# Code Optimization Techniques for Embedded DSP Microprocessors

Stan Liao   Srinivas Devadas
MIT Department of EECS
Cambridge, MA 02139

Kurt Keutzer   Steve Tjiang   Albert Wang
Synopsys, Inc.
Mountain View, CA 94043

**Abstract—We address the problem of code optimization for embedded DSP microprocessors. Such processors (e.g., those in the TMS320 series) have highly irregular datapaths, and conventional code generation methods typically result in inefficient code. In this paper we formulate and solve some optimization problems that arise in code generation for processors with irregular datapaths. In addition to instruction scheduling and register allocation, we also formulate the accumulator spilling and mode selection problems that arise in DSP microprocessors. We present optimal and heuristic algorithms that determine an instruction schedule simultaneously optimizing accumulator spilling and mode selection. Experimental results are presented.**

**Keywords—code generation, optimization, digital signal processors**

## I. Introduction

An increasingly common micro-architecture for embedded systems is to integrate a microprocessor or microcontroller, a ROM and an ASIC all on a single IC. Such a micro-architecture can currently be found in many diverse embedded systems, e.g., FAX modems, laser printers, and cellular telephones.

The programmable component in embedded systems can be an application-specific instruction processor (ASIP), a general-purpose microprocessor such as the SPARC, a microcontroller such as the Intel 8051, or a digital signal processing (DSP) microprocessor such as the TMS320C25. This paper focuses on the DSP application domain, where embedded systems are increasingly used. Many of these systems use processors from the TMS320C2x, 56K or ADSP families, all fixed-point DSP microprocessors with irregular datapaths.

As the complexity of embedded systems grow, the need to decrease development costs and time to market mandates the use of high-level languages (HLLs) in programming DSP processors; only short, time-critical portions of the program can be assembly-coded. Recent statistics from Dataquest support this trend: high-level languages (HLLs) such as C (and C++) are gradually replacing assembly language, because using HLLs greatly lowers the cost of development and maintenance of embedded systems. However, current compilers for fixed-point DSP microprocessors generate poor code — thus programming in a HLL can incur significant code performance and code size penalties.

While optimizing compilers have proved effective for RISC processors, the irregular datapaths and small number of registers found in DSP processors remain a challenge to compilers. The direct application of conventional code optimization methods (e.g., [1]) has, so far, been unable to generate code that efficiently uses the features of fixed-point DSP microprocessors.

Code size matters a great deal in embedded systems since program code resides in on-chip ROM, the size of which directly translates into silicon area and cost. Designers often devote a significant of time to reduce code size so that the code will fit into available ROM; exceeding on-chip ROM size could require expensive redesign of the entire IC [6]. As a result, a compiler that automatically generates small, dense code will result in a significant productivity gain as well.

There has been relatively little previous work in the area of code generation for DSP processors. Cheng and Lin present methods for code generation for the TMS320C40 in [2]. The algorithms they use are similar to high-level scheduling and allocation methods. Various groups in the hardware design community have recently started working on the problem of retargetable code generation for embedded processors [9]; the focus, however, has been on horizontally microcoded architectures.

In this paper, we develop code optimization techniques for DSP microprocessors with irregular datapaths that improve code performance and reduce code size. Our techniques are applicable to a broad class of DSP microprocessors, including those in the TMS320, DSP56K, and ADSP series. The optimization problems we target include instruction scheduling and register allocation. We also formulate the *accumulator spilling* and *mode selection* problems that arise in DSP microprocessors. The individual problems of scheduling and register allocation have traditionally been solved independently in compilers. We present optimal and heuristic algorithms that determine an instruction schedule while simultaneously optimizing accumulator spilling and mode selection. The experimental results we have obtained show significant improvements over existing code generation methods.

The paper is organized as follows. We use the TMS320C25 to illustrate our architecture model in Section II. In Section III we formulate the mode selection problem. We describe the general register allocation problem in Section IV and go on to formulate the accumulator spilling problem that is specific to DSP microprocessors. A branch-and-bound scheduling algorithm that produces a schedule for a basic block with a minimal number of accumulator spills and mode switches is presented in Section V. Experimental results are presented in Section VI. We conclude in Section VII with our ongoing research.

## II. Example: TMS320C25

We describe Texas Instruments' popular TMS320C25 DSP microprocessor [7], highlighting the key features of this irregular architecture that are not addressed in traditional compiler optimizations. Fig. 1 shows a simplified model of its datapath.

