Exploiting Conditional Instructions in Code Generation for Embedded VLIW Processors

Rainer Leupers*
University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany
email: leupers@ls12.cs.uni-dortmund.de

Abstract—This paper presents a new code optimization technique for a class of embedded processors. Modern embedded processor architectures show deep instruction pipelines and highly parallel VLIW-like instruction sets. For such architectures, any change in the control flow of a machine program due to a conditional jump may cause a significant code performance penalty. Therefore, the instruction sets of recent VLIW machines offer support for branch-free execution of conditional statements in the form of so-called conditional instructions. Whether an if-then-else statement is implemented by a conditional jump scheme or by conditional instructions has a strong impact on its worst-case execution time. However, the optimal selection is difficult particularly for nested conditionals. We present a dynamic programming technique for selecting the fastest implementation for nested if-then-else statements based on estimations. The efficacy is demonstrated for a real-life VLIW DSP.

1 Introduction

A major goal of current research efforts in the area of code generation for embedded processors is to overcome the insufficient code quality of current DSP compilers, which still makes time-consuming assembly-level programming of DSP software essential [1]. Traditionally, DSPs have been mainly used for implementation of data flow dominated applications. Consequently, much of the recent work on code optimization for DSPs has focused on data flow graphs, e.g., [2, 3, 4, 5, 6]. However, automotive applications and telecom protocol functions also require a significant amount of control functionality to be implemented by DSPs. Effective compilation techniques are particularly important for control dominated applications, since usually no assembly code library support is available for control functions.

The source code of control dominated applications typically contains a large number of if-then-else (ITE) statements. Classical compiler technology uses conditional jumps for implementation of ITE statements. However, the presence of many conditional jumps has a negative impact on code performance in particular for deeply pipelined and highly parallel VLIW-like processors. Therefore, an alternative architectural support for ITE statements, called conditional instructions, have been implemented in several recent embedded processors. Such instructions may implement ITE statements without altering the control flow in a machine program, i.e., without modifying the program counter.

In this paper, we present a technique for optimized implementation of nested ITE statements for a class of VLIW processors. Since embedded applications mostly have to meet real-time constraints, the goal is to minimize the worst-case execution time of a (control dominated) software function. This is performed by appropriately selecting either a conditional jump or a conditional instruction based implementation scheme for each ITE statement.

The structure of the paper is as follows: Section 2 gives an introduction to conditional instructions and their use in machine programs. Section 3 describes the alternative implementation schemes for ITE statements using conditional jumps and conditional instructions. Sections 4 and 5 describe the proposed two-pass optimization technique. In section 6, we provide experimental results for a recent VLIW DSP, the TI C62xx. Finally, section 7 gives conclusions.

2 Conditional instructions

A conditional instruction ”[C] I" consists of a Boolean condition C and a "regular" machine instruction I, e.g., an arithmetic operation, a register move, or a jump. Instruction I is executed, if and only if the C evaluates to true at the point of time when the control flow in a machine program reaches I. Otherwise, instruction I behaves like a no-operation. The condition C is stored in a machine register Ri. By definition, C is false if Ri = 0, and true otherwise. The notation ”[!C] I" denotes a conditional instruction with a negated condition.

Several recent VLIW-like architectures, such as the Texas Instruments C62xx or the Philips Trimedia TM1000, show full support for conditional instructions. Also the next generation of 64-bit Intel processors will be equipped with this feature [8]. For such machines, the decision of whether to implement an ITE statement by conditional jumps or by conditional instruction execution has a large impact on code quality in terms of worst-case execution time. In the following, we denote these two schemes by c-jump and c-exec.

Using c-exec instead of c-jump may increase the code

* The author acknowledges the support by HP EEsof
performance by a twofold effect: Due to control hazards in the instruction pipeline, execution of jump instructions usually causes a jump penalty, i.e., the pipeline needs to be stalled for some machine cycles. This problem is avoided if conditional instructions are used, because these do not alter the control flow in a program. In turn, this leads to larger branch-free (basic) blocks in the program, which give higher opportunities for parallelization of instructions. On the other hand, using c-exec may also degrade performance due to resource contentions.

Previous work on exploiting conditional instructions during compilation [9, 10] is primarily based on estimated execution times only for single basic blocks. Furthermore, focus has been on minimizing the average execution time. In contrast, our technique estimates the worst-case execution time, which is more relevant for real-time applications, and it handles complete (possibly nested) ITE statements with multiple blocks at a time.

