Short-Circuit Compiler Transformation: Optimizing Conditional Blocks

Mohammad Ali Ghodrat Tony Givargis Alex Nicolau

Department of Computer Sciences
Center for Embedded Computer Systems
University of California, Irvine, CA-92697
{mghodrat, givargis, nicolau}@ics.uci.edu

Abstract— We present the short-circuit code transformation technique, intended for embedded compilers. The transformation technique optimizes conditional blocks in high-level programs. Specifically, the transformation takes advantage of the fact that the Boolean value of the conditional expression, determining the true/false paths, can be statically analyzed to determine cases when one or the other of the true/false paths are guaranteed to execute. In such cases, code is generated to bypass the evaluation of the conditional expression. In instances when the bypass code is faster to evaluate than the conditional expression, a net performance gain is obtained. Our experiments with the Mediabench applications show that the short-circuit transformation yields a an average of 35.1% improvement in execution time for SPARC and an average of 36.3% improvement in execution time for ARM. We also measured an average of 36.4% reduction in power consumption for ARM.

I. Introduction

Software has become a key element in the design of embedded systems. In part, the increasing complexity and the shortening of time-to-market window force developers to rely heavily on software [16]. Given the stringent design constraints and performance requirements of embedded systems, as software becomes more dominant, the importance of aggressive compiler optimizations also increases [15]. Furthermore, unlike a traditional compiler, intended for desktop computing, it is acceptable for a compiler intended for embedded computing to take longer to execute in order to enable aggressive compiler optimizations, such as the one presented in [17].

In this paper we present a novel *short-circuit* code transformation technique to reduce execution time of conditional blocks of the following form.

```
if $C_{expr}$ then
    $S_{then}$
else
    $S_{else}$
end if
```

The short-circuit code transformation technique takes advantage of the fact that the Boolean value of the conditional expression C_{expr} can be statically analyzed to determine cases when one of S_{then} or S_{else} is to execute. Consequently, in such cases, code may be generated to bypass the evaluation of the conditional expression C_{expr}. In instances when the bypass code is faster to evaluate

```
for all x = min to max do
    for all y = min to max do
        if ($x^2 + y^2 - x^2 \times y == 0$) then
            color(x,y,BLACK);
        end if
    end for
end for
```

(a)

```
for all x = min to max do
    for all y = min to max do
        if y < 0 then
            continue;
        else
            if ($x^2 + y^2 - x^2 \times y == 0$) then
                color(x,y,BLACK);
            end if
        end if
    end for
end for
```

(b)

Fig. 1. Motivational Example

than the conditional expression C_{expr}, a net performance gain is obtained.

To illustrate, consider the problem of finding the points of collision between two surfaces. A typical approach for doing this is shown in Figure 1(a). Static analysis of the condition $C_{expr} : (x^2 + y^2 - x^2 \times y == 0)$, yields the fact that the condition C_{expr} is `false` when $y < 0$. Therefore, the above code may be transformed as shown in Figure 1(b).

The basis for the above transformation is the aggressive static analysis performed on Boolean expressions, as outlined in [10]. In Figure 2, it corresponds to the "Domain Space Partitioning" stage, where the authors propose a method for creating a `true`/`false` map of the entire domain space for any arbitrary mixed integer/Boolean expression. Here, we used their idea to propose a more complete framework for short-circuit transformation.

In the above example, and assuming $min = -max$, a net performance gain of 16% is obtained. Gain is computed using a combination of profiling and target-processor performance model, as outlined in the remainder of this paper.

This paper is organized as follows. In Section II, we review the previous related work. In Section III, we describe the compiler transformation technique in detail. In Section IV, we state some additional remarks about the transformation technique. In Section V, we show our experimental results.We conclude in Section VI.