The TMS320C25 is an accumulator-based machine. In addition to the usual ALU, there is a separate multiplier which takes input from the T register and memory and places the result in the P register. With the separate multiplier the machine can execute one-cycle multiply-accumulate operations. Note that there are no general-purpose registers other than the accumulator. Most operations involve an operand taken from the memory.

**32nd ACM/IEEE Design Automation Conference ®**

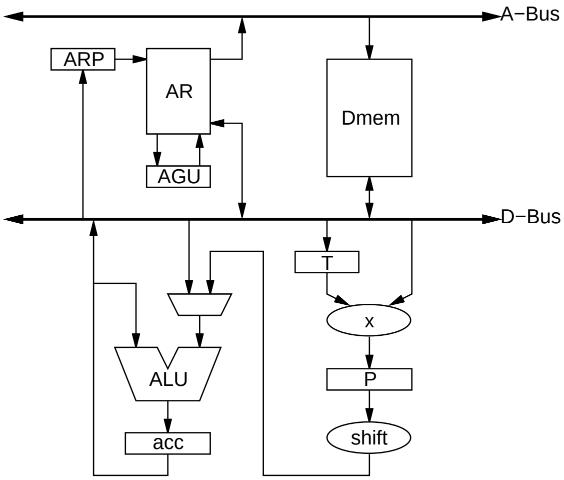


Fig. 1. TMS320C25 datapath (simplified model)

The memory is addressed by the address register file (AR0 through AR7), which is in turn addressed by the 3-bit address register pointer (ARP) (denoting the "current" AR). The address generation unit (AGU) allows the current AR to be auto-incremented or auto-decremented during the execution of any TMS320C25 instruction that uses indirect addressing mode.

Another feature of the TMS320C25 that is usually missing in general register machines is the use of *modes* (or *residual control* in microprogramming terminology). The most commonly used mode classes are sign-extension and product-shift. Some instructions are affected by the setting of the modes, and if the current mode setting is different from that desired by the instruction, then it must be set to the appropriate values.

The use of address registers and modes is likely intended to allow for more compact code. However, this means that the compiler's job is made more difficult. In the subsequent sections we will examine the problem of mode settings and present a formulation for the generalized problem. We will then extend this framework for the problem of minimizing the spills of the accumulator.

## III. Mode Optimization Problem

The goal of mode optimization is to schedule the instructions so that the number of mode-setting instructions is minimized.

### A. Simple Mode Optimization

Let the DAG $G = \langle V, E \rangle$ for a basic block be given. Let $r$ be the number of modes, and $l : V \rightarrow \{1, \ldots, r\}$ label each node $v \in V$ with a mode $l(v)$. Let $C = [c_{ij}]$ be the cost matrix, where $c_{ij} \geq 0$ is the cost of switching from mode $i$ to mode $j$. We assume that the following hold for the cost matrix $C$:

$$c_{ii} = 0, \quad \text{for all } i \tag{1}$$

$$c_{ik} \leq c_{ij} + c_{jk}, \quad \text{for all } i, j, k \tag{2}$$

Inequality (2) (triangular inequality) will be used later to establish lower bounds in the branch-and-bound algorithm. The *simple mode optimization problem* (SMOPT) is the problem of finding a linear schedule $S$ that is a topological sort of $G$, $(v_1, v_2, \ldots, v_n)$, such that

$$\text{mode\_cost}(S) = \sum_{i=1}^{n} c_{l(v_i) l(v_{i+1})} \tag{3}$$

is minimized.

*Theorem 1:* The decision problem for simple mode optimization is NP-complete.

### B. Multiple Mode Classes and Don't-Cares

In Section III-A we only considered a single mode class. A *mode class* is a set of mutually exclusive modes, and at any point the machine can only be in exactly one of the modes. For example, sign extension and product shift are two mode classes in the TMS320C25. We assume that each mode class has a cost matrix whose values are independent of every other mode class. In the absence of *don't-care*s, multiple mode classes are equivalent to a single mode class: their Cartesian product.

