Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures

Guido Araujo[1] and Sharad Malik

Department of Electrical Engineering
Princeton University
Princeton, NJ 08540

Abstract

This paper examines the problem of code-generation for expression trees on non-homogeneous register set architectures. It proposes and proves the optimality of an $O(n)$ algorithm for the tasks of instruction selection, register allocation and scheduling on a class of architectures defined as the $[1, \infty]$ Model. Optimality is guaranteed by sufficient conditions derived from the Register Transfer Graph (RTG), a structural representation of the architecture which depends exclusively on the processor Instruction Set Architecture (ISA). Experimental results using the TMS320C25 as the target processor show the efficacy of the approach.

1 Introduction

Non-homogeneous register architectures are frequently encountered in Application Specific Instruction Set Processors (ASIPs). These processors usually have a set of very specialized functional units, and associated registers, that are used to efficiently implement operations with hard performance requirements which frequently occur in the application domain of the processor. Examples of these type of units are: Multiply and Accumulate Units (MAC) and Address Calculation Units (ACU). At the same time, the performance and area constraints on the design of ASIP commonly result in data-paths with restricted connectivity. This specialization is reflected in the Instruction Set Architecture (ISA) design of these processors which typically have very specialized instructions that take operands and store the resulting computation into well defined registers. As a result, limited freedom is available to the code-generation algorithm.

The class of non-homogeneous architectures this paper refers to will be called the $[1, \infty]$ *Model.* This is a subset of the memory-register architectures in which only one or an infinite amount (memory) of a particular storage resource type (from now on called *locations*) is available. A notorious instance of this class is the TMS320C25 processor which will be considered the target architecture for the rest of this paper.

One of the main problems in generating code for programs basic blocks [1] is to select the best combination of processor instructions which can optimally implement the operations in the basic block *Direct Acyclic Graph* (DAG) representation. This task, called *instruction selection*, is equivalent to the DAG-covering problem, which is known to be NP-complete [2]. An approach to ease the problem is to split the DAG into expression trees and to perform tree-covering on each tree separately.

Code-generation for expression trees can be divided into three individual problems: *instruction selection*, *register allocation* and *scheduling*. In this paper we propose an optimal two phase algorithm which performs instruction selection, register allocation and instruction scheduling for an expression tree in polynomial time, under the following constraints: (a) the target processor should fit the $[1, \infty]$ Model; (b) the ISA of the processor must satisfy an (easy!) optimality criterion that we will define later. The first pass of the algorithm, described in Sec.2, performs instruction selection and register allocation simultaneously, using a variation of the Aho-Johnson algorithm [3] that we extend for non-homogeneous architectures using tree-grammar parsing. The second pass, described in Sec.3, is an $O(n)$ algorithm that takes an optimally covered expression tree and schedules instructions such that no memory spills are required. The algorithm we propose for this phase uses the concept of *Register Transfer Graph* (RTG) that we define. We also provide a criterion, based on the RTG that enables the ASIP designer to verify, using exclusively the ISA of the target architecture, if the code resulting from scheduling trees on its ISA has the potential to be optimal.

Section Sec.4 contains results of applying the algorithm to some expression trees extracted from DSP benchmarks. Finally, in Section Sec.5 we summarize our major contributions and provide directions on how this work can be extended.

2 Optimal Instruction Selection and Register Allocation

Algorithms for instruction selection are usually based on pattern matching techniques [1]. Another equally effective approach is to use trellis diagrams [4]. The method we will describe below is based on tree pattern matching and aims to provide easy and

[1] Work partially supported by CNPq (Brazil) under fellowship award 204033/87.0

fast retargetability.

In the context of code generation, tree-covering of an expression tree is a pattern matching task in which instructions of the target processor are represented using IR tree-patterns [1]. Each pattern has an associated cost which reflects the number of execution cycles that the instruction corresponding to the pattern will take to execute. The tree-covering task is the problem of covering the expression tree with a set of patterns such that their total number of execution cycles is minimized.

