG is acyclic, any DAG edge (u, v), where $L(u) = L(v) = p$, is a natural edge. Now consider the case where $r_i = a$. Register a in Figure 6 contains a self-loop, thus nothing can be said about these edges.

5.1.3 *Pseudo-Natural Edges.* In the following two lemmas we show that DAG edges can sometimes interact such that one edge out of a set of two must result in storage in memory. The edges in this set are called *pseudo-natural edges*.

LEMMA 3. *Consider operation v and its operand nodes u and w in Figure 11 (a). If partial register allocation of these operations is such that* $L(u) = L(v) = L(w) = r_i$, $i = 1, \ldots |R|$, *then edges* (u, v) *are* (w, v) *pseudo-natural edges.*

PROOF. Note that no binary operation v can take both its operands simultaneously from the same register. We have to consider two situations:

(a) If node r_i has a self-loop in the architecture RTG, one of the edges, e.g., (u, v), could be matched by an instruction that takes one operand from r_i. On the other hand, when this same instruction matches the other edge, i.e. (w, v), it will make use of a register contained in an RTG loop (not a self-loop) that goes from r_i back to r_i. As in Lemma 2, matching (w, v) will introduce a sequence of transfer operations that necessarily go through memory, making (w, v) and (u, v) pseudo-natural edges.

(b) If no self-loop node r_i exists in the architecture RTG, then both edges are natural edges according to Lemma 2. □

LEMMA 4. *Consider operation v and its operand nodes u and w of Figure 11(b). Let the partial register allocation of these nodes be such that* $L(u) = L(w) = r_j$ *and* $L(v) = r_i$. *If all RTG paths between each pair of nodes are such that only one path does not go through memory, then* (u, v) *and* (w, v) *are pseudo-natural edges.*

PROOF. The proof is trivial and follows from the fact that since operation v cannot take both of its operands from the same register r_j at the same time, it has to use two paths in the RTG to bring data from register r_j. Since only one path from r_j to r_i does not go through memory, then the other path has to pass through memory. □

Based on the lemmas above, we need to decide which edge between (u,v) and (w, v) is to be disconnected from the DAG. Loss of optimality might occur, depending on which edge is selected. The selected pseudo-natural edge is identified using a double line segment to distinguish it from natural edges. Unlike natural edges, breaking pseudo-natural edges might result in compromising the optimality of code generation for the component trees. However, there is a good chance that this might not happen in actual practice.

Example 4. Consider Lemmas 3 and 4 and the RTG of Figure 6. Observe the expression DAG of Figure 9 after pseudo-natural edges have been identified.

(3) Lemma 3 is satisfied for the case where $r_i = a$ or $r_i = p$.

(4) In this case, if $r_j = p$ and $r_i = a$, only one path exists in the RTG from p to a that does not go through memory.

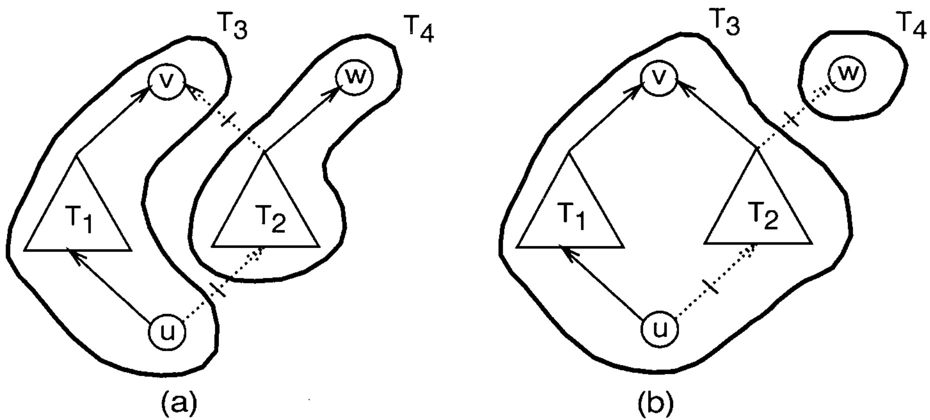


Fig. 12. (a) Cyclic RAW dependency; (b) constraining the tree scheduler.