3 ITE implementation schemes

This section presents and analyzes the different implementation schemes used in our approach. Both for c-jump and c-exec two cases have to be considered, depending on whether or not ITE statements are nested.

3.1 c-jump

Let \( S = (\text{cond}, B_T, B_E) \) be an ITE statement, where \( B_T \) and \( B_E \) are basic blocks. The standard replacement scheme using conditional jumps is:

\[
\begin{align*}
\text{c} & := \text{evaluate}(\text{cond}) \\
& \{\text{c}\} \text{ goto then\_label } \\
& B_T \text{ goto join\_label } \\
\text{then\_label}: & \quad B_T \\
\text{join\_label}: & \quad \ldots
\end{align*}
\]

The condition is evaluated into a register \( c \), and dependent on its value, either \( B_T \) or \( B_E \) get executed. Then, control flow joins at the next instruction after \( S \).

Let \( T(B) \) denote the time to execute a basic block \( B \), and let \( J \) be the (machine-dependent) jump penalty. If the conditional jump is taken (i.e., condition \( c \) is true) then the execution time for the ITE statement \( S \) is \( T_1(S) = J + T(B_T) \). If the jump is not taken, then two jump instructions are executed, and the time is \( T_2(S) = 2 \cdot J + T(B_E) \).

The worst-case execution time (neglecting the time for condition evaluation) is \( T(S) = \max(T_1(S), T_2(S)) \).

3.2 c-exec

For a non-nested ITE statement \( S = (\text{cond}, B_T, B_E) \), the c-exec implementation scheme looks as follows:

\[
\begin{align*}
\text{c} & := \text{evaluate}(\text{cond}) \\
& \{\text{c}\} B_T \text{ goto then\_label } \\
& \{!\text{c}\} B_E \text{ goto join\_label } \\
\text{then\_label}: & \quad B_T \\
\text{join\_label}: & \quad \ldots
\end{align*}
\]

The notation \( \{c\} B \) denotes the conditional execution of all instructions in a block \( B \). The worst-case execution time when using c-exec is \( T(S) = T(B_T \circ B_E) \), where \( \{c\} \) denotes the concatenation of blocks. In total, c-exec leads to a shorter worst-case execution time than c-jump, exactly if:

\[
T(B_T \circ B_E) < \max(J + T(B_T), 2 \cdot J + T(B_E))
\]

Note that, in VLIW processors, \( T(B_T \circ B_E) \) is frequently much less than \( T(B_T) + T(B_E) \), because the instructions in \( B_T \) and \( B_E \) may be partially executed in parallel. On the other hand it is obvious that c-exec is not guaranteed to be the fastest alternative in any case.

3.3 c-jump with precondition

The above c-jump and c-exec implementation schemes are in general only valid for innermost ITE statements, where the then and else blocks are basic blocks. However, in general we have to cope with nested ITE statements. As shown above, the c-exec scheme requires that the then and else blocks \( B_T \) and \( B_E \) of ITE statement \( S \) be executed dependent on a condition \( c \). In this case, we say that the statements in \( B_T \) and \( B_E \) have \( c \) as a precondition. If some statement \( S' \) in \( B_T \) or \( B_E \) is an assignment or a jump, we can simply attach the precondition \( c \) to \( S' \) by forming a conditional instruction \( \{c\} S' \). However, if \( S' \) in turn is an ITE statement \((d', B_T', B_E')\), then both \( B_T' \) and \( B_E' \) must not be executed if precondition \( c \) is false, independent of the value of \( d' \). Thus, in order to retain the correct program behavior, preconditions have to be propagated to the inner ITE statements. This requires the following generalized implementation schemes.

Let \( S = (\text{cond}, B_T, B_E) \) be an arbitrary ITE statement, and let \( p \) be the precondition of \( S \). Then, the following c-jump implementation scheme is used:

\[
\begin{align*}
& \{p\} \text{ c} := \text{evaluate}(\text{cond}) \\
& \{!p\} \text{ c} := 0 \\
& \{c\} \text{ goto then\_label } \\
& \{!c\} \text{ goto join\_label } \\
\text{then\_label}: & \quad B_T \\
\text{join\_label}: & \quad \ldots
\end{align*}
\]

The whole statement \( S \) must only be executed if \( p \) is true. The condition for executing \( B_T \) is \( p \land \text{cond} \), while \( B_E \) must be executed exactly if \( p \land \text{NOT}(\text{cond}) \) is true. After execution of the conditional instruction \( \{!p\} \text{ c} = 0 \), the condition register \( c \) contains the value of \( p \land \text{cond} \). If this value is true, then a jump to \( B_T \) is taken, and \( B_T \) is executed unconditionally. If the jump is not taken, then either \( p \) or \( \text{cond} \) are false. By propagating \( p \) as a precondition to \( B_E \) it is guaranteed, that \( B_E \) is executed if and only if \( p \land \text{NOT}(\text{cond}) \) is true.