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 non-homogeneous register architectures usually requires allocating register types for the operands and resulting data. As a consequence, the IR patterns associated with instructions in this kind of processor should carry information regarding the type of register the instruction uses. Furthermore, the strong binding between instructions and the register types they use suggests that instruction selection and register allocation are two tasks that should be performed together.

Instruction	Dest.	Cost	Pattern
add m	a	1	PLUS(a,m)
apac	a	1	PLUS(a,p)
spac	a	1	MINUS(a,p)
mpy m	p	1	MUL(t,m)
mpky k	p	1	MUL(t,CONST)
lack k	a	1	CONST
pac	a	1	p
sacl m	m	1	a
lac m	a	1	m
lt m	t	1	m

Table 1: Partial ISA of the TMS320C25 processor

Consider, for example, the IR patterns in Tab.1 corresponding to a subset of the instructions in the TMS320C25 ISA. In Tab.1 each instruction has associated with it a tree-pattern whose nodes are composed of operations (PLUS,MINUS,MUL), registers (a,p,t), constants (CONST) and memory reference (m). Notice that each instruction implicitly defines the registers it uses. For example, the instruction *apac* can only take its operands from registers a and p, and always computes the result back into a. Transference instructions like *pac* (transfer data from p to a) can also be represented this way by using only the source register as the IR pattern.

2.1 Problem Definition

The problem we address here is the problem of determining the best cover of the 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 are to the location where the instruction requires them to be.

2.2 Problem Solution

Tree-grammar parsers have been used as a way to implement code-generators [5][6]. They combine dynamic programming and efficient tree-pattern matching algorithms for optimal instruction selection. Tools for automatic generation of code-generators based on tree-grammars parsing [5][6][7] are an effective way to provide fast retargetability. This is extremely desirable for the design process that involves ASIPs, since typically several possible architectures will be evaluated before one is selected. Based on these observations we have implemented our instruction selection/register allocation algorithm using Olive [7], a simple, fast and easy-to-use code-generator generator. Olive takes as input a set of grammar rules, where tree patterns are described in a prefixed linearized form, similar to the notation used in Tab.1. Similarly as grammar description for programming languages [1], tree-grammars are formed by terminals and non-terminals, which are called grammar symbols. Register allocation is specified by assigning one grammar non-terminal for each location in the architecture where data can be stored. When this is done the Olive rules assume the following format:

$location : pattern \, \{cost\} = \{action\};$

where:

1. *location* is a non-terminal node representing a storage resource where the instruction result should be stored;
2. *pattern* is an IR tree corresponding to a processor instruction, containing grammar terminals and non-terminals;
3. *cost* is a piece of code that computes the cost of matching a subtree of the expression tree using *pattern*;
4. *action* is another piece of code used to emit the instruction corresponding to *pattern*.

Consider, for example, the Olive description for the TMS320C25 architecture (Fig.1) obtained from the IR patterns of Tab.1. Notice that non-terminals are represented by lowercase letters and terminals by capital letters. Rules 1 to 3 and 4 to 5 correspond

```
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) mpky 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
```

Figure 1: Partial Olive specification of the TMS320C25 processor (instruction numbers and names on the right are not part of the specification)

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 transference instructions and play an important role in the implementation of the proposed algorithm since they are responsible for bringing the cost of moving data through the data path into the total cost of a match. We should point out that, for sake of simplicity, we did not represent in Fig.1 all patterns corresponding to commutative operations, although we will assume their existence whenever required.

If we do not consider instruction scheduling and its associated spills at this point, then the algorithm proposed above is optimal (refer to [3] for proof).