Let us first consider the presence of don't-cares (denoted by $-$) in a single mode class. A node is labeled $-$, if it is not affected by the current mode. Hence, we can first disregard these nodes, try to optimally schedule the other nodes, and then put these nodes back in the schedule, consistently with the original DAG. To do so, we construct from $G$ a *reduced* DAG $G' = \langle V', E' \rangle$ as follows:

*Procedure: Don't-Care Reduction*

1. For each path $v_1, v_2, \ldots, v_{k-1}, v_k$ in $G$ where $l(v_j) = -$, for $2 \leq j \leq k-1$, and $l(v_1), l(v_k) \neq -$, we add an edge $(v_1, v_k)$. This preserves the precedence relationship from $G$.
2. Remove each node $v \in V$ such that $l(v) = -$, and all edges incident on or emanating from $v$.

Now let us consider $m$ mode classes. The label of each node is an $m$-tuple, some or all of whose components may be $-$. For each mode class $p$ we can construct a reduced DAG from $G$ by projecting the label to the corresponding component and then reducing the graph using the above procedure; this derived graph is called the $p$-reduced DAG. We may then schedule these reduced DAGs separately. Each of these schedules is called a *reduced schedule.*

*Definition 1:* Let $S_p$ and $S_q$ be valid reduced schedules for the $p$- and $q$-reduced DAGs. $S_p$ and $S_q$ are said to be *compatible* if they can be merged to form a schedule that is valid with respect to $G$.

For example, the reduced schedules *ABEG* and *AEFG* are compatible; whereas *ABEG* and *AGDE* are not, since they have conflicting orders for nodes $E$ and $G$. We state without proof the following proposition.

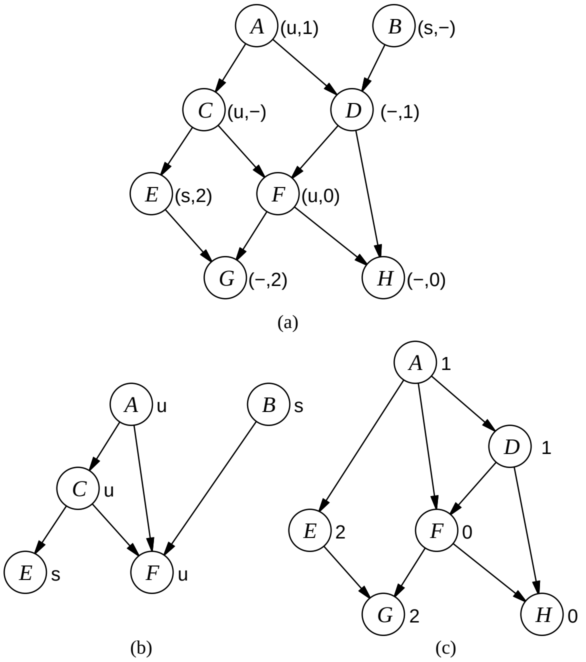
*Proposition 1:* Let $S_p$ be an optimal reduced schedule for the $p$-reduced DAG, $p = 1, \ldots, m$. If all $S_p$'s are compatible, then they can be merged to form an optimal schedule for $G$.

Unfortunately, not all optimal reduced schedules are compatible. Thus, if reduced DAGs are employed in a heuristic algorithm, some non-optimal reduced schedules will have to be chosen in order to achieve compatibility. In a branch-and-bound algorithm, we consider all modes classes simultaneously and compute the lower-bound by resolving the don't-cares conservatively.

### C. Example

Let us consider the expression DAG $G$ shown in Fig. 2(a). There are two mode classes, $p$={u,s} and $q$={0,1,2}. The corresponding reduced DAGs are shown in Fig. 2(b) and (c).

Assuming that any mode change incurs a unit cost, there are two optimal reduced schedules for the $p$-reduced DAG: $S_{p1}$ =*ACBEF* and $S_{p2}$ =*BACFE*. There is only one optimal reduced schedule for the $q$-reduced DAG: $S_q$ =*ADFHEG*. $S_{p1}$ and $S_q$ are not compatible, since the former schedules $F$ after $E$ whereas the latter schedules $F$ before $E$. However, $S_{p2}$ and $S_q$ are compatible, we obtain the optimal schedule for $G$: *BACDFHEG*.



(a)

(b)

(c)

Fig. 2. (a) An expression DAG $G$. (b) $p$-reduced DAG. (c) $q$-reduced DAG.

## IV. Register Allocation and Accumulator Spilling

This section describes the effect of the scheduling step on the number of values that have to be stored to perform a given computation. We first focus on conventional architectures and then switch focus to irregular datapaths such as the TMS320C25.