II. Previous work

There are similarities between our proposed work and what is in compiler literature known as *lazy evaluation* of Boolean expressions [3]. This optimization is based on the observation that the value of a binary Boolean operation, composed of two operands, may be determined from the value of the first operand. For example, if

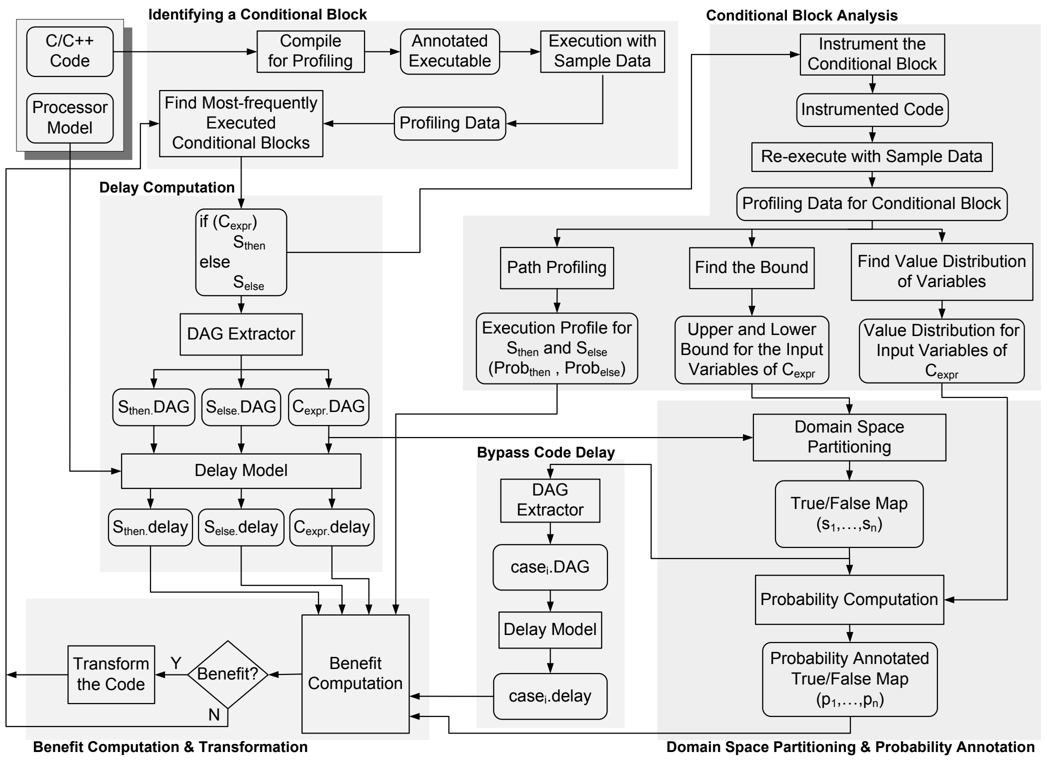

Identifying a Conditional Block
C/C++ Code
Processor Model
Compile for Profiling
Annotated Executable
Execution with Sample Data
Find Most-frequently Executed Conditional Blocks
Profiling Data
Conditional Block Analysis
Instrument the Conditional Block
Instrumented Code
Re-execute with Sample Data
Profiling Data for Conditional Block
Path Profiling
Find the Bound
Find Value Distribution of Variables
Execution Profile for S_{then} and S_{else} ($Prob_{then}$, $Prob_{else}$)
Upper and Lower Bound for the Input Variables of C_{expr}
Value Distribution for Input Variables of C_{expr}
Delay Computation
if (C_{expr}) S_{then} else S_{else}
DAG Extractor
S_{then}.DAG
S_{else}.DAG
C_{expr}.DAG
Delay Model
S_{then}.delay
S_{else}.delay
C_{expr}.delay
Bypass Code Delay
DAG Extractor
$case_i$.DAG
Delay Model
$case_i$.delay
Domain Space Partitioning
True/False Map (s_1,…,s_n)
Probability Computation
Probability Annotated True/False Map (p_1,…,p_n)
Benefit Computation
Benefit?
Y
N
Transform the Code
Benefit Computation & Transformation
Domain Space Partitioning & Probability Annotation

Fig. 2. Short Circuit Evaluation Technique

$C_{expr} = C_1$ **and** C_2, for those cases when C_1 evaluates to `false`, evaluation of C_2 can be bypassed. The work in [13] and [2] are two of the earliest work in lazy evaluation.

The work in [12] extends lazy evaluation by proposing an ordering of the operands of the conditional expression that minimizes the average execution time. Specifically, when a Boolean expression is composed of multiple sub-expressions, the probability of being `true`/`false` and the time to compute each of the sub-expressions is used to determine an optimal ordering of lazy evaluation.