3 Scheduling

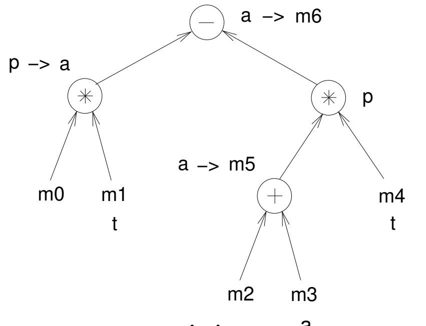
Optimal instruction selection and register allocation for an expression tree is not enough to produce optimal code. For optimal code the instructions must be scheduled in such a way that no additional memory spills are introduced. Optimal algorithms exist for the scheduling problem for the class of homogeneous register architectures [8][3].

The Sethi-Ullman [8] algorithm optimaly schedules instructions using a two pass approach. The resulting code has the least possible number of instructions (corresponding to the fewest number of memory spills) and registers.

Aho-Johnson [3] showed that, by using dynamic programming, optimal code can be generated in linear time for a wide class of homogeneous register set architectures. The schedule they proposed is based on their *Strong Normal Form Theorem*. One code sequence is in SNF if it is formed by a set of code sub-sequences separated by memory storages, where each code sub-sequence is determined by a *Strongly Contiguous* schedule (SC-schedule). One code sequence is a SC-schedule if it was formed as follows: at every selected match m, with children subtrees T_1 and T_2, continuously schedule the instructions corresponding to subtree T_1 followed by the instructions corresponding to T_2 and m. Although still used, SC-schedules are not an effective approach for code generation on non-homogeneous architectures.

3.1 Problem Definition

The inefficacy of SC-schedules on non-homogeneous register architectures derives from the fact that on these architectures the final code sequence is extremely dependent on the order that subtrees are evaluated. Consider for example the IR tree of Fig.2(a). The expression tree was matched using the algorithm proposed in Sec.2 and the TMS320C25 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 are shown in Fig.2. Memory position m_7 was used whenever a spilling location was required by the scheduler. For the code of Fig.2(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



(a)

lt	m1	lt	m4	lac	m3
mpy	m0	lac	m3	add	m2
pac		add	m2	sacl	m5
sacl	m7	sacl	m5	lt	m1
lac	m3	mpy	m5	mpy	m0
add	m2	lt	m1	pac	
sacl	m5	pac		lt	m4
lt	m4	sacl	m7	mpy	m5
mpy	m5	mpy	m0	spac	
lac	m7	pac		sacl	m6
spac		lt	m7		
sacl	m6	mpky	1		
		spac			
		sacl	m6		
(b)		(c)		(d)	

Figure 2: (a) Matched IR tree for the TMS320C25; (b) SC Left-first schedule; (c) SC Right-first schedule; (d) Optimal schedule

code of Fig.2(c). Notice that neither the SC-schedules in Fig.2(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 Fig.2(d). The question raised by the above example is if there exists a guaranteed schedule such that no spilling is required. We will prove this schedule exist, under certain conditions that depend exclusively on the ISA of the target processor, and that according Aho-Johnson [3] it is a *SNF* schedule.

3.2 Problem Solution

In this section we define the concept of RTG and show how it can be used to derive sufficient conditions for optimal code generation for $[1,\infty]$ architectures. Using this result we propose a linear time algorithm for optimal SNF code scheduling of expression trees on these architectures and prove its optimality.

Let T be an expression tree with unary and binary operations. Let $L : T \rightarrow R \cup M$ be a function which 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, v_1 and v_2 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 Fig.3(a). The following results are valid for any expression tree for which at most two operands of each pattern are from storage locations (the other operands can be immediate values or indexing registers).

Definition 1 (Allocation Deadlock) *We say that an expression tree contains an allocation deadlock if the following conditions are true: (a) $L(v_1) \neq L(v_2)$ and (b) there exist nodes w_1 and w_2, $w_1 \in T_1$ and $w_2 \in T_2$ such that $L(w_1) = L(v_2)$ and $L(w_2) = L(v_1)$.* □

The above definition of allocation deadlock can be visualized in Fig.3(a). As one can see, it is the situation when two sibling subtrees T_1 and T_2 contain each 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.