### A. RISC Architectures

*Register allocation* deals with allocating variables in the given basic block to a minimum number of registers. If at any time we have to store a set of values whose cardinality exceeds the number of available registers, *spilling* into memory is the only alternative. The register allocation problem for RISC machines is quite different from machines such as the TMS320C25 since the latter does not have a register file. However, in both cases it is possible to arrive at a cost function for scheduling that simplifies the succeeding register allocation step.

### B. Effect of Scheduling on Register Allocation

It is well known that different schedules corresponding to a data-dependency graph require differing number of registers. We formalize this effect in the sequel.

We are given a basic block represented as a DAG $G = \langle V, E \rangle$. We will assume a sequential model of processor execution, however, the discussion can easily be generalized to include parallelism corresponding to multiple execution threads. Each node $v \in V$ is assumed to have two input variables $i_1(v)$ and $i_2(v)$ and an output variable $o(v)$. We further assume the graph is in static single assignment form, that is, every assignment is to a unique variable.

If we schedule the nodes in $G$, the the life-times of all the variables can be calculated. The **life-time** of a variable is the duration between its unique assignment and last use. Since most processors allow the reading and writing of values into a register in a single instruction, if a variable is written in instruction $i$ and is read in instruction $j > i$, we will denote its life-time as the (open-ended) interval $[i, j)$.

```
v1 = v2 + v3
v4 = v2 - v3
v5 = v1 * v2
v6 = v4 & v3
v7 = v5 | v6
```

(a)

```
R1 = R2 + R3
R4 = R2 - R3
R1 = R1 * R2
R4 = R4 & R3
R4 = R1 | R4
```

(b)

```
R1 = R2 + R3
R1 = R1 * R2
R2 = R2 - R3
R2 = R2 & R3
R3 = R1 | R2
```

(c)

Fig. 3. (a) Code sequence (b) Register allocation on original code sequence (c) Register allocation on reordered code sequence

Variables with non-overlapping life-times can be merged into the same register. For example, the variable $i_1(v)$ with life-time $[i, j)$ can be merged with variable $i_2(v)$ with life-time $[k, l)$ if $k \geq j$.

The number of registers required is proportional to the overlap of the live periods of the variables, or to put it differently, the number of registers required is the *maximal density* of variable life-times across the entire sequence. Given a set of variable life-times in a schedule we can compute the maximal density in linear time.

The *simple register allocation problem* is to find the best possible grouping of variables with non-overlapping life-times into a minimum number of sets. Given a fixed schedule the simple register allocation problem (SRAOPT) can be solved in polynomial time. (A polynomial-time solution is only possible when static single assignment for each variable is assumed, in which case the interference graph is an *interval graph*, which can be colored in polynomial time [3].) However, there is freedom in the ordering of the nodes of the given DAG as long the dependency constraints are not violated. Given a code sequence, exploiting this freedom can result in a smaller set of registers being required. This is illustrated in Fig. 3. In Fig. 3(a), an example code sequence being executed on a processor with a single arithmetic unit is shown. Without changing the order of the operations in the code sequence, the minimum number of registers required is 4, as shown in Fig. 3(b). Allowing re-ordering of operations within the sequence produces a 3 register solution in Fig. 3(c). (A similar example was given in [10].)

Finding the optimal ordering of operations within a sequence, so as to allocate a minimum set of registers reduces to a one-dimensional linear arrangement problem similar to SMOPT of Section III. The *register allocation problem* (RAOPT), involves finding a schedule $S$ that is a topological sort of a basic block represented by a DAG $G$, $(v_1, v_2, \ldots, v_n)$, such that

$$\text{reg\_cost}(S) = \max_{i=1}^{n} \text{ density}(i) \tag{4}$$

is minimized, where $\text{density}(i)$ corresponds to the number of variables that are live at instruction $i$.

### C. *DSP Microprocessor Architectures*

Register allocation for processors with a general-purpose register file is relatively straightforward. Obtaining a schedule that minimizes the maximal density of variable life-times will result in minimal spilling into memory.

For processors such as the TMS320C25, the register allocation problem is more complicated due to several reasons, the foremost of which is the indirect addressing mechanism that is used in the datapath.

