Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures

Guido Araujo[1] and Sharad Malik

Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
{guido,malik}@ee.princeton.edu

Mike Tien-Chien Lee

Fujitsu Laboratories of America
San Jose, CA 95134
lee@fla.fujitsu.com

Abstract

In this paper we address the problem of code generation for basic blocks in heterogeneous memory-register DSP processors. We propose a new a technique, based on register-transfer paths, that can be used for efficiently dismantling basic block DAGs (Directed Acyclic Graphs) into expression trees. This approach builds on recent results which report optimal code generation algorithm for expression trees for these architectures. This technique has been implemented and experimentally validated for the TMS320C25, a popular fixed point DSP processor. The results show that good code quality can be obtained using the proposed technique. An analysis of the type of DAGs found in the DSPstone benchmark programs reveals that the majority of basic blocks in this benchmark set are expression trees and leaf DAGs. This leads to our claim that tree based algorithms, like the one described in this paper, should be the technique of choice for basic block code generation with heterogeneous memory-register architectures.

1 Introduction

Digital Signal Processors (DSPs) are heterogeneous register set architectures designed to meet performance and area constraints imposed by data-intensive applications found in signal processing and communication domains. In order to achieve this goal, designers choose to implement a set of highly specialized functional units that execute operations frequently required in these algorithms. Examples of these type of units are: Multiply and Accumulate Units (MAC) and Address Calculation Units (ACU). It is also common for DSP designers to use dedicated interconnections between a few specialized registers and the available functional units to reduce the cost of the design. This specialization of functional units, registers, and interconnection is reflected in the the processor Instruction Set Architecture (ISA). Frequently the ISA of DSPs, and *Application Specific Instruction Set Processor* (ASIPs), have dedicated instructions which take operands and store results into specific registers as opposed to using generic register files.

This paper addresses the problem of generating code for basic blocks in heterogeneous register set architectures. Similar effective approaches for this problem have been proposed in [1][2]. In general they combine pattern matching algorithms with data-routing [3] and DAG or tree scheduling techniques. Approaches exclusively based on trees, like the one explored here, have not received much attention though.

This paper is divided into four sections as follows. Sec.2 describes the class of target architectures we are considering, its machine representation and the *Register Transfer Graph* (RTG). The RTG is a structural representation of the datapath, which is used subsequently in this paper. Sec.3 contains a formal definition of the problem and an example illustrating its relevance. Sec.4 shows how register-transfer paths can be used to dismantle expression DAGs into trees. The results of applying such an approach on typical digital signal processing algorithms is presented in Sec.5. Finally in Sec.6 we list some important conclusions.

2 The Architectural Model

This paper proposes an approach for the code generation problem for basic blocks on a class of memory-register DSPs architectures. Of the DSP architectures which can be identified in this class we will consider the TMS320C25 processor [4] as the representative target architecture for the rest of this paper.

In this section we describe the machine representation used throughout this work. Initially, machine instructions are written in functional notation where operation patterns are used to describe instructions of the processor ISA. This description is particularly useful for code generation based on tree-grammar parsing as in [5].

Example 1 Consider, for example, the partial functional description of the TMS320C25 ISA in Fig.1. The symbol on the left side of the colon represents the location in the datapath which will store the result of the operation described on the right. On the far right the numbers in parenthesis represent instruction identifiers and after them are the mnemonics used to describe the instruction which implements that operation. Storage

[1] Work partially supported by CNPq and Dept. of Computer Science, UNICAMP (Brazil)

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA ©1996 ACM, Inc. 0-89791-833-9/96/0006..$3.50

```
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
```

Figure 1: Partial description of the TMS320C25 processor ISA

locations in the datapath are represented by lowercase letters (a, p, t, m), where a is an accumulator, p the product register, m memory and t is used to store one of the operands of the multiplier. Capital letters are used to specify operations (PLUS, MINUS, MUL) or constants (CONST).