The work in [6] studies the effects of lazy evaluation on code space. In their work, the authors conclude that, while the ultimate code space requirement is dependent on the target architecture and the Boolean expressions, lazy evaluation does not always result in larger code size.

We are unaware of work, other than lazy evaluation, that is similar to what is proposed in this paper. Unlike lazy evaluation, our proposed technique uses aggressive arithmetic analysis to guide the transformation of conditional blocks. Furthermore, our approach looks at the arithmetic structure in addition to Boolean structure of expressions. Finally, our approach is completely orthogonal to the sub-expression ordering proposed in [12]. In fact, the two techniques can be combined for additional average case performance gains.

In our methodology, we have used profiling data. There are several work in literature which have used profiling result for optimization. Trace scheduling [9] and other techniques derived from it [8] use profiling data for compiler optimization. In [11] the authors use profiling data for cache optimization. We comment on feasibility of profile-based optimization in Section IV.

III. Short circuit transformation

Figure 2 depicts our overall strategy. The input is a C/C++ application and the model of the processor on which the application is intended to execute. In this transformation we take a candidate conditional block (subsection A) of the form shown in Figure 3(a) and transform it to a form like the one shown in Figure 3(b).

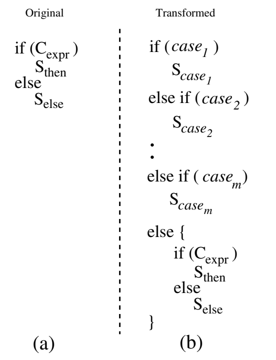

Original
Transformed
if (C_{expr}) S_{then} else S_{else}
(a)
if ($case_1$) S_{case_1} else if ($case_2$) S_{case_2} : else if ($case_m$) S_{case_m} else { if (C_{expr}) S_{then} else S_{else} }
(b)

Fig. 3. Transformation Procedure

In the transformation we find cases $case_i$ for which the Boolean value of C_{expr} is statically determined. For example, $C_{expr} : (x^2 + y^2 - x^2 \times y == 0)$ takes on the Boolean value `false` in $case_1 : y < 0$. Next, we order the cases $case_i$ based on their probability of occurrence and transform the code as shown in Figure 3(b). In the transformed code, S_{case_i} is the same as S_{then} if and only if the Boolean value of C_{expr} is `true` for $case_i$ and S_{else} otherwise.

To find the cases $case_i$ we apply the *domain space partitioning* algorithm (Section D and Section C). To find the case occurrence probabilities, we use profiling data (Section C). To determine the goodness of the transformation

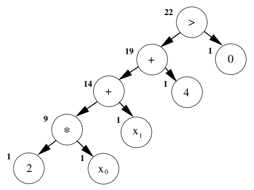


Fig. 4. DAG of $2x_0 + x_1 + 4 > 0$

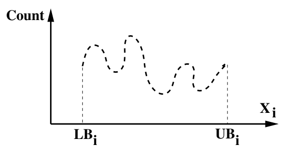


Fig. 5. Profiling Data

(Section F), we use the case occurrence probabilities (Section C), bypass code delay, and execution delay of the true/false paths (Section B). Each of these processing steps are outlined in the following text.

A. Identifying a conditional block

The application code is first compiled to generate an annotated executable. Profiling data (i.e., the number of times a conditional block is executed) is then obtained by executing the annotated executable using sample input data.

Using the profiling data, a candidate conditional block (i.e., one with a high execution count) is selected for optimization. We represent the conditional block using a triplet $< C_{expr}, S_{then}, S_{else} >$. C_{expr} is the conditional expression, S_{then} is the statement executed when C_{expr} is evaluated to `true`, and S_{else} is the statement executed when C_{expr} is evaluated to `false`.

The conditional expression C_{expr} is either a *simple condition* or a *complex condition.* A simple condition is in the form of ($expr_1$ *ROP* $expr_2$). Here, $expr_1$ and $expr_2$ are *arithmetic expressions* and *ROP* is a relational operator ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$). An arithmetic expression is formed over the language ($+$, $-$, $\times$, constant, variable). A complex condition is either a simple condition or two complex conditions merged using *logical operators* (&&, ||, !). Specifically $!C$ computes the negation of the complex condition C; (C_1&&C_2) computes logical-and of complex conditions C_1 and C_2; and ($C_1||C_2$) computes logical-or of complex conditions C_1 and C_2.