- The TMS320C25 has a single accumulator and no general-purpose registers.
- The address registers `AR0` through `AR7` are in turn addressed by the register `ARP`.
- The address registers `AR0` through `AR7` can be auto-incremented and auto-decremented during the execution of any TMS320C25 instruction that uses indirect addressing mode. However, loading a new address requires a separate instruction.
- The `ARP` can be switched to point to a different address register during any instruction using indirect addressing mode.

All of the above complications can be taken into account by formulating the register allocation problem for a fixed code schedule as an *offset assignment problem* [8]. In this paper, we will focus on the first item above, and formulate the minimal spilling problem for accumulator-based architectures.

In the TMS320C25, instructions such as `ADD`, `MAC` and `LAC` (load-accumulator) write into the accumulator. The `MPY` instruction writes into the `P` register. Given a schedule it is easy to determine the number of accumulator spills using life-time analysis. If the accumulator value is used in the immediately following instruction and nowhere else, then spilling into memory is not required, else we need to spill the contents into memory for later access. Different schedules will result in different numbers of accumulator spills.

The *accumulator spilling problem* (ASOPT), involves finding a linear schedule $S$ that is a topological sort of a basic block represented by a DAG $G$, $(v_1, v_2, \ldots, v_n)$, such that

$$\text{spill\_cost}(S) = \text{Number of accumulator spills in } S \qquad (5)$$

is minimized.

## V. A Branch-and-Bound Algorithm for Scheduling

In this section we present a branch-and-bound algorithm which, given a basic block represented as a DAG, determines an optimal code schedule under a specified cost function.

### A. *Cost Function*

The cost we use includes the number of accumulator spills required by the schedule $S$ (see Eqn. (5)) as well as the number of mode switches required (see Eqn. (3)).

The cost function we use in the branch-and-bound method is:

$$C(S) \;=\; W_S \times \text{spill\_cost}(S) \;+\; W_M \times \text{mode\_cost}(S)$$

where $W_S$ and $W_M$ depend on the relative cost of instructions required to spill the accumulator to memory and instructions required to accomplish a mode switch.

### B. *Branching Search*

Branching over all possible solutions is accomplished using the recursive branching strategy of Fig. 4. Initially, **find-optimal-schedule**() is called with the original DAG $G = \langle V, E \rangle$, and $P = \phi$ as parameters. It returns the best schedule $S$ and the cost of the schedule $C(S)$.

```
find-optimal-schedule( $G$, $P$ ):
{
    /* $G$ = DAG of basic block */
    /* $P$ = Current partial schedule for DAG */
    /* $C(P)$ = Cost of partial schedule */
    $N$ = find-scheduleable-nodes( $G$, $P$ );
    if ( $N \equiv \phi$ ) {
        if ( $C(P) < C(S)$ )
            [ $S$, $C(S)$ ] = [ $P$, $C(P)$ ];
            return [ $S$, $C(S)$ ];
    }

    foreach node $v$ in $N$ {
        $LB$ = lower-bound( $G - v$, $P \cup v$ );
        if ( $C(P \cup v) + LB < C(S)$ ) {
            [ $T$, $C(T)$ ] = find-optimal-schedule( $G - v$, $P \cup v$ );
            if ( $C(T) < C(S)$ )
                [ $S$, $C(S)$ ] = [ $T$, $C(T)$ ];
        }
    }
    return [ $S$, $C(S)$ ];
}
```

Fig. 4. Branch-and-bound procedure to determine optimal schedule

The procedure **find-scheduleable-nodes**() determines the set of nodes $N \in V$ that can be scheduled given the partial schedule $P$ such that dependency constraints are not violated. If there are no scheduleable nodes it means that the schedule is complete. If the cost of this complete schedule is less than the best cost seen thus far, we save the complete schedule $P$ as the best schedule and return to the previous level of recursion.

If there are scheduleable nodes in $N$, we select each of them in sequence and recursively call **find-optimal-schedule**(). Once we have chosen a node $v$ to add to $P$, we first compute a lower bound on the cost of any schedule we will see in this recursion path. If the cost of the partial schedule $P \cup v$ plus the computed lower bound on the unscheduled DAG $G - v$ is greater than or equal to the best cost seen thus far, there is no need to explore this recursion path. The best schedule with its associated cost is returned by procedure **find-optimal-schedule**().

### C. *Lower Bound Computation*

The procedure **lower-bound**() is critical to improving the efficiency of the search. If we can compute tight lower bounds, then we can prune the search considerably by reducing the depth of recursion.