This paper uses a representation of the architecture ISA, known as the *Register Transfer Graph* (RTG) [6]. The RTG is a structural representation of the datapath topology which contains information about the instructions in the processor ISA. The nodes in the RTG describe locations in the datapath, such as registers, register files and memories. Memories and register file nodes are represented by double circles in order to distinguish them from single register nodes. An edge between nodes r_1 and r_2 in the RTG defines a path between these locations in the datapath. Labels on this edge correspond to the identifiers of the instructions in the ISA which take one operand from the location r_1 and store the result into location r_2.

Example 2 The RTG for the TMS320C25 architecture in Fig.2 can be easily derived from the partial ISA description of Fig.1. The instruction identifiers in Fig.1 were used to label each edge of the RTG. For example, from Fig.2 one can see that accumulator a has two incoming edges numbered (1). One edge originating at node a (self-loop edge) and another at node m, together describe the transfer operations required by instruction PLUS(a,m).

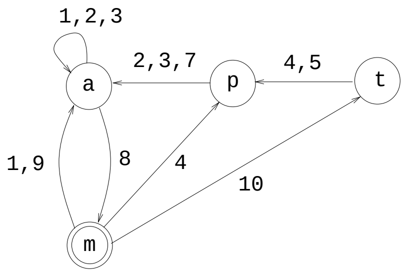


Figure 2: TMS320C25 RTG representation

One of the major problems in code generation is to determine schedules which minimize resource conflicts that occur when one instruction needs to use a datapath register that contains a valid data. Traditionally one deals with this problem by saving the contents of the register into another location, usually memory, an operation know as *memory spilling*. It has been shown [6] that architectures which belong to the memory-register class can generate expression trees which have spilling free schedules, provided these architectures satisfy certain criterion based exclusively on their RTG. The criterion is simple and can be stated as follows:

Definition 1 (RTG criterion) *An architecture satisfies the RTG criterion if there exists one memory node on each cycle (not a self-loop) of the architecture RTG (Fig.3)* □

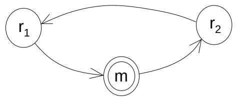


Figure 3: RTG criterion satisfied for a pair of registers r_1 and r_2

It is easy to check from Fig.2 that the TMS320C25 architecture satisfies the RTG criterion. Observe that any cycle between any pair of nodes in Fig.2 will have m on it.

In the following sections we will show how the RTG criterion can also be used in the task of dismanteling the expression DAG.

3 Problem Definition

One of the first tasks in generating code for an expression DAG is to select the best set of instructions which perform the DAG operations. This task, known as instructions selection, subsumes the problem of DAG covering which is known to be NP-complete [7]. In practical solutions to this problem, heuristics have been proposed which divide the DAG into its component trees by selecting an appropriate set of trees. However, this dismantling of the DAG into component trees is not unique and there are several ways in which this can be done. Traditionally, the heuristic employed in the case of homogeneous register architectures is to disconnect multiple fanout nodes of the DAG.

In order to divide a DAG into its component trees one has to be able to disconnect, or to break, edges in the DAG. For the code generation task, breaking a DAG edge between nodes u and v (Fig.4(a)) requires the allocation of a temporary storage (say m_1) to save the result of operation u while this is not consumed by operation v. This storage location is traditionally the memory but it can, in general, be any place in the datapath. Breaking the edge of an expression DAG also requires that a constraint RAW (*R̲ead A̲fter W̲rite*) edge be introduced between the two m_1 nodes in order to guarantee that the original ordering of the operations will be maintained by the scheduler. This edge is represented by a dashed line in Fig.4(a). For the sake of simplicity, marking one edge for breaking can be represented by a small line segment transverse to the subject edge, as it is shown in Fig.4(b).

Heuristics for DAG code generation based on trees in heterogeneous architectures have not received much attention. This may largely have to do with the fact that, until recently, optimality could not be guaranteed for trees on these architectures, and thus breaking DAGs into trees was considered to probably generate extremely inefficient code. Only recently, optimal code generation algorithms for expression trees have been proposed in [6].