For expressing C_{expr} with variables x_1, x_2, ..., x_k, we traverse the control/data flow graph (CDFG) representing the input code, from the point where C_{expr} is used, backward. During the traversal, we substitute the subexpression $x_{i_{expr}}$ defining x_i for the existing variables x_i. We continue to replace the intermediate variable x_i until we either reach to the first definition of x_i, a conditional block or an unbounded loop where x_i is defined. An example of this, is shown in the example presented in Section E.

Each of C_{expr}, S_{then}, and S_{else} is expressed as a directed acyclic graph (DAG). For example, the DAG for the conditional expression $C : 2x_0 + x_1 + 4 > 0$ is shown in Figure 4.

B. Delay computation

We define the delay $C_{expr}.delay$, $S_{then}.delay$, and $S_{else}.delay$ to be the number of cycles necessary to compute the corresponding DAG on the target processor.

A simple methodology to compute the delay of a DAG is as follows. For a leaf node N_{leaf}, we define the delay as *one*, when N_{leaf} is an immediate/register operand or n when N_{leaf} is a memory reference. Here, n is the average processor cycles required to perform a load operation. For an internal node $N_{internal}$, we define the delay as the sum of the delays of the left and right children, plus the cost of the internal node, obtained from a processor-specific lookup table. Table I shows, in part, the delay for common DAG operations in a MIPS-like processor. As an example, the delay for the DAG shown in Figure 4 is computed to be 22 (as shown with annotations on the left of the nodes in Figure 4). We note that, when available, a more detailed delay model (e.g., one taking processor stalls and pipeline dynamics) may be used in place of the one proposed here.

TABLE I Example of Delay Values for DAG Nodes

	Integer			Float			Relational		
Operator	$+$	$-$	$\times$	$+$	$-$	$\times$	$<$	$>$	$==$
Delay (cycle)	1	1	7	4	4	7	2	2	2

C. Conditional block analysis

The input to this step is a candidate conditional block. The output is profiling data, specific to the conditional block, that is used in several future steps of the transformation. The profiling data includes: (1) the percentage of time each of the S_{then} and S_{else} execute, (2) lower and upper bounds for the variables in the conditional expression C_{expr}, and (3) the value distribution of variables in C_{expr}. For example, Figure 5 demonstrates the upper/lower bounds and the value distribution for a variable x_i in C_{expr}.

D. Domain space partitioning & probability annotation

In this step, we apply the *domain space partitioning* algorithm on C_{expr} and obtain a series of non-overlapping spaces within the domain space of C_{expr}, bounded to the upper/lower values computed during profiling. Furthermore, using the value distribution of variables in C_{expr}, we annotate each of these spaces with a probability of occurrence.

Given the conditional expression C_{expr} with variables x_1, x_2, ..., x_k, the domain space partitioning problem [10] is to partition the domain space of C_{expr} into a minimal set of k-dimensional spaces $s_1, s_2, ..., s_n$ with each space s_i having one of `true`(T), `false`(F), or `unknown`(U) Boolean value. If space s_i has a Boolean value of `true`, then C_{expr} evaluates to `true` for every point in space s_i. If space s_i has a Boolean value of `false`, then C_{expr} evaluates to `false` for every point in space s_i. If space s_i has a Boolean value of `unknown`, then C_{expr} may evaluate to `true` for some points in space s_i and `false` for others. The Boolean value for space s_i is denoted as BV_i in the remainder of this paper.

For example, consider $C_{expr} : 2 \times x_0 + x_1 + 4 > 0$. Let us assume the upper and lower bounds for x_0 and x_1 are -5 and 5 respectively. Therefore, the domain of C_{expr} is a 2-dimensional space defined by the Cartesian product $[-5, 5] \times [-5, 5]$. Figure 6 (from [10]) shows the partitioned domain space and the corresponding Boolean values.