Given a DAG $\tilde{G}\langle V, E\rangle$ we need to compute a lower bound over all possible schedules $\tilde{P}$ consistent with $\tilde{G}$. This entails computing a lower bound for mode_cost($\tilde{P}$) and a lower bound for spill_cost($\tilde{P}$).

#### C.1 Lower Bound for Spill Cost

We assume that the accumulator has no useful value upon entry to the basic block in Eqn. (5) as well as in the lower bound computation below.

We mark the nodes in the given DAG $\tilde{G}$ using the steps below. Initially all nodes are unmarked.

1. Nodes whose outputs are outputs of the basic block and which write into the accumulator (e.g., `MAC` and `ADD`) will spill their contents, and these nodes are marked.

2. If a node $v$ in the $\tilde{G}$ has more than 2 fanouts, it means that $o(v)$ is used as an input in two other instructions and this implies that the accumulator contents corresponding to $o(v)$ has to be spilled into memory.
3. If node $v$ receives inputs from nodes $x$ and $y$ which correspond to instructions that write into the accumulator, then either $o(x) = i_1(v)$ or $o(y) = i_2(v)$ has to be spilled. Therefore, if both $x$ and $y$ are unmarked, we will mark $x$ or we will mark $y$ (but not both).

The number of marked nodes in $\tilde{G}$ corresponds to a lower bound on the number of accumulator spills in any schedule consistent with $\tilde{G}$.

### C.2 Lower Bound for Mode Cost

To estimate a lower bound for the mode cost given an unscheduled DAG $\tilde{G}$, we simply compute the maximum cost for mode switching using Eqn. (3) along *any path* of $\tilde{G}$. This follows from Inequality (2), since any schedule must contain this path as a subsequence, and the cost of switching from mode $i \rightarrow j$ cannot be greater than that of mode $i \rightarrow j \rightarrow k$ for any $k$ (for otherwise we can replace the former by the latter).

### *D. Hashing*

The branching procedure of Fig. 4 may perform a significant amount of redundant computation. Consider a situation where we have constructed a partial schedule $P_1$ which corresponds to the set of nodes $V_1$. The optimal scheduling subproblem is then solved for $G - V_1$ with appropriate initial conditions corresponding to the mode and accumulator contents of the last instruction in $P_1$. Now, if we compute a different partial schedule $P_2$ corresponding to the same set of nodes $V_1$, and the last instruction in $P_2$ is the same as the last instruction in $P_1$, we solve exactly the same optimal scheduling subproblem for $G - V_1$. However, there is no mechanism in the procedure of Fig. 4 to detect that we have solved the subproblem for $G - V_1$ already.

The lower bounding technique alleviates the above inefficiency to a certain extent. However, since the bounds may not always be tight, significant redundant computation may occur.

A hashing mechanism of “remembering” previously computed optimal solutions for parts of the original DAG can greatly improve the efficiency of the search. We hash each partial schedule $P_i$ such that $|P_i| \leq L$, where $L$ is a user-specified parameter in the range $2 \leq L \leq |V|$. The hash is computed in such a way that if different partial schedules $P_i$ and $P_j$ contain the same set of nodes $V_1$, the same hash is computed. Once the subproblem of finding an optimal schedule for $G - V_1$ has been solved for $P_i$, we store the result in a hash table that is accessible by the hash for $P_i$. Before we begin to solve the subproblem associated with $P_j$, we check to see if $|P_j| \leq L$, if so we compute the hash for $P_j$ and access the hash table. If we get a “hit” in the hash table, we immediately return the best solution for the subproblem of scheduling $G - V_1$. A hit in the hash table implies that $P_i$ and $P_j$ correspond to the same set of instructions, and further that the last instructions in $P_i$ and $P_j$ are the same.

### *E. Heuristics*

Once the nodes in $N$ have been determined by the procedure **find-scheduleable-nodes**(), we can recursively call **find-optimal-schedule**() for each of the nodes in $N$ in any order. However, to improve efficiency it is worthwhile to first explore partial solutions that have a good chance of being extended to optimal solutions. This determination can only be made heuristically.