3.4 c-exec with precondition

Alternatively, a c-exec scheme can be used, in which case the implementation scheme is the following:
of additional instructions required to compute preconditions contributes to the total execution time. This setup code consists and jump c-exec.

First (bottom-up) pass, a cost estimation table is computed both bottom-up and top-down through different ITE nesting levels. We therefore use a two-pass dynamic programming technique described in the following two sections. In the first (bottom-up) pass, a cost estimation table is computed for each ITE statement. The tables are used in the second (top-down) pass to select the fastest implementation for the ITE statements at each level.

### 4 Cost table computation

**Setup costs:** The setup cost of an ITE statement $S$ is defined as the number of instructions in the setup code in the implementation of $S$. The setup cost is a constant implied by the chosen implementation scheme and the existence of a precondition for $S$. Thus, it can be determined by table lookup. Table 1 shows the setup costs for the implementation schemes shown in sections 3.1 to 3.4 and defines a notation for each case.

<table>
<thead>
<tr>
<th>S</th>
<th>c-jump</th>
<th>c-exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>precondition: no</td>
<td>$N_{c-jump}$ = 0</td>
<td>$N_{c-exec}$ = 0</td>
</tr>
<tr>
<td>precondition: yes</td>
<td>$P_{c-jump}$ = 1</td>
<td>$P_{c-exec}$ = 3</td>
</tr>
</tbody>
</table>

**Statement blocks:** Let $B = \{s_1, \ldots, s_t\}$ be a block of statements. We denote the estimated costs of $B$ by $T^P (B)$ (if $B$ has a precondition) and $T^N (B)$ (if $B$ has no precondition), respectively. The costs of a block $B$ are defined as the sum of the costs $T^P (s)$ or $T^N (s)$ of all statements $s$ in $B$. If statement $s_i$ is an assignment or an unconditional jump, then we set $T^P (s_i) = T^N (s_i) = 1$, because the execution time of an assignment or a jump instruction does not depend on whether the instruction is conditional. Otherwise, $s_i$ is in turn an ITE statement $s_i = (c, B_T, B_E)$, in which case the costs of the two alternatives c-jump and c-exec must be estimated.

**c-jump scheme:** Since we compute the cost tables bottom-up, the cost values for the blocks $B_T$ and $B_E$ are already known when the costs for $s_i$ are estimated. If we implement $s_i$ by c-jump and $s_i$ has no precondition, then both $B_T$ and $B_E$ have no precondition, and we estimate the worst-case execution time by $T^N (B)$ (cf. section 3.1):

$$T_{c-jump}^N (s_i) = N_{c-jump} + \max (J + T^N (B_T), 2 \cdot J + T^N (B_E))$$

where $T^N (B_T)$ (the cost of the fastest implementation of $B_T$ in absence of a precondition) is given as

$$T^N (B_T) = \min (T_{c-jump}^N (B_T), T_{c-exec}^N (B_T))$$

and analogously for $T^N (B_E)$. If $s_i$ has a precondition, then, according to the implementation scheme from section 3.3, the else block $B_E$ also has a precondition, while $B_T$ has none. Thus, we obtain

$$T_{c-jump}^P (s_i) = P_{c-jump} + \max (J + T^N (B_T), 2 \cdot J + T^P (B_E))$$

with

$$T^P (B_E) = \min (T_{c-jump}^P (B_E), T_{c-exec}^P (B_E))$$

1 Actually, a larger setup cost table with two further dimensions is required: First, if an ITE statement has an empty else block, then the c-jump and c-exec schemes are slightly different, and so are the setup costs. Second, the setup costs depend on whether the target processor directly supports negated conditions.
The estimation functions \( T_{c-ex}^N \) and \( T_{c-ex}^P \) are defined in the following.

