A Global Mode Instruction Minimization Technique for Embedded DSPs

Thomas Charles Wilson  
Dept. of Computing and Information Science  
University of Guelph, Guelph, Ont. Canada  
tomw@snowhite.cis.uoguelph.ca

Gary William Grewal  
School of Engineering  
University of Guelph, Guelph, Ont. Canada  
gwg@snowhite.cis.uoguelph.ca

Abstract

This paper addresses the problem of minimizing mode setting instructions for embedded DSPs. Many such processors use a state register to control the mode of ALU operations (e.g., sign extension, round, and shift). Often two or more modes can be changed by a single instruction. A method is given to determine the minimum number of instructions needed to properly set modes, assuming a schedule has been determined. Our approach models the problem as a minimum cover, and is not limited to a basic block. Block frequency information is exploited to encourage mode changes in less frequently executed blocks whenever possible. Special attention is given to the proper optimization of loops.

1. Introduction

Many of today’s digital signal processors (DSP) use a state register containing mode variables to control the behavior of certain ALU instructions. For example, the Motorola 56000 uses mode variables to specify the scaling to be performed in the ALU shifter and the rounding position in the ALU multiply-accumulator. In general mode variables are used for carry and overflow handling, rounding methods, sign extension, shift options, and saturation arithmetic. If an instruction is controlled by mode variables, the variables must be set to the correct values prior to execution of the instruction.

Mode setting instructions must be inserted into the schedule to (re)set these variables to the correct values. Since two or more modes can often be changed at the same time using one instruction, it is useful to determine the minimum number of instructions needed to properly set modes. Of course, many instructions are oblivious to some or all modes, which introduces many possible points at which to reset any particular mode.

In general, a mode may have any number of possible values, and the machine may have multiple modes. A mode class is a set of modes that can be changed with the same instruction. Some DSPs have multiple mode classes. Mode classes are assumed to be disjoint and independent. The problem is to determine the minimum number of mode setting instructions to be inserted for each mode class. By applying the algorithm separately to each mode class, the total minimum number of instructions can be determined.

To date, little work on the problem of mode optimization for DSPs with irregular datapaths has been presented. Liao et. al. [1-2] deal with this problem within a single control block while finding a schedule, using branch and bound, which requires the minimum number of mode changes and spills. In contrast, we assume that the schedule has already been determined, and we consider several (consecutively executed) blocks simultaneously. Our approach considers boundary conditions inherited from surrounding blocks and is sensitive to block frequency issues, in order to install mode changes in less frequently executed blocks whenever possible. This gives our solution a global perspective. We model the problem as a minimum cover problem and solve it using an enhanced genetic algorithm.

In Section 2 we show how our mode optimization module (MOM) models and solves the minimum cover problem. Section 3 demonstrates our methodology for program decomposition. Special attention is given to segments which represent consecutively executing blocks, especially if they constitute a loop. Finally, Section 4 contains our results, and Section 5 the conclusion.
2. Minimum Cover Problem

The mode optimization problem can be modeled as a minimum cover problem. Fig. 1(a) shows a schedule, three modes belonging to the same mode class, and the mode values required by each instruction. (For illustrative purposes, we assume binary mode variables having values: 1 or 2. A don’t care condition, 0, exists if an instruction is not affected by the value of a mode.) The first step is to determine the range of control steps where a mode setting instruction could be inserted for each required mode change (shown as rectangles). This can be done by scanning each column linearly, starting from step 1. Notice that mode 2 must be set to 1 prior to the execution of the instruction on step 1. However, once set, there is no need to reset mode 2 prior to step 5. Hence, steps 2 and 3 do not appear as part of a mode change region.

The scan identifies 9 distinct mode change ranges for the three modes. This information is used to build a 0-1 matrix indicating which instructions occur in each range (Fig. 1(b)). Each column in the matrix represents a range. Each row in the matrix identifies the ranges that could be covered by inserting a mode setting instruction at that point. The objective is to determine the minimum number of instructions (rows) necessary to cover all of the mode changes (columns) in the matrix.

In the following formulation, steps are indexed by \( i \), and mode change ranges by \( r \). \( X_i \) is a 0-1 variable; if \( X_i = 1 \) then mode setting occurs immediately before step \( i \). Matrix \( A \) is a 0-1 matrix indicating by \( A_{ir} = 1 \) whether step \( i \) covers mode change range \( r \). The objective function (1) minimizes the number of mode setting instructions provided (2) that each mode change is covered by at least one instruction:

\[
\begin{align*}
(1) \quad & \min \left[ \sum_i X_i \right] \\
(2) \quad & \forall r: \quad \sum_i X_i A_{ir} \geq 1
\end{align*}
\]

This 0-1 integer linear programming formulation can be solved directly and often efficiently. However, long run times are possible due to the NP-Complete nature of the minimum cover problem [3], especially in the presence of many don’t cares (long ranges) that produce many equivalent solutions.

![Figure 1: Mode table; minimum cover matrix.](image)

To compensate, we have developed a powerful extension to the classical genetic algorithm [4]. This enhanced GA (EGA) can work unattended on linearly constrained problems and converges typically an order of magnitude faster than a classical GA. We have successfully used the EGA in synthesis [5] to do scheduling, module allocation, and binding simultaneously, and efficiently. Furthermore, the EGA automatically adapts to different problem instances without user intervention, making it suitable for use in a code generation system and, in particular, in reducing mode setting instructions.