We sort the nodes in $N$ based on a cost estimate to obtain a sorted list $sort(N)$. The cost estimate for any $v \in N$ is equal to $C(P \cup v) - C(P)$ which includes both the spill cost and the mode switching cost. Nodes in $N$ are sorted in increasing order of this cost estimate.

| basic | fixed costs | | | orig. sched. | | |
|---|---|---|---|---|---|---|
| block | I | C | V | L | S | M |
| SpeedCtl386 | 11 | 0 | 5 | 6 | 1 | 10 |
| SpeedCtl389 | 11 | 0 | 6 | 7 | 1 | 8 |
| Compaction3 | 13 | 0 | 6 | 12 | 6 | 6 |
| FFTBR28 | 22 | 4 | 9 | 17 | 4 | 11 |
| ChenDct1 | 78 | 20 | 17 | 47 | 11 | 25 |
| ChenDct4 | 80 | 20 | 17 | 46 | 11 | 25 |
| ChenIDct1 | 78 | 26 | 10 | 55 | 20 | 33 |
| ChenIDct4 | 72 | 18 | 10 | 39 | 12 | 17 |
| LeeDct1 | 76 | 17 | 25 | 45 | 4 | 26 |
| LeeDct4 | 50 | 8 | 21 | 31 | 2 | 16 |
| LeeIDct1 | 79 | 26 | 10 | 49 | 17 | 32 |
| LeeIDct4 | 64 | 15 | 20 | 35 | 2 | 21 |

TABLE I
FIXED COSTS AND ORIGINAL SCHEDULE

| basic | heur. sched. | | | ratio | ratio |
|---|---|---|---|---|---|
| block | L | S | M | (LSM) | (all) |
| Speedctl386 | 5 | 0 | 8 | 0.764 | 0.879 |
| Speedctl389 | 6 | 0 | 6 | 0.750 | 0.879 |
| Compaction3 | 7 | 1 | 2 | 0.417 | 0.674 |
| FFTBR28 | 12 | 2 | 6 | 0.625 | 0.820 |
| ChenDct1 | 32 | 6 | 6 | 0.530 | 0.803 |
| ChenDct4 | 31 | 6 | 6 | 0.524 | 0.804 |
| ChenIDct1 | 29 | 6 | 5 | 0.370 | 0.694 |
| ChenIDct4 | 29 | 6 | 5 | 0.588 | 0.833 |
| LeeDct1 | 38 | 5 | 2 | 0.600 | 0.844 |
| LeeDct4 | 26 | 1 | 2 | 0.592 | 0.844 |
| LeeIDct1 | 25 | 2 | 2 | 0.296 | 0.676 |
| LeeIDct4 | 27 | 2 | 4 | 0.569 | 0.840 |

TABLE II
HEURISTIC SCHEDULE AND RATIOS

For small to moderate-sized basic blocks the optimal algorithm of Fig. 4 that explores all possible solutions is viable. For large basic blocks, we have to resort to heuristic search techniques. A fast, greedy heuristic is based on the node sorting method described above. We only explore solutions corresponding to the first $t$ nodes in the sorted list $sort(N)$. The **foreach** loop of Fig. 4 is replaced with $t$ calls to **find-optimal-schedule**(), where typically $1 \leq t \leq 3$.

## VI. EXPERIMENTS AND RESULTS

We have implemented the heuristic algorithm of Section V-E to perform scheduling for minimum cost. Our experiments are based on a code generator for a simplified TMS320C25 architecture with all the features described in Section II. A full-featured TMS320C25 code generator is currently under development (see Section VII).

To accurately account for the various types of costs, we attribute the following cost components to each node.

- Instruction (I). Each node in the DAG is associated with an instruction that has a cost of 1.
- Common subexpression (C). If the node has two or more uses, it is

a common subexpression. Under our assumption of aggressive common subexpression elimination, the result of this node is stored to memory rather than be recomputed at a later time.

- Live-on-exit variable (V). The result of the computation for this node is live upon exit of this basic block.
- Load (L). One operand of the node is not in the accumulator; therefore, it needs to be loaded.
- Spill (S). The result of the previously scheduled node will be used later but not now.
- Mode change (M). The node requires a mode setting different from the current setting. The mode classes considered are sign-extension and product-shift.

The first three items are fixed costs. Under any schedule, the number of instructions, common subexpressions, and live-on-exit variables remains the same. Therefore, the only optimizable costs are those of loads, spills, and mode changes.

We present our experimental results in Tables I and II. The former gives the original schedule generated by the front-end after common subexpression elimination. This schedule closely resembles that found in the source code. The latter shows the results we have obtained after our heuristic scheduling.