**c-exec scheme:** If \( s_1 \) is implemented by c-exec, then the blocks \( B_T \) and \( B_E \) are effectively concatenated to form a single "large" block. Therefore, we need to estimate the parallelization of the instructions in \( B_T \) and \( B_E \). For this purpose, we incorporate a machine-dependent parameter \( K \) that reflects the available instruction-level parallelism, and which has to be determined empirically. The use of \( K \) is based on the following observation: If there is a large difference in the estimated execution times of \( B_T \) and \( B_E \), say \( T(B_T) \gg T(B_E) \), then it is likely that \( B_E \) will almost completely fit into instruction slots not occupied by computations from \( B_T \), so that \( T(B_T \circ B_E) \) is only slightly larger than \( T(B_T) \).

This can be modeled by subtracting a fraction of \( T(B_T) \) from \( T(B_T \circ B_E) \). Furthermore, the possible parallelization of \( B_T \) and \( B_E \) is inversely related to the nesting level \( L(s_1) \) (with \( L(s) := 1 \) for any innermost ITE statement \( s \)) of statement \( s_1 \), because the size of blocks \( B_T \) and \( B_E \) tends to grow with \( L(s_1) \), leading to more resource contentions between the blocks. For estimating the times for \( B_T \) and \( B_E \), we use the \( T^P \) values as defined in section 4, since in the c-exec scheme both blocks have to be executed under a precondition. Summarizing, a useful estimation is

\[
T(B_T \circ B_E) = M + m - z
\]

with

\[
M = \max(T^P(B_T), T^P(B_E))
\]

\[
m = \min(T^P(B_T), T^P(B_E))
\]

\[
z = \min\left(\frac{K}{L(s_1)} \cdot M, m\right)
\]

The "\( \min \)" in the latter formula ensures that \( T(B_T \circ B_E) \) is never estimated less than \( M \). Like for c-jump, we compute two cost estimations for c-exec, again dependent on the presence of a precondition:

\[
T_{c-ex}^N(s_1) = N_{c-ex} + M + m - z
\]

\[
T_{c-ex}^P(s_1) = P_{c-ex} + M + m - z
\]

### 5 Implementation selection

After bottom-up cost table computation, each ITE statement has been annotated with the four values \( T_{c-jump}^N, T_{c-jump}^P, T_{c-ex}^N \), and \( T_{c-ex}^P \). In a top-down pass, we can now select the best implementations for each ITE statement, starting from the "root" statement \( S^* = \text{cond} \), \( B_T \), \( B_E \). When selecting the implementation for \( S^* \), we can exploit the fact that \( S^* \) cannot have a precondition, since it is the outermost ITE statement. Therefore, we just need to compare the values \( T_{c-jump}^N(S^*) \) and \( T_{c-ex}^N(S^*) \). Suppose, \( T_{c-jump}^N(S^*) < T_{c-ex}^N(S^*) \). Then, c-jump is the faster implementation. Furthermore, from the implementation scheme in section 3.1 we know that \( B_T \) and \( B_E \) will have no precondition. Thus, for all ITE statements in \( B_T \) and \( B_E \), we only need to compare the \( T_{c-jump}^N \) and \( T_{c-ex}^N \) values in order to decide on the best implementation. From this decision, we in turn know which values need to be compared for the ITE statements at the next lower level. Conversely, if \( T_{c-jump}^N(S) \geq T_{c-ex}^N(S) \), we prefer the c-exec version for \( S \), so that \( B_T \) and \( B_E \) have a precondition. Then, for all ITE statements in \( B_T \) and \( B_E \), we only need to compare the \( T_{c-jump}^P \) and \( T_{c-ex}^P \) values in order to select the best implementation.

In each step of the top-down pass, the fastest implementation for an ITE statement \( S = \text{cond} \), \( B_T \), \( B_E \) is decided. In turn, this decision makes the presence or the absence of a precondition for \( B_T \) and \( B_E \) known. This information is exploited at the next lower level, and the process is continued down to the innermost ITE statements. The complete optimization procedure is illustrated in fig. 2.

Since in both the bottom-up and the top-down pass each ITE statement is visited only once, the total runtime of the dynamic programming technique is linear in the number of ITE statements.