Each of spaces $s_1, s_2, ..., s_n$ is annotated with a probability of occurrence, denoted as p_i and computed as follows:

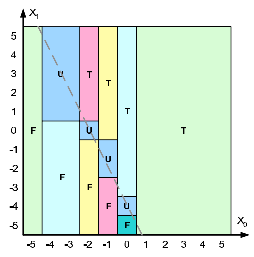


Fig. 6. Partitioned Domain of $2x_0 + x_1 + 4 > 0$

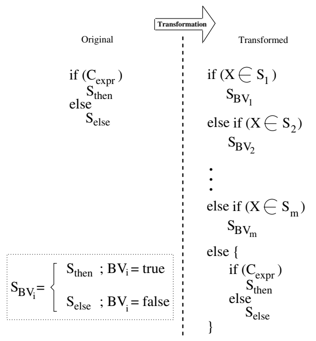
$$p_i = N_i / \sum_{i=1}^{n} N_i \quad i = 1...n \tag{1}$$

Here, N_i is the number of vectors $< x_0, ..., x_k >$ within the bounds of s_i. N_i is directly derived from the value distribution of variables in C_{expr}, gathered during the profiling.

Next, we eliminate the spaces s_i with `unknown` Boolean values and sort the remaining m ($m \leq n$) spaces according to the probabilities p_i. This new set of sorted spaces are used to emit the bypass code, as described next.

E. Transformation procedure

Figure 7 shows the structure of the transformed code. The transformed structure is made of two sections. The first section is a sequence of cases, in the form of `if`, `else if`, ..., `else if`. There are exactly m such cases, each corresponding with one of the m spaces. Furthermore, these cases appear in the transformed code in the order of decreasing probability of occurrence. The second section is the original conditional block embraced within the final `else`.



Transformation
Original
Transformed
if (C_{expr}) S_{then} else S_{else}
if ($X \in S_1$) S_{BV_1}
else if ($X \in S_2$) S_{BV_2}
else if ($X \in S_m$) S_{BV_m}
else { if (C_{expr}) S_{then} else S_{else} }
$S_{BV_i} = \begin{cases} S_{then} ; BV_i = true \\ S_{else} ; BV_i = false \end{cases}$

Fig. 7. Transformation Procedure

Each bypass $case_i$ has a conditional expression in the form of $X \in s_i$. X is a vector of length k, in the form of $< x_1, x_2, ..., x_k >$, where x_i's are the variables in C_{expr}. The expression $X \in s_i$ is an abbreviation for a conditional statement of the following form: $(lb_1 \leq x_1 \leq ub_1)$ && $(lb_2 \leq x_2 \leq ub_2)$...&& $(lb_k \leq x_k \leq ub_k)$, where lb_j/ub_j define the boundary of s_i along the x_j dimension. Let us denote this condition in its DAG form and as $case_i.DAG$. We can now compute the delay of $case_i.DAG$ (Section B) and denote it as $case_i.DAG.delay$.

Each bypass $case_i$ has a conditional statement S_{BV_i}, which is either S_{then} when BV_i is `true` and S_{else} otherwise. Accordingly, we denote the execution time of S_{BV_i} as $S_{BV_i}.delay$:

$$S_{BV_i}.delay = \begin{cases} S_{then}.delay; & BV_i = true \\ S_{else}.delay; & BV_i = false \end{cases} \tag{2}$$

As an example of the transformation procedure, Figure 8 shows a code segment from the MP3 encoder [18]. Here, the conditional block within the nested loops requires the evaluation of the costly expression C_{expr} : $15.8 + 7.5 \times t_1[i][j] - 17.5 \times \sqrt{1.0 + t_1[i][j] \times t_1[i][j]} \leq -100$.

```
for(j=0;j<CBANDS;j++){
    for(i=0;i<CBANDS;i++){
        t1[i][j] += 0.474;
        t3 = 15.811389+7.5*t1[i][j]–17.5*sqrt((double) (1.0+t1[i][j]*t1[i][j]));
        if(t3 <= –100) {
            s[i][j] = 0;
        }
        else {
            t3 = (t2[i][j] + t3)*LN_TO_LOG10;
            s[i][j] = exp(t3);
        }
    }
}
```

Fig. 8. Sample MP3 Code Segment

Based on static analysis of C_{expr} and profiling we obtain the `true`/`false` map shown in Table II. Table II shows that C_{expr} evaluates to `false` whenever $-4.7 < t_1 < 11.862$. Here, 47.6% of the time the inequality C_{expr} : $15.8 + 7.5 \times t_1[i][j] - 17.5 \times \sqrt{1.0 + t_1[i][j] \times t_1[i][j]} \leq -100$ evaluates to `true` during actual execution of the code. From the above analysis, we emit the transformed code as shown in Figure 9.