3. Beyond the Basic Block

This Section describes the strategy for program decomposition and explains how MOM can be applied in a global fashion. Considering multiple blocks simultaneously can avoid redundant mode settings. MOM optimizes entire segments of code. A segment refers to a set of consecutively executing blocks which, at a higher level of abstraction, constitute a single path of control, possibly through a loop. The decomposition first identifies the largest and most frequently executed segments, to encourage early and total optimization of more critical blocks. Less critical segments wind up containing code that adjusts for inter-segment differences.

3.1 Straight-Line Code

Straight-line code conforms to the mode settings inherited from any previously processed surrounding blocks. If a segment has neither its predecessor nor successor blocks predetermined, the optimization occurs without boundary constraints, as described in Section 2. However, if an adjoining block is already processed, the mode settings in effect at the boundaries must be included in the analysis of the current segment and may
require compensating mode setting instructions to be inserted inside the segment. New rows are added to the top and bottom of the table to indicate, respectively, incoming and required outgoing mode values. When the mode leaving the predecessor block is a don't care condition, an instruction inside the segment requiring a specific value must be set inside the segment.

3.2 Loops

Unlike straight-line code, loops dictate their own boundary conditions. Loops require the mode settings at the top and bottom of the loop to match. If the mode settings are consistent with those in the loop’s predecessor and successor blocks, then no problem exists. In the case of a mismatch, an extra mode setting instruction must be inserted outside the loop to resolve the conflict. If the adjacent segment contains straight-line code, this segment will introduce the required instruction anyway. Otherwise, extra bridge blocks may be introduced into the block flow graph to provide a place for an inter-segment adjustment to occur.

If the modes used at the beginning and end of a loop differ, an extra mode setting instruction must be added near either the top or bottom of the loop to resolve the conflict. Known mode settings in the predecessor and successor blocks influence this choice.

Consider Fig. 2(a). The mode setting shown at the top and bottom of the loop reflect the first and last interior uses. Since they differ, the mode must be changed either before the first use or after the last. Figure 2(b) shows the effect of inserting a mode setting instruction at the bottom of the loop. Because the surrounding segments already have definite values, instructions must be inserted into both bridge blocks to adapt the loop to its surroundings. These 2 extra instructions can be avoided simply by altering the modes at the top of the loop instead of the bottom (Fig. 2(c)).

Mode settings in the loop’s predecessor and successor blocks, along with the mode settings at the top and bottom of the loop, determine an appropriate strategy for minimizing mode setting instructions. For an individual mode and segment, there are four general boundary situations: (1) neither the predecessor nor successor is determined; (2) only one of the successor and predecessor is determined; (3) both are determined, and both have identical mode settings; and (4) both are determined, but they differ in their mode settings.

![Diagram of Loop Rescheduling](image)

*Figure 2: Loop rescheduling.*

In cases 1 and 4 the (arbitrary) decision is made to insert the mode setting instruction at the top of the loop. In the remaining 2 cases, the decision depends on the modes specified in the predecessor and successor blocks. Figure 3 (next page) shows the different strategies employed (i.e., Bottom or Top) for each of the 4 boundary conditions. In cases not marked, no change is made inside the loop, and the surrounding blocks must accommodate the loop’s requirements.

4. An Example

Figure 4 shows a simple 4 block structure, along with the schedule and mode settings for each operation. The blocks are grouped into two segments. Blocks 1, 2, and 4, along the most frequently executed path, constitute segment 1 and will be processed first. Segment 2 contains only block 3 and will be processed last. Circles depict scheduled operations.

![Diagram of Original Schedule and Mode Settings](image)

*Figure 4: Original schedule and mode settings.*
Figure 5(a) shows the schedule resulting from optimization of the first segment. Two extra mode setting instructions are inserted into segment 1, resulting in an 8 step schedule. Notice that two or more modes are set in each of these instructions.

At this point, the mode settings at each block boundary in segment 1 are known. The mode settings upon exit from block 1 and entry to block 4 prescribe the boundary conditions during the optimization of segment 2. Figure 5(b) shows the schedule that results. This time, 3 mode setting instructions are required. A "corrective" instruction is required on step 8, just before exiting block 3. This example indicates how extra instructions tend to appear in the less frequently executed segments. (The previous example was run on a SPARCStation 10 and completed in less than 5 seconds, using an enhanced genetic algorithm.)

<table>
<thead>
<tr>
<th></th>
<th>set modes (1,1,1)</th>
<th>set modes (1,1,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

(a) (b)

Figure 5: Resulting schedules for segments.

5. Conclusion

The problem of mode instruction minimization for embedded DSPs is a problem that has received very little attention to date. In this paper, we have presented a general, architecture-independent solution. We have shown how the problem can be modeled naturally as a minimum cover problem, which permits a variety of solution techniques, including genetic algorithms. Our methodology is global, allowing multiple blocks to be considered simultaneously. Every attempt is made to place mode setting instructions in the less frequently executed blocks, and great care is taken to avoid redundant mode settings, especially within loops.

The concepts presented apply to more than just the management of processor modes. They are useful whenever multiple environment or protection conditions can be altered with a single instruction, e.g., with masks, any sort of shift register controls, i/o device modes, etc.

References


Figure 3: Strategies for boundary conditions.