The key idea in this paper is a heuristic which uses architectural information from the RTG in the selection of

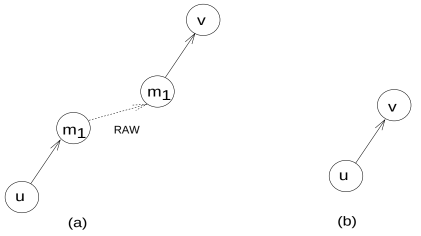
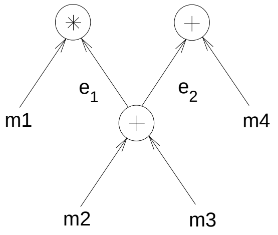


Figure 4: (a) Breaking the edge; (b) Simplified notation

component trees of a DAG, such that the resulting code quality is improved. Consider for example the DAG of Fig.5. Notice that three different approaches can be used to decompose this DAG into its component trees, depending which edge (e_1 or e_2) is selected to break. As one can see in Fig.5(b) one extra instruction corresponding to a 17% overhead is generated when the dismantling heuristic is based on breaking edge e_2 instead of e_1. Coincidentally the code in Fig.5(a) is also the best sequential code one can generate from the subject DAG. Observe from the TMS320C25 architectural description in Fig.1, that the multiplication operation requests its operands in memory (m) and t and that the result of the addition operation always produce its result into the accumulator a. Notice also from Fig.2 that to bring any data in a to register t one has to go through m. By carefully analyzing Fig.5 one can see that the result of the addition operation $m_2 + m_3$ has to be stored into a and must be moved to m or t in order to be used as an operand of the multiplication operation. But to move data from a to t one has to go though memory (m). Suppose the memory position selected to store this temporary result is m_5. Hence, by breaking DAG edge e_1 one is just assigning in advance a memory node which will appear on that edge, during the instruction selection phase of the code generation. Notice that the existence of a register-transfer path which 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.



```
lac    m2
add    m3
sacl   m5
lt     m1
mpy    m5
add    m4
```

(a)

```
lac    m2
add    m3
sacl   m5
lt     m1
mpy    m5
lac    m5
add    m4
```

(b)

Figure 5: (a) Breaking edge e_1; (b) Breaking edge e_2

Notice also that when edge e_2 is broken, pattern PLUS(a,m) (instruction *add* m_4) can not 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 at the bottom of the code of Fig.5(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. The question one can ask at this point is if the above observations can be generalized. We will see in the next sections that this is indeed possible.

4 Problem Solution

The heuristic we propose to address the problem just described is divided into four phases. In the first phase (Sec.4.1) partial register allocation is done for those datapath operations which can be clearly allocated before any code generation task is performed in the DAG. During the second phase (Sec.4.2), architectural information is employed to identify special edges in the DAG which can be broken without introducing any loss of optimality for the subsequent tree mapping stages. In the third phase (Sec.4.3) edges are marked and disconnected from the DAG. Finally component trees are scheduled and optimal code generated for each component tree.

4.1 Partial Register Allocation

A general property of heterogeneous register architectures is that the result of specific operations are always stored in well defined datapath locations. Take for example operations *add* and *mul* in the target processor TMS320C25. Notice that they implicitly define the primary storage resources that are used for the operation result. In the case of the TMS320C25 (Fig.1), no register allocation task is required to determine that registers a and p are respectively used to store the immediate result of operations *add* and *mul*. Thus, partial allocation of the registers used to store the result of operations can be performed well in advance, even before the task of breaking the edges of the expression DAG takes place.

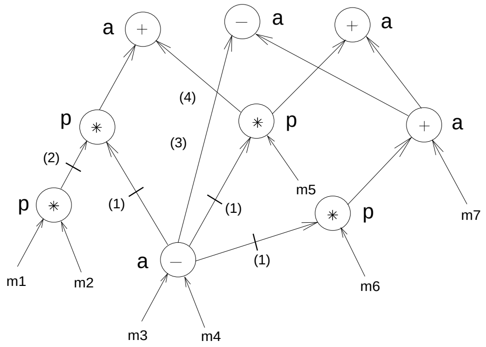


Figure 6: Expression DAG after partial register allocation was performed and natural edges identified

4.2 Natural Edges

We saw from Sec.3 that some edges have specific properties originating from the target architecture, which allow us to disconnect them from the DAG without compromising the optimality of the subsequent code generation for trees. These edges, termed *natural edges*, are defined as follows.