TABLE II `TRUE`/`FALSE` MAP OF C_{expr} IN MP3

Space	Boolean Value (BV)	Probability
[-4.7,11.862]	false	0.475939
[-30,-4.702]	true	0.312169
[11.864,30]	true	0.160242

```
for(j=0;j<CBANDS;j++){
    for(i=0;i<CBANDS;i++){
        if ( t1[i][j]<= 11.862 && t1[i][j]>= –4.700)
        {
            t1[i][j] += 0.474;
            t3 = 15.811389+7.5*t1[i][j]–17.5*sqrt((double) (1.0+t1[i][j]*t1[i][j]));
            t3 = (t2[i][j] + t3)*LN_TO_LOG10;
            s[i][j] = exp(t3);
        }
        else if ( t1[i][j]<= –4.702 && t1[i][j]>=–30 )
            s[i][j] = 0;
        else if ( t1[i][j]>=11.864 && t1[i][j]<=30 )
            s[i][j] = 0;
        else
        {
            t1[i][j] += 0.474;
            t3 = 15.811389+7.5*t1[i][j]–17.5*sqrt((double) (1.0+t1[i][j]*t1[i][j]));
            if(t3 <= –100)
                s[i][j] = 0;
            else {
                t3 = (t2[i][j] + t3)*LN_TO_LOG10;
                s[i][j] = exp(t3);
            }
        }
    }
}
```

Fig. 9. Transformed MP3 Code

F. Computing the benefit of transformation

The short-circuit transformation is beneficial when:

$$T_{new} < T_{original} \tag{3}$$

In Equation 3, $T_{original}$ is the estimated time to execute the original conditional block and T_{new} is the estimated time to execute the transformed conditional block. $T_{original}$ can be computed using the $C_{expr}.delay$,

$S_{then}.delay$ and $S_{else}.delay$, along with the probability of execution of each of the `true`/`false` paths, obtained from profiling, namely:

$$T_{original} = C_{expr}.delay + Prob_{then} \times S_{then}.delay + Prob_{else} \times S_{else}.delay \quad (4)$$

T_{new} is calculated as follows:

$$\begin{aligned} T_{new} =& p_1 \times (case_1.delay + S_{BV_1}.delay) \\ &+ p_2 \times (case_1.delay + case_2.delay + S_{BV_2}.delay) \\ &+ \dots \\ &+ p_m \times (case_1.delay + \dots + case_m.delay + S_{BV_m}.delay) \\ &+ (1 - p_1 - \dots - p_m) \times T_{original} \end{aligned} \quad (5)$$

Or, in abbreviated format as:

$$T_{new} = \sum_{i=1}^{m} [p_i \times (\sum_{j=1}^{i} case_j.delay + S_{BV_i}.delay)] + (1 - \sum_{i=1}^{m} p_i) \times T_{original} \quad (6)$$

T_{new} is calculated by taking into account the delay of each of the cases that have been added to the transformed conditional block structure. For the first case ($case_1$) the execution time is dependent on the probability that $case_1$ will be selected during execution (p_1), the cost of evaluating the bypass condition ($case_1.delay$), and the cost of executing the statements within $case_1$ (S_{BV_1}). For subsequent cases ($case_i$), the delay is dependent on the number of previous bypass condition evaluations ($\sum_{j=1}^{i} case_j.delay$), the probability that $case_i$ will be selected during execution (p_i), the cost of evaluating the bypass condition ($case_i.delay$), and the cost of executing the statements within $case_i$ (S_{BV_i}).

G. Computing the code size increase

For an embedded system, the code footprint may be a constraint. Hence, we give estimate of code size before and after the proposed transformation. The estimated code size (in number of instruction) for the original code segment in Figure 7 is computed as follows. If $Size_{C_{expr}}$ is the code size for computing C_{expr}, $Size_{then}$ and $Size_{else}$ are the code size for computing the statements inside the associated conditional block and for implementing the conditional block we require one comparison and two branch instructions, then:

$$Size_{original} = Size_{C_{expr}} + Size_{then} + Size_{else} + 3$$

The estimated code size for the transformed code segment in Figure 7 is computed as follows:

$$Size_{new} = m \times Size_{case} + Size_{then/else} + Size_{original} + m + 1$$

where $Size_{case}$ is the size of the code segment needed to compute each of $X \in S_i$ in Figure 7. $Size_{then/else}$ is the total size of the code added to all the branches. $m+1$ is added to this summation because there are $m+1$ total branches for each case.