### 6 Experimental results

We have evaluated the optimization technique for a TI C62xx [7], a fixed-point VLIW DSP that issues up to 8 instructions per cycle. We have compared the worst-case execution times of code generated by our technique to the code generated by the TI C6x ANSI-C compiler. For experimentation, we have compiled 10 control-intensive pieces of ANSI-C code extracted from the ADPCM transcoder "g721.c" included in the DSPStone benchmark suite [1] and from an ANSI-C MPEG package. Table 2 shows the characteristics (number of ITE statements, maximum nesting level, number of statements in the intermediate representation) of the code fragments we have tested.

<table>
<thead>
<tr>
<th>source</th>
<th>#ITE</th>
<th>nest</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>adapt_quant</td>
<td>4</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>adapt_predict1</td>
<td>3</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>adapt_predict2</td>
<td>6</td>
<td>2</td>
<td>44</td>
</tr>
<tr>
<td>defl_jmp</td>
<td>2</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>output_conv</td>
<td>4</td>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>code_adj1</td>
<td>5</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>code_adj2</td>
<td>17</td>
<td>9</td>
<td>86</td>
</tr>
<tr>
<td>code_adj3</td>
<td>17</td>
<td>5</td>
<td>95</td>
</tr>
<tr>
<td>detect_pos</td>
<td>1</td>
<td>3</td>
<td>45</td>
</tr>
<tr>
<td>find_mv</td>
<td>4</td>
<td>4</td>
<td>45</td>
</tr>
</tbody>
</table>

Table 2: Characteristics of tested C source codes
of the symbolic sequential assembly while using the same code generation techniques as the TI C6x C compiler. For the dynamic programming technique, we have set the jump penalty\(^2\) \(J\) to a value of 4, while for the parallelism parameter \(K\) (see section 4) we empirically found a value of 3 most appropriate. The code generated in this way has been compared to the code that was directly generated through the TI C compiler. The worst-case execution times (in instruction cycles) are shown in table 3.

<table>
<thead>
<tr>
<th>source</th>
<th>c-jump</th>
<th>c-exec</th>
<th>opt</th>
<th>IT</th>
</tr>
</thead>
<tbody>
<tr>
<td>adapt_quant</td>
<td>21</td>
<td>11</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>adapt_predict</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>adapt_predict2</td>
<td>26</td>
<td>21</td>
<td>22</td>
<td>27</td>
</tr>
<tr>
<td>dtrl_comp</td>
<td>9</td>
<td>12</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>outp_conv</td>
<td>26</td>
<td>30</td>
<td>24</td>
<td>21</td>
</tr>
<tr>
<td>code_adj1</td>
<td>32</td>
<td>23</td>
<td>23</td>
<td>30</td>
</tr>
<tr>
<td>code_adj2</td>
<td>57</td>
<td>17</td>
<td>49</td>
<td>31</td>
</tr>
<tr>
<td>code_adj3</td>
<td>39</td>
<td>244</td>
<td>30</td>
<td>41</td>
</tr>
<tr>
<td>detect_pos</td>
<td>28</td>
<td>27</td>
<td>27</td>
<td>29</td>
</tr>
<tr>
<td>find_mv</td>
<td>27</td>
<td>30</td>
<td>30</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 3: Experimental results: worst-case execution time

Columns 2 and 3 show the execution times when using the c-jump and c-exec schemes only, respectively. Column 4 shows the results for optimized ITE implementation, and Column 5 gives the corresponding results for the TI C compiler. In most cases, our technique was able to generate faster code. In a few cases, due to inaccuracies in the estimations, the “optimized” solutions computed by dynamic programming are worse than the pure c-jump or c-exec solutions. However, the two larger and deeply nested examples (“code_adj2” and “code_adj3”) indicate that exclusively using either c-jump or c-exec is not a good approach, but that the optimum in general is located somewhere in between.

\(^2\)The TI C62xx has 5 branch delay slots. However, the value of \(J\) was chosen smaller, since sometimes the delay slots are filled with useful instructions, resulting in a lower average jump penalty.

7 Conclusions

This paper has presented a new optimization technique for mapping control-intensive programs to embedded VLIW processors. The technique makes use of conditional instructions and aims at globally selecting the fastest ITE implementations across all nesting levels. Its efficacy has been demonstrated for a recent VLIW DSP. Since the optimization is guided by machine-dependent parameters, we expect that comparable results can be achieved for similar VLIW machines. The dynamic programming technique is not affected by the replacement of the cost estimation functions. Therefore, the simple and fast estimation techniques presented here might be substituted in the future by more accurate and time-intensive procedures that exploit schedulability information.

References