After rules 1 – 4 of Examples 3 and 4 are applied, the expression DAG of Figure 9 results. Each marked edge in Figure 9 has on its side the number corresponding to a rule from Examples 3 and 4.

5.2 Dismantling Algorithm

The task of dismantling an expression DAG may potentially introduce cyclic *Read After Write* (RAW) dependencies between the resulting tree components, leading to an impossible schedule. A similar problem was also encountered in Aho et al. [1977a] and in Liao et al. [1995] when these authors studied the problem of scheduling *worm-graphs* derived from DAGs in single-register architectures. Consider, for example, the reconvergent paths from nodes u to v and the component trees T_1 and T_2 of Figure 12(a). Dismantling the DAG of Figure 12(a) requires that at least one of the edges of the multiple fanout nodes u and T_2 be disconnected. Assume that edges (u, T_2) and (T_2, v) have been selected as the edges to be broken. In this case, nodes u, v and tree T_1 can be collapsed into a single component tree T_3, dismantling the DAG into trees T_3 and T_4. When an edge between two nodes is broken, a RAW edge is introduced (dashed lines in Figure 12), in order to guarantee that the original data dependencies are preserved by the scheduler. In this case, the resulting RAW edges form a cycle between component trees T_3 and T_4, which results in an infeasible schedule for the component trees.

Note that dismantling is also possible if edge (T_2, w) is broken instead of (T_2, v) (Figure 12(b)). When this occurs, RAW edge (u, T_2) is brought into the resulting component tree (T_3). As a consequence, the potential optimality of the tree scheduler algorithm *OptSchedule* cannot be guaranteed anymore, since now it has to satisfy the constraint imposed by the new RAW edge inside T_3. A possible solution is to modify the tree scheduler

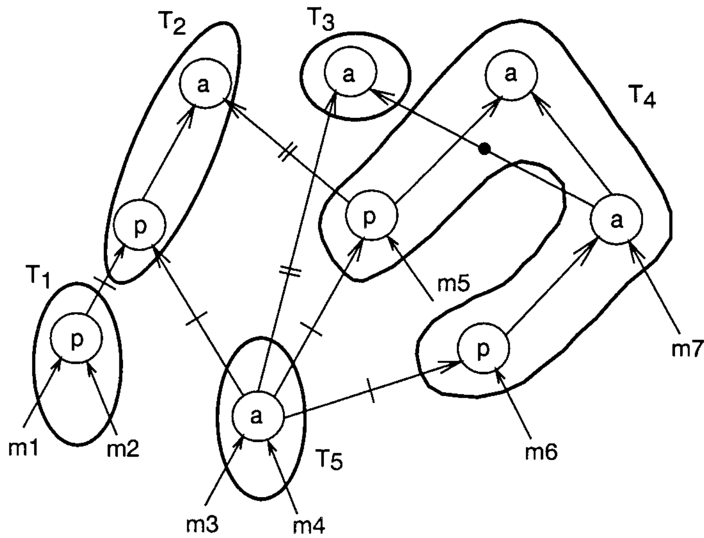


Fig. 13. Resulting component trees after dismantling.

algorithm such that it can satisfy any RAW constraint inserted into the tree. Unfortunately, this is a very difficult task for which an efficient solution does not seem to exist. Hence, we have to dismantle the DAG to avoid inserting RAW edges into the component trees.

From the two situations analyzed above, we conclude that edges on both reconvergent paths have to be disconnected in order to guarantee proper scheduling of operations inside and between component trees. An algorithm that dismantles the DAG should disconnect edges by using as many natural and pseudo-natural edges as possible. We have designed such an algorithm, which we call *Dismantle*.