Definition 2 (Natural Edges) *If the instruction selection matching of edge (u, v) always produces a sequence of data transfer operations in the datapath which pass through memory, edge (u, v) is referred to as a natural edge.*

Now given an expression DAG D, and a target architecture which satisfies the RTG criterion, it can be shown that a number of edge are natural edges. In order to do that let us state a set of simple lemmas.

Let r_1 and r_2 be a pair of registers in the datapath of a memory-register architecture which satisfies the RTG criterion according to Fig.3. Also let $L : D \rightarrow R \cup M$, be a function which 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.

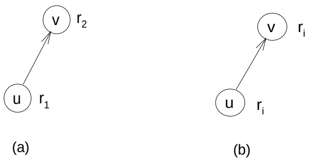


Figure 7: (a) (u, v) is natural; (b) (u, v) is natural if r_i has no self-loop in RTG

Lemma 1 **($r_1 \rightarrow r_2$)** *Every edge (u, v) in D for which $L(u) = r_1$ and $L(v) = r_2$ is a natural edge.*

Proof. Observe in Fig.3 that a path from registers r_1 to r_2 will be traversed whenever instruction selection is performed on edge (u, v). Thus a memory operation will always be selected during instruction selection on (u, v) and therefore (u, v) is a natural edge (Fig.7(a)).

Lemma 2 **($r_i \rightarrow r_i$)** *Edges (u, v) for which $L(u) = L(v) = r_i$, $i = 1, 2$ are natural edges only if no self-loop exists on register node r_i in the RTG representation of the target architecture (Fig.7(b)).*

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

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

Example 5 Consider each one of the lemmas above and the RTG of TMS320C25 in Fig.2.

(1) From Lemma 1 one 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 natural edge.

(2) Consider now Lemma 2. First take the situation when $r_i = p$. By looking at the RTG of Fig.2 one can see that register p has no self-loop. Since the TMS320C25 satisfies the RTG criterion then any DAG edge (u, v) such that $L(u) = L(v) = p$ is a natural edge. Now consider the case when $r_i = a$. From the RTG one can see that register a contains a self-loop and thus nothing can be said regarding these edges.

In the following two lemmas we show that DAG edges can sometimes interact such that natural edges will defined. This will introduce us to the concept of *pseudo-natural edges*.

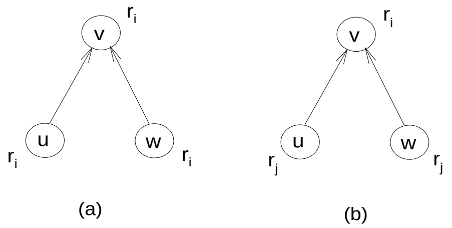


Figure 8: (a) One of the edges is always natural; (b) Edge (w, v) is a natural edge

Lemma 3 **(r_i op $r_i \rightarrow r_i$)** *Consider operation v and its operand nodes u and w in Fig.8(a). If partial register allocation of these operations is such that $L(u) = L(v) = L(w) = r_i$, $i = 1, \ldots |R|$ then one of the edges (u, v) or (w, v) is a natural edge.*

Proof. Notice that no binary operation v can take both its operands simultaneously from the same register. We have to consider here 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 which 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 which is contained in an RTG loop (not a self-loop) that goes from r_i back to r_i. Similarly as in Lemma 2 matching (w, v) will introduce a sequence of transfer operations which necessarily goes through a memory node in the RTG, making (w, v) a natural edge.

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

□

Lemma 4 **(r_j op $r_j \rightarrow r_i$)** *Consider operation v and its operand nodes u and w of Fig.8(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 a memory node, then one (u, v) or (w, v) is a natural edge.*

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 and therefore the corresponding edge is a natural edge. □

From these lemmas, we see that we need to decide which edge between (u, v) and (w, v) is to be disconnected from the DAG. Different instruction selection costs might result depending on which edge is selected. In this case we call the corresponding natural edges *pseudo-natural edges* to distinguish them from natural edges. Pseudo-natural edges are identified using a double line segment to distinguish them 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 6 Consider Lemmas 3 and 4 above and the RTG of Fig.2:

(3) Lemma 3 is satisfied for the case when $r_i = a$ 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 which does not go through a memory node. Therefore one of the edges is a pseudo-natural edge.