For computing $Size_{case}$, we calculate the size of code required to compute $(lb_1 \leq x_1 \leq ub_1)$ && $(lb_2 \leq x_2 \leq ub_2)$...&& $(lb_k \leq x_k \leq ub_k)$ as mentioned in Section E. Specifically, if k is the number of variables in C_{expr} and each of the $(lb_i \leq x_i \leq ub_i)$ requires two comparisons and two branches, this will be computed as:

$$Size_{case} = k \times 4$$

$Size_{then/else}$ is incremented by $Size_{then}$ for each $case_i$ (where C_{expr} is `true`), and by $Size_{else}$ for each $case_i$ (where C_{expr} is `false`). So:

$$Size_{then/else} = \sum_{i=1}^{m} BV_i \times Size_{then} + (1 - BV_i) \times Size_{else}$$

Where BV_i is defined in Section D.

IV. Additional remarks

There are a number of issues worthy of discussion regarding short-circuit evaluation:

1. The static benefit computation, described in the previous section, assumes that the conditional expression C_{expr} of the candidate conditional block does not share a subexpression with the statements S_{then} or S_{else}. Otherwise, if C_{expr} shares a subexpression with one of S_{then} or S_{else}, or both, then the performance gains of the transformed code may be less substantial. For example, Figure 8 shows an instance where parts of the conditional expression (t_3) is used in S_{else}.

2. Any or all of the cases ($case_1$, $case_2$, ..., $case_m$) may be left out in the transformed code without affecting the correctness. This is because the original conditional block is always present in the last section of the transformed code, i.e., the final `else`. This fact can be exploited to obtain more optimal transformations. For instance, one or more of the cases may be left out in order to improve the execution time. Specifically, to address the subexpression issue mentioned above, one can eliminate the cases that use a subexpression of C_{expr}. Figure 10 shows how the first case of Figure 9 is eliminated to obtain an optimized solution.

```
for(j=0;j<CBANDS;j++){
   for(i=0;i<CBANDS;i++){
      if ( t1[i][j]<= −4.702 && t1[i][j]>=−30 )
            s[i][j] = 0;
      else if ( t1[i][j]>=11.864 && t1[i][j]<=30 )
            s[i][j] = 0;
      else
      {
            t1[i][j] += 0.474;
            t3 = 15.811389+7.5*t1[i][j]−17.5*sqrt((double) (1.0+t1[i][j]*t1[i][j]));
            if(t3 <= −100)
                  s[i][j] = 0;
            else {
                  t3 = (t2[i][j] + t3)*LN_TO_LOG10;
                  s[i][j] = exp(t3);
            }
      }
   }
}
```

Fig. 10. Transformed & Optimized MP3 Code

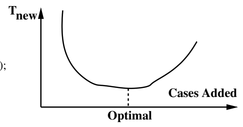


Fig. 11. Transformed Code Behavior

3. At some point, as the number of cases that are added to the transformed code structure grows, T_{new} increases (Equation 6 and shown in Figure 11). One method to find the minimum is by adding the next highest probability case to the transformed structure as long as T_{new} is strictly decreasing.

4. The approach presented here can apply to nested conditional blocks by assuming that the inner conditional block is a single statement appropriately included in S_{then} or S_{else}. Similarly, the approach presented here can apply to conditional blocks in the form of `if`, `else if`, ..., `else if`, `else`, by pre-transforming them into a `if/else` form with nested conditional blocks.

5. We note that profile-based optimizations (even if it yields slowdowns) have been proven useful in many other contexts in computer science, notably in Trace Scheduling [9] and its derivatives (e.g., Superblock and Hyperblock) [8] as well as other techniques (e.g., cache optimization [11]). And the practice has proven itself useful over the last two decades.

   While input data may indeed change somewhat the results, branches are typically highly biased in most programs and thus any reasonable profiling will pick that bias up, enabling our technique to exploit it.