The *Dismantle* algorithm starts by first breaking all natural edges, since breaking them adds no cost to the total cost for the final code. After this *Dismantle* proceeds to identify reconvergent paths. It traverses paths in the DAG looking for edges marked as pseudo-natural edges. If a pseudo-natural edge can be used to break an existing reconvergent path, the edge is broken. Otherwise the outgoing edge that starts the reconvergent path at the corresponding multiple fanout node is broken. These edges are marked with a black dot in Figure 13. At this point all reconvergent paths in the expression DAG have been disconnected. Additional edges are then broken such that no node ends up with more than one outgoing edge (these edges are also marked with black dots). The resulting DAG is shown in Figure 13. It decomposes the original DAG into five expression trees ($T_1 - T_5$). Finally, these expression trees are scheduled and code is generated for each expression tree.

Table II. Number of Cycles to Compute Expression Trees Using Right-Left, Left-Right, and *OptSchedule*

		Scheduling Algorithms		
Tree	Origin	Left-first	Right-first	*OptSchedule*
1	real_update	5	5	5
2	complex_update	8	12	8
3	dot_product	8	8	8
4	matrix_1x3	5	5	5
5	matrix	5	7	5
6	iir_one_biquad	14	12	10
7	convolution	6	6	6
8	fir	5	5	5
9	fir2dim	8	12	8
10	lms	12	10	10

6. EXPERIMENTAL RESULTS

DSPstone [Zivojnovic et al. 1994] is a benchmark designed to evaluate the code quality generated by compilers for different DSP processors. *DSPstone* is divided into three benchmark suites: *Application, DSP-kernel, and C-kernel*. The Application benchmark consists of the program *adpcm*, a well-known speech–encoding algorithm. The DSP-kernel benchmark consists of a number of code fragments, which cover the most frequently used DSP algorithms. The C-kernel suite aims to test typical C program statements. The *DSPstone* project was supported by a number of major DSP manufacturers (Analog Devices, AT&T, Motorola, NEC, and Texas Instruments). We used this benchmark for experimental evaluation.

6.1 Expression Trees

We have applied algorithm *OptSchedule* to expression trees extracted from programs in the DSP-kernel benchmark. The metric used to compare the code was the number of cycles that it takes to compute the expression tree.

Observe in Table II that algorithm *OptSchedule* produces the best code when compared with two SC schedules, which is what was expected, since we have proved its optimality. Note that although SC schedules can sometimes produce optimal code, it can also generate bad quality code, as is the case for expression tree 6. We can also verify that the same expression tree generates different code quality when different SC schedules are used. The structure of the expression tree dictates the best SC schedule, and this structure is a function of the way programmers write the code.

6.2 DAG Types Distribution

Expression DAGs are classified in trees, leaf DAGs, and full DAGs. Leaf DAGs are DAGs for which only leaf nodes have outdegree greater than one. We classify a DAG as a full DAG if it is neither a tree nor a leaf DAG. As one can see in Table III, the classification reveals that of all basic blocks analyzed 56% were trees, 38% leaf DAGs, and 6% full DAGs. From the set

Table III. Types of DAGs in Typical Digital Signal Processing Algorithms

DSP kernel	Basic Blocks	Trees	Leaf DAGs	DAGs
real_update	1	1	0	0
complex_update	1	0	0	1
dot_product	1	1	0	0
matrix_1x3	4	3	1	0
matrix	6	4	2	0
iir_one_biquad	1	0	0	1
convolution	2	1	1	0
fir	3	1	2	0
fir2dim	9	6	3	0
lms	4	1	2	1

of benchmarks in Table III we see that the majority of the basic blocks found in these programs are trees and leaf DAGs. Another experiment was performed, this time using the DSPstone application benchmark *adpcm*. As before, basic blocks were analyzed to determine the frequency of trees, leaf DAGs, and DAGs. In this case, 94% of the basic blocks in this program were found to be trees, 3% leaf DAGs, and 3% full DAGs. Although dynamic counting of basic blocks is required in order to provide information on the impact on execution time, one can reasonably argue that a large portion of this program execution time is spent in processing expression trees. Thus, tree-based code generation is very suitable for this application domain.

6.3 Expression DAGs

In Table IV we list a series of expression DAGs extracted from programs in the DSP-kernel benchmark. We selected the largest DAG found in each kernel for comparison with hand-written code. Hand-written assembly code (or *assembly reference code*) for each DSP-kernel program is available from the DSPstone benchmark suite [Zivojnovic et al. 1994].