Lemma 1 *Let u be any node in expression tree T and T_u the subtree rooted at u. If T does not have a spill free schedule then it contains at least one node u for which T_u has an allocation deadlock.*

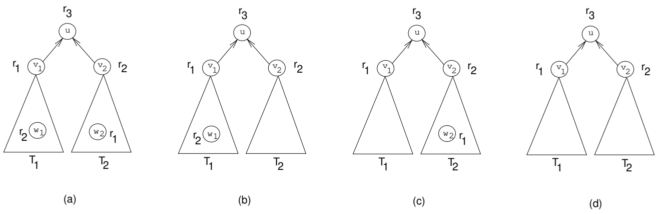


Figure 3: (a) Tree with allocation deadlock; (b) – (d) Trees without allocation deadlock

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) $L(v_1) = L(v_2)$. This case will not happen since no non-unary operator of an expression tree takes its two operands simultaneously from the same location, unless this location is in memory. In this case any SC-schedule can be used to schedule T_u.

(b) $L(v_1) \neq L(v_2)$, $L(w_1) = L(v_2)$, but no node w_2 exist 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 (Fig.3(b)).

(c) $L(v_1) \neq L(v_2)$, $L(w_2) = L(v_1)$, but no node w_1 exist for which $L(w_1) = L(v_2)$. This is symmetric to the previous case (see Fig.3(c)).

(d) $L(v_1) \neq L(v_2)$, but no nodes w_1 and w_2 exist. This case is trivial, any SC-schedule results in a valid schedule (Fig.3(d)).

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

Corollary 1 (Free Schedule) *If for all nodes u in T, T_u is free of allocation deadlocks, then T has an optimal schedule which does not require any memory spilling. Moreover this schedule can be determined by recursively proceeding as follows: for each node u in T schedule first the child v of u for which $L(v) \neq L(t)$, where t is any node contained in the subtree rooted at the sibling of v.*

Proof. Directly from the lemma above. □

Definition 2 (RTG) *The RTG is a directed labeled graph where each node represents a location in the data-path 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 into location r_j.* □

The nodes in the RTG are divided into three types: single register (or simply register) nodes, register files and memories nodes. Memory nodes (M) and register file nodes represent a set of locations of the same type which can store multiple operands. In the RTG they are distinguished from register nodes by means of a double circle. Notice that the RTG is a labeled graph where each edge has labels corresponding to the instructions that require that operation.

Definition 3 (RTG Criterion) *Let r_1, r_2 and r_3 be nodes in a RTG such that: (a) r_3 has incoming edges from register nodes r_1 and r_2, and these edges have one common label l; (b) there exists at least one directed cycle between nodes r_1 and r_2. We say that the RTG Criterion is satisfied if there exists one memory node on each cycle between r_1 and r_2 (e.g. Fig.4(a)).* □

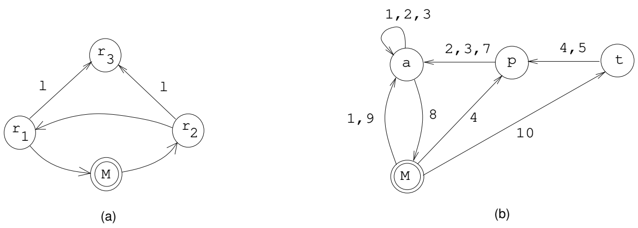


Figure 4: (a) RTG that satisfies the RTG Criterion; (b) TMS320C25 architecture satisfies the RTG criterion

Example 1 Consider, for example, the partial OLIVE description in Fig.1 for the ISA of the TMS320C25 processor. The RTG of Fig.4(b) was formed from that description. The numbers in parenthesis on the right side of Fig.1 are used to label each edge of the graph. Notice that only registers a and p are destinations of instructions which take two other locations as operands. Since all cycles between these locations contain M we can say that the TMS320C25 architecture satisfies the RTG Criterion. □

Theorem 1 (RTG Theorem) *If an ISA satisfies the RTG Criterion then for any expression tree there exists a schedule that is free of memory spills.*