After rules 1–4 of Examples 5 and 6 are applied, the expression DAG of Fig.6 results. Each marked edge in Fig.6 has on its side a number corresponding to a rule in Examples 5 and 6.

4.3 Dismantling Algorithm

The task of dismanteling an expression DAG may potentially introduce cyclic RAW dependencies between the resulting tree components leading to an impossible schedule. Consider, for example, the reconvergent paths from nodes u to v and the component trees T_1 and T_2 of Fig.9(a). Dismantling the DAG of Fig.9(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 break. In this case nodes u, v and tree T_1 can be collapsed into a single component tree T_3, breaking the DAG into trees T_3 and T_4. As we have mentioned before if an edge between two nodes is broken then a RAW edge should be introduced between them. 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. Notice that dismanteling is also possible if edge (T_2, w) is broken instead of (T_2, v) (Fig.9(b)). When this occur 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 in [6] can not be guaranteed anymore, since now it has to satisfy the constraint imposed by the new RAW edge inside T_3. From the two situations analyzed above, we can conclude that edges on both reconvergent paths have to be disconnected in order to guarantee proper scheduling of operations inside component trees and between component trees.

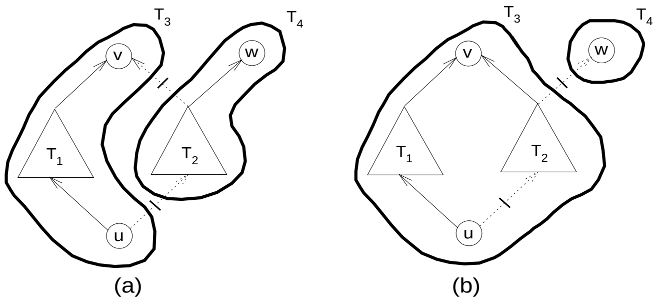


Figure 9: (a) Cyclic RAW dependency; (b) Constraining the tree scheduler

An algorithm which dismantles the DAG should disconnect edges by using as much as 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 these edges adds no cost to the total cost of the final code. After that *Dismantle* proceeds indentifying 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 which starts the reconvergent path at the corresponding multiple fanout node is broken. At this point all reconvergent paths in the expression DAG have been disconnected. Finally, additional edges are broken such that no node ends up with more than one outgoing edge. Broken edges are then disconnected from the DAG, temporary memory nodes created and RAW edges introduced between component trees. After algorithm *Dismantle* is executed, the DAG is decomposed into its component trees T_i as in Fig.10. Each component tree T_i is represented by a circle containing DAG edges and nodes. Broken edges which are not natural or pseudo-natural edges are identified by a dark circle mark. Topological ordering is then performed in order to schedule the component trees. At this point optimal code is generated for each component tree individually using the technique proposed in [6].

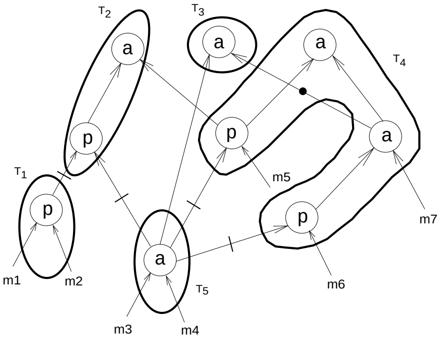


Figure 10: Resulting component trees after dismantling

5 Results

In Tab.1 we list a series of 13 expression DAGs extracted from typical digital signal processing programs. These are profiled sections of kernel programs from the DSPstone benchmark suite [8]. We have selected the largest DAG found on each kernel for the purpose of comparison with hand-written code. Hand-written assembly and compiled code were generated for each DAG and the resulting number of cycles for a single loop execution reported in Tab.1. Compiled code was also generated using a standard heuristic which dismantles the DAG by breaking all edges at multiple fanout nodes (column *Standard Heuristic*). The costs reflect the number of processor cycles and the overhead with respect to hand-written code. Notice that the overhead is due only