Compiled code was generated for each DAG, and the resulting number of cycles for a single loop execution is reported in Table IV. Compiled code was also generated using a standard heuristic, which dismantles the DAG by breaking all edges at multiple fanout nodes (column *Standard Heuristic*). Table IV shows the number of processor cycles and the overhead with respect to hand-written code. Note that the overhead is due only to the DAG dismantling technique. The average overhead when comparing the compiled (*Dismantle Heuristic*) and the assembly reference code was 7%. Leaf nodes are treated the same way in both heuristics. They are simply duplicated into different nodes—one for each outgoing edge. As a consequence, both heuristics have the same performance for the case of leaf DAGs. The average overhead (*Dismantle Heuristic*) for the case of full DAGs was higher (11%) than for the case of Leaf DAGs (4%). The discrepancy is due to the existence of memory-register and immediate instructions in the processor ISA, which can have zero cost multiple fanout operands when these are memory references or constant values. Although the heuristic gains may seem very small, every byte matters. Since DSPs have

Table IV. Experiments with DAGs: Leaf DAG (L); Full DAG (F)

DAG Origin	DAG Type	Handwritten Code	Standard Heuristic		*Dismantle* Heuristic	
complex _update	F	16	18	12%	18	12%
matrix _1x3	L	5	5	0%	5	0%
matrix	L	5	5	0%	5	0%
iir _one _biquad	F	15	17	13%	16	7%
convolution	L	5	5	0%	5	0%
fir	L	4	5	20%	5	20%
fir2dim	L	5	5	0%	5	0%
lms	F	7	9	28%	8	14%

restricted on-chip memory size, generation of high quality code is the most important goal for the compiler.

7. CONCLUSION

With increasing demand for wireless and multimedia systems, it is expected that use of DSPs will continue to grow, but research on compiling techniques for DSPs has not received adequate attention. These devices continue to offer new research challenges, which originate in the need for high quality code at low cost and power consumption.

We propose an optimal $O(n)$ instruction selection, register allocation, and instruction scheduling algorithm for expression trees, for a class of heterogeneous register DSP architectures that have acyclic RTGs. We then extend this by proposing heuristics for the case when basic blocks are DAGs. This approach is based on the concept of natural and pseudo-natural edges and seeks to use architectural information to help in the task of dismantling the expression DAG into a forest of trees.

However, the question of how to generate good code for architectures that have cyclic RTGs remains open. As mentioned before, expression trees generated in these architectures can also benefit from the optimality, provided they are free of any allocation deadlock. An interesting question which follows from this is how many expression trees with this property are generated in programs running on these architectures. More work is under way to answer this and other questions.

Acknowledgments

This research was supported in part by the Brazilian Council for Research and Development (CNPq) under contract 204033/87-0 and by the Institute of Computing (UNICAMP), Brazil.

References

AHO, A. V., GANAPATHI, M., AND TJIANG, S. W. K. 1989. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst. 11*, 4 (Oct.), 491–516.

AHO, A. AND JOHNSON, S. 1976. Optimal code generation for expression trees. *J. ACM 23*, 3 (July), 488–501.

AHO, A., JOHNSON, S., AND ULLMAN, J. 1977. Code generation for expressions with common subexpressions. *J. ACM 24*, 1 (Jan.), 146–160.

AHO, A., JOHNSON., S., AND ULLMAN, J. 1977. Code generation for machines with multiregister operations. In *Proc. 4th ACM Symposium on Principles of Programming Languages*. 21–28.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

APPEL, A. W. AND SUPOWIT, K. J. 1987. Generalization of the Sethi-Ullman algorithm for register allocation. *Softw. Pract. Exper. 17*, 6 (June), 417–421.

ARAUJO, G. AND MALIK, S. 1995. Optimal code generation for embedded memory non-homogeneous register architectures. In *Proceedings of the Eighth International Symposium on System Synthesis* (Cannes, France, Sept. 13–15, 1995). ACM Press, New York, NY, 36–41.