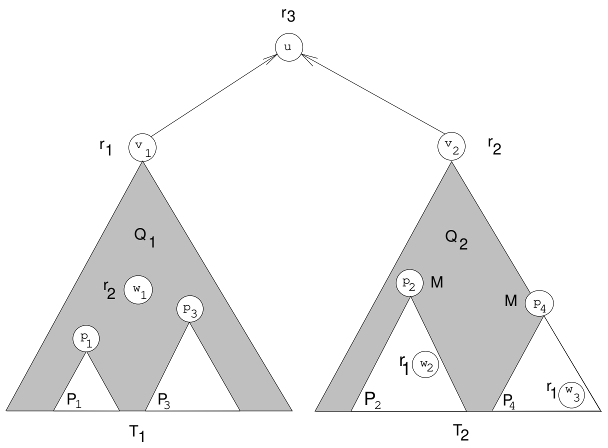


Figure 5: The RTG Theorem

Proof. Let T be an expression tree rooted at u, and v_1 and v_2 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 Fig.5) as the subtrees formed in T_i after removing all nodes from subtrees P_k. We will show that if the RTG Criterion is satisfied an optimal schedule can always be determined by properly ordering the schedules for $P_1 - P_4$ and $Q_1 - Q_2$. Here we have to address two cases: (a) If r_3 corresponds to a RTG node for which no allocation deadlock can possibly occur (conditions (a) or (b) from Definition 3 not satisfied) then T can be scheduled using the schedule determined in Corollary 1; (b) Now consider that r_3 corresponds to a RTG node for which an allocation deadlock is possible, as in Fig.4(a). One can see from there and Fig.5 that if the RTG Criterion is satisfied then for each node in T_2 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 M, i.e. p_2. Notice that one can recursively schedule subtrees P_2 and P_4 in T_2 for which the root was allocated to memory, what 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. Therefore the tree $T_1 \cup Q_2$ can now be scheduled using Corollary 1 and no spill will be required.

Notice that the same result can be obtained if one first recursively schedule all subtrees $P_1 - P_4$ (white areas in Fig.5) followed by applying Corollary 1 to schedule subtree $Q_1 \cup Q_2 \cup \{u\}$. □

3.2.1 Optimal Scheduling Algorithm

In the following we present *OptSchedule* (Fig.6) a two pass $O(n)$ algorithm based on the proof of Theorem 1. The algorithm , takes an optimaly allocated expression tree resulting from the approach described in Sec.2 and produces spill free code.

In its first pass, represented by procedure *GetUsage* (Fig.6) the algorithm does a preorder traversal of the tree. At each node u two sets are computed: $memset(u)$ and $regset(u)$. Set $memset(u)$ contains pointers to those nodes inside T_1 and T_2 which were allocated to M and that have no other node allocated to M on its path to u (e.g. node is p_2 in Fig.5). Set $regset(u)$ keeps the names of registers which were allocated to nodes into subtrees Q_1 and Q_2.

```
GetUsage (u)
begin
    memset(u) = φ;
    regset(u) = φ;
    if match(u) is not memory
        regset(u) = match(u);
    foreach v in children(u)
        GetUsage(v);
        if match(v) is memory
            memset(u) = memset(u) ⋃ {v};
        else
            memset(u) = memset(u) ⋃ memset(v);
            regset(u) = regset(u) ⋃ regset(v);
        endif;
    endfor;
end

OptSchedule (u)
begin
    foreach p in memset(u)
        OptSchedule(p);
    foreach v in children(u)
        FreeSchedule(v);
    emit(u);
end

FreeSchedule (u)
begin
    if match(u) is memory
        return;
    if u is not a leaf
        v1 = unique(children(u));
        foreach w in children(v1)
            FreeSchedule(w);
    endif;
    emit(u);
end
```

Figure 6: First pass: GetUsage; Second pass: OptSchedule and FreeSchedule

In the second pass *OptSchedule* executes a series of two tasks. First, for all $p \in memset(u)$ *OptSchedule* recursively schedules all subtrees rooted at p. These