SpeedCtl is a routine in an ADPCM transcoder applying the CCITT recommendation G.721. Compaction is a notch-filter routine. FFTBR is an Fast Fourier Transform routine with a mechanism to prevent overflows. These three are taken from the DSPstone benchmark suite [11]. ChenDct, ChenIDct, LeeDct, and LeeIDct are discrete cosine transform routines in a JPEG package. We have chosen the largest basic blocks from these routines. The column labeled “ratio (LSM)” gives the ratio of only the optimizable costs (loads, spills, and mode changes), where as the “ratio (all)” column gives the ratio with the fixed costs (I, C, and V) taken into account. These results are very encouraging. Even with the simple heuristic we were able to achieve substantial improvement over the schedule given by the front-end.

## VII. Conclusions and Ongoing Work

Code generation for irregular datapaths, such as those used in DSP microprocessors, is a problem that has received relatively little attention to date. With the advent and increasing use of embedded systems, this problem has become very important. In this paper we presented scheduling algorithms that are able to exploit the features of the TMS320C25 microprocessor. Our initial results indicate that these algorithms obtain substantial improvements in code size and performance over conventional code generation techniques.

We are currently developing a framework for retargetable code generation [9]. There are many avenues for further work in this area. Our framework is directly applicable to traces [4][5] rather than just basic blocks, and experiments on traces will be conducted in the near future. Traces will allow for more global optimization and afford the possibility of even greater savings over conventional optimization. One way to avoid the possible code explosion caused by trace scheduling is to restrict the movement across basic blocks to mode-setting instructions. This way we ensure that along the most frequent traces the number of such instructions is minimized.

The framework can be easily generalized to accumulator-based machines which also have a general-purpose register file such as the TMS320C40. This can easily be done by adding Eqn. (4) to the cost function.

Storage assignment [8] is a very important post-scheduling problem that has to be solved in order to ensure that memory accesses have minimal cost. For machines (such as the TMS320C25) without indexing addressing mode, variables are accessed through address registers (the ARs) and it is desirable that the auto-increment/decrement feature be efficiently utilized. The placement of variables in storage has a significant impact on the size and performance of the generated code. For instance, if variables are accessed in the order abacd and the following assignment is made: a:1, b:2, c:3, and d:4, accessing a followed by c requires an explicit instruction to increase the AR by two; every other access can be accomplished using auto-increment or decrement. On the other hand, if we use the assignment: a:2, b:1, c:3, and d:4, then all changes in the AR can be done via auto-increment or decrement; no explicit instruction for changing the AR is necessary. This problem is related to the mode optimization problem in that the AR can be considered a mode class and our goal is again to minimize the number of “mode-setting” instructions. Its relationship with scheduling is, however, much more complicated because before the actual assignment is made, the information is only symbolic, and it is very difficult to estimate the effect of scheduling on offset assignment.

Finally, to fully exploit the features of many DSP microprocessors, zero-overhead loops have to be detected and appropriate code generated wherever possible.

## VIII. Acknowledgements

This research was supported in part by the Advanced Research Projects Agency under contract DABT63-94-C-0053, and in part by a NSF Young Investigator Award with matching funds from Mitsubishi Corporation.

## References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] W-K. Cheng and Y-L. Lin. Code Generation for a DSP Processor. In *Proceedings of the Int’l Symposium on High-Level Synthesis*, pages 82–87, May 1994.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[4] John R. Ellis. *A Compiler for VLIW Architectures*. MIT Press, 1985.

[5] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trnsactions on Computers*, C-30(7):478–490, 1981.

[6] J. G. Ganssle. *The Art of Programming Embedded Systems*. San Diego, CA: Academic Press, Inc., 1992.

[7] Texas Instruments. *TMS320C2x User’s Guide*. Texas Instruments, January 1993. Revision C.

[8] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage Assignment to Decrease Code Size. In *Proceedings of ACM SIGPLAN’95 Conference on Programming Language Design and Implementation*, June 1995.

[9] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995. Proceedings of the 1994 Dagstuhl Workshop on Code Generation for Embedded Processors.

[10] S. S. Pinter. Register Allocation with Instruction Scheduling: a New Approach. In *Proceedings of the ACM Programming Language Design and Implementation Conference*, pages 248–257, June 1993.

[11] V. Živojnović, J. Martínez Velarde, and C. Schläger. DSPstone: A DSP-oriented Benchmarking Methodology. Technical report, Aachen University of Technology, August 1994.