ARAUJO, G., MALIK, S., AND LEE, M. 1996. Using register-transfer paths in code generation for heterogeneous memory-register architectures. In *Proceedings of the 33rd Conference on Design Automation*. 591–596.

BRUNO, J. AND SETHI, R. 1975. The generation of optimal code for stack machines. *J. ACM 22*, 3 (July), 382–396.

BRUNO, J. AND SETHI, J. 1976. Code generation for one-register machine. *J. ACM 23*, 3 (July), 502–510.

COFFMAN, E. AND SETHI, R. 1983. Instructions sets for evaluating arithmetic expressions. *J. ACM 30*, 3 (July), 457–478.

FRASER, C., HANSON, D., AND PROEBSTING, T. 1993. Engineering a simple, efficient code generator. *J. ACM 22*, 12 (Mar.), 248–262.

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability*. W. H. Freeman & Co., New York, NY.

HOFFMAN, C. AND O'DONNELL, M. 1992. Pattern matching in trees. *J. ACM 29*, 1 (Jan.), 68–95.

KOLSON, D., NICOLAU, A. N. D., AND KENNEDY, K. 1996. Optimal register assignment to loops for embedded code generation. *ACM Trans. Des. Autom. Electron. Syst. 1*, 2 (Apr.), 251–279.

LANNER, D., CORNERO, M., GOOSENS, G., AND DE MAN, H. 1994. Data routing: a paradigm for efficient data-path synthesis and code generation. In *Proceedings of the High-Level Synthesis Symposium*. 17–22.

LAPSLEY, P., BIER, J., SHOHAM, A., AND LEE, E. A. 1996. *DSP Processor Fundamentals: Architectures and Features*. IEEE Press, Piscataway, NJ.

LEE, E. A. 1988. Programmable DSP architectures: Part I. *IEEE ASSP Mag.*, 4–19.

LEE, E. A. 1989. Programmable DSP architectures: Part II. *IEEE ASSP Mag.*, 4–14.

LIAO, S., DEVADAS, S., KEUTZER, K., AND TJIANG, S. 1995. Instruction selection using binate covering for code size optimization. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design* (ICCAD-95, San Jose, CA, Nov. 5–9, 1995). IEEE Computer Society Press, Los Alamitos, CA, 393–399.

LIEM, C. M. T. AND P., P. 1994. Instruction-set matching and selection for DSP and ASIP code generation. In *Proceedings of the European Design and Test Conference*. 31–37.

MARWEDEL, P. AND GOOSENS, J., Eds. 1995. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Hingham, MA.

MARWEDEL, P. 1993. Tree-based mapping of algorithms to predefined structures. In *Proceedings of the international conference on Computer-aided design* (ICCAD '93). 586–593.

MOTOROLA, 1990. *DSP56000/DSP56001 Digital Signal Processor User's Manual*. Motorola Inc., Phoenix, AZ.

PRABHALA, B. 1980. Efficient computation of expressions with common subexpressions. *J. ACM 27*, 1 (Jan.), 146–163.

SETHI, R. 1975. Complete register allocation problems. *SIAM J. Comput. 4*, 3 (Sept.), 226–248.

SETHI, R. 1970. The generation of optimal code for arithmetic expressions. *J. ACM 17*, 4 (Oct.), 715–728.

Texas Instruments, 1990. *Digital Signal Processing Applications with the TMS320 Family*. Texas Instruments, Austin, TX.

Tjiang, S. 1993. *An Olive Twig*. Synopsys, Inc., Mountain View, CA.

Wess, B. 1990. On the optimal code generation for signal flow computation. In *Proceedings of the International Conference on Proceedings of the International Conference on Circuits and Systems*. 444–447.

Wess, B. 1992. Automatic instruction code generation based on trellis diagrams. In *Proceedings of the International Conference on Proceedings of the International Conference on Circuits and Systems*. 645–648.

Zivojnovic, V., Velarde, J., and Sclåager, C. 1994. *DSPstone, a DSP benchmarking methodology*. Aachen University of Technology, Aachen, Germany.

Received: January 1997; revised: June 1997; accepted: August 1997