DAG	DAG	Hand-	Standard		Our		Basic	Trees	Leaf	Full
Origin	Type	written	Heuristic		Approach		Blocks		DAGs	DAGs
		#Instr.	#Instr.	Overhead	#Instr.	Overhead				
real_update	T	5	5	0%	5	0%	1	1	0	0
complex_update	L	16	18	12%	18	12%	1	0	1	0
dot_product	L	5	5	0%	5	0%	1	1	0	0
matrix_1x3	T	5	5	0%	5	0%	4	3	1	0
matrix	T	5	5	0%	5	0%	6	4	2	0
iir_one_biquad	F	15	17	13%	16	7%	1	0	0	1
convolution	L	5	5	0%	5	0%	2	1	1	0
fir	L	4	5	20%	5	20%	3	1	2	0
fir2dim	T	5	5	0%	5	0%	9	6	3	0
lms	F	7	9	28%	8	14%	4	1	2	1

Table 1: Experiments with DAGs (T = Tree; L = Leaf DAG; F = Full DAG)

to the DAG dismantling technique. DAGs were classified in trees, leaf DAGs and full DAGs. Leaf DAGs are DAGs for which only leaf nodes have multiple fanout edges. We classify a DAG as a full DAG if it is not a tree nor a leaf DAG. As one can see our approach performs better than the standard heuristic for the cases which are full DAGs. The average overhead when comparing the compiled and the assembly reference code was 6%. Although the overhead for DAGs is still high, the existence of a large number of low overhead trees compensates that. Leaf nodes are treaded 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. Notice that the average overhead for the case of full DAGs was higher (11%) than for the case of leaf DAGs (8%). The discrepancy is certainly 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. An analysis of the source programs from which DAGs in Tab.1 were extracted showed indeed that a large number of common expressions are simply program variables and constants.

We also performed an experiment on the DSPstone kernel benchmark in order to determine the type of DAGs found on basic blocks of typical digital signal processing programs. As one can see from Tab.1, the results revealed that of all 32 basic blocks analyzed 56% were trees, 38% leaf DAGs and only 6% full DAGs. Another experiment was performed, this time using the DSPstone application benchmark *adpcm*, a well known speech encoding algorithm. 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 are trees, 3% leaf DAGs and 3% full DAGs. If *adpcm* represents the average mixture of typical DSP programs then the weighted-average overhead due to the dismantling is smaller than 1%. Although the heuristics mentioned above might have similar impact in the final code quality, using an approach based on natural edges can considerably improve the code quality of critical parts of the program (e.g. loop body).

6 Conclusions

This paper proposes a tree based heuristic for code generation with memory-register architectures which satisfy the RTG criterion. It shows that decomposing DAGs into trees using the concept of natural edges and performing tree code generation is an effective approach for this type of architecture. The fundamental reasons that support this claim can be summarized as follows: (a) for memory-register architectures some DAG edges can be natural edges, making memory spilling a natural operation for this type of target architecture; (b) the experiments have shown that the majority of the DAGs in the DSPstone benchmark are trees, for which an optimal $O(n)$ code generation algorithm [6] exist.

References

[1] C. Liem, Trevor M, and Paulin P. Instruction-set matching and selection for DSP and ASIP code generation. In *European Design and Test Conference*, pages 31–37, 1994.

[2] P. Marwedel. Tree-based mapping of algorithms to prediffined structures. In *Int.Conf. on Computer-Aided Design*, pages 586–593, 1993.

[3] Lanner D., Cornero M., Goosens G., and De Man H. Data routing: a paradigm for efficient data-path synthesis and code generation. In *High-Level Synthesis Symposium*, pages 17–22, 1994.

[4] Texas Instruments, Inc. *Digital Signal Processing Applications with the TMS320 Family*, 1990.

[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] G. Araujo and S. Malik. Optimal code generation for embedded memory non-homogeneous register architectures. In *Proc.* 8^{th} *International Symposium on System Synthesis*, pages 36–41, September 1995.

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

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