Tree	Origin	Scheduling Algorithms		
		LR	RL	OS
1	real_updates	5	5	5
2	complex_update	8	12	8
3	iir_one_biquad	8	12	8
4	fir	5	5	5
5	filt4	5	5	5
6	flatv	6	6	6
7	calcp	10	8	8
8	decorr	5	5	5
9	lag	15	25	13
10	interp	12	9	9

Table 2: Number of cycles to compute expression trees using: Right-Left (RL), Left-Right (LR) and *OptSchedule* (OS)

subtrees are P_k $(k = 1, 2, \ldots)$ and correspond to the blank areas inside T_1 and T_2 in Fig.5. Once this is finished, different memory positions are live but not conflicting, and subtrees T_1 and T_2 are reduced to Q_1 and Q_2 respectively. Second, procedure *FreeSchedule* (Fig.6) (the implementation of the proof of Corollary 1) traverses the subject tree in preorder. At each node it uses function *unique* to determine the child of u, say v_1, whose $match(v_1)$ is a register that is not present in the *regset* of the other child of u. The existence of v_1 is guaranteed, as it was shown in Theorem 1. Once v_1 is determined, the algorithm schedules its subtree Q_1 followed by Q_2 and u.

Theorem 2 *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 1. Also from Theorem 1 one can see that the algorithm touches every node in T only once. Hence the algorithm running time is $O(n)$. □

4 Results

The proposed approach was applied to some expression trees which were extracted from the IR form of a set of programs. In Tab.2 trees from 1 to 4 can be found in DSP-kernel programs of the DSPstone benchmark [9]. Trees from 5 to 10 were extracted from a set of programs used to implement signal processing and speech encoding/decoding in a cellular telephone unit.

The metric used to compare the code was the number of cycles that takes to compute the expression tree in the target processor. This is possible since we are not considering *Instruction Level Parallelism* (ILP) here. From Tab.2 one can see that algorithm *OptSchedule* (OS) produces the best code when compared with two SC-schedules, what was expected since we have proved its optimality. Notice that although SC-schedules can occasionally produce optimal code, it can also generate bad quality code as it is the case for expression tree 9. Also notice that numbers in Tab.2 do not take into consideration the cost of the instructions required to compute the address of variables in memory. Minimizing this cost is a problem known as *offset assignment*, which can be efficiently solved using the technique proposed in [10].

5 Conclusion and Future Work

We have proposed an optimal instruction selection, register allocation and instruction scheduling algorithm for a class of non-homogeneous architectures that satisfy the $[1, \infty]$ Model and the RTG Criterion that we define. We have shown that the RTG is a model that can be effectively used to improve the understanding of the interaction between the ISA design and the code-generation task. The target architecture was described using OLIVE, an efficient and easily retargetable code-generator generator. Currently we have been working on the a generalization of this approach to a broad architecture model, the $[N(M)]$ Model, where N classes of registers with M registers are available.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison Wesley, Boston, 1988.

[2] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[3] A.V. Aho and S.C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[4] B. Wess. Automatic instruction code generation based on trellis diagrams. In *Proc. Int. Conf. Circuits and Systems*, volume 2, pages 645–648, 1992.

[5] A.V. Aho, M. Ganapathi, and S.W.K Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Prog. Lang. and Systems*, 11(4):491–516, October 1989.

[6] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a simple, efficient code generator. *Journal of the ACM*, 22(12):248–262, March 1993.

[7] Tjiang S.W.K. An olive twig. Technical report, Synopsys Inc., 1993.

[8] R. Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[9] V. Zivojnovic, J.M. Velarde, and C. Sclåger. DSPstone, a DSP benchmarking methodology. Technical report, Aachen University of Thecnology, August 1994.

[10] Liao S.Y., Devadas S., Keutzer K., Tjiang S., and Wang A. Storage assignment to decrease code size. Accepted for publication in 1995 ACM Conference on Programming Language Design and Implementation.