6. The proposed transformation yields correct results regardless of the input data. The generated code yields the most speedup if the input data has similar characteristics of the profiled data. It is theoretically possible for some input data to cause a small slow-down. Such a slowdown will happen when all the by-pass cases are evaluated to false (see Figure 7). This scenario did not present itself in our experiments.

7. The approach presented here can apply generally to any application, but it takes time to analyze the code and generates the optimized code. For an embedded software this increase in compile time is justified which might not be the case generally.

8. As with any other compiler optimization ([1] and [7]), the optimization presented in this paper applies to some regions of the code, namely conditional blocks. In embedded software, conditional blocks are common, making the proposed approach practical.

V. Experiments

To evaluate the proposed code transformation technique, several *code segments (kernels)* from MediaBench [5] application suite were chosen. We also experimented with an MP3 encoder implementation obtained from [18] as well as a collision detection algorithm chosen from computer graphics domain. By *code segment*, we mean the region of code that was impacted by the transformation. For example, if the transformed code was a conditional block within a for-loop, then the time taken to execute that entire for-loop before and after the optimization was used to determine the speedup. To be more precise, the *code segment* will always be the smallest set of basic blocks, touched by our algorithm, with a single entry and multiple exists. The characteristics of the code segments selected for our experiments are listed in Table III. In Table IV *Conditional expressions* column shows the particular conditional expression(s) in the code segment selected for optimization. If there are more than one conditional expression in a code segment, then we run our algorithm for each instance of conditional expression separately (i.e., the algorithm is run iteratively as long as improvements are obtained). Also, in Table IV, *Application* shows where we picked the code segment and *Function description* shows the functionality of the code.

TABLE III Selected Application List

Code seg.	Application-Function desc.	Conditional expressions
1	MESA-Compute the fogged color indexes	$exp(c^2 * z^2) > 1$
2	MESA-Compute the fogged color	$0 \leq exp(-c^2 * z^2) \leq 1$
3	MP3-Layer 3 Psych. Analysis	$15.8 + 7.5 * t - 17.5 * \sqrt{(1.0 + t^2)} \leq -60$
4	MP3-Psych. Analysis	$15.8 + 7.5 * t1 - 17.5 * \sqrt{(1.0 + t1^2)} < -100$
5	Graphics-Check for collision	$x * x + y * y - x * x * y == 0$
6	MPEGDEC Initialize Decoder	$(i < 0), (i > 255)$
7	MPEGENC-Ver./Hor. Filter,2:1 Subsample	$(i < 5), (i < 4), (i < 3), (i < 2), (i < 1)$
8	MP3-Layer 3 Psych. Analysis	$j < sync\_flush, j < BLKSIZE$
9	MP3-Read and align audio data	$j < 64$
10	MPEG-IDCT Initialize	$(i < -256), (i > 255)$
11	MPEGDEC-Ver./Hor. Interpolation Filter	$(i < 2), (i < 1)$

We applied our transformation technique at the source level to each of the chosen benchmarks, compiled the original and the transformed code, and measured the improvement. We did this experiment for two types of CPU: SPARC and ARM. For SPARC we measured the performance improvement together with code size increase. For ARM, we measured improvement on performance and power.

A. SPARC

The results of experiments on SPARC CPU are summarized in Table IV. In Table IV, $T_{original}$ and T_{new} columns show the execution time for the selected code segment before and after the proposed transformation, as reported by the *clock*() function of Unix. *Speedup*(%) shows the execution time improvement in each case. The two columns *Original size* and *New size* show the size of the selected code segment before and after transformation in number of assembly instructions. The *code size increase* shows the percentage increase in code size. As can be seen in Table IV the first two examples have a negative increase. This is due to total removal of the C_{expr} in both of the cases.

The experiments were run on a Sun workstation, with 2 SPARC CPUs (1503 MHz SUNW,UltraSPARC-IIIi) and 2 GB of memory. We used GCC compiler version 3.4.1 (with no optimization switch) in order to generate executables.

In the best case, we observed application speedup of

TABLE IV Result of Experiments for SPARC

Code seg.	$T_{original}$ (μS)	T_{new} (μS)	Speedup (%)	Original size	New size	Code size increase(%)
1	76	9	88.15	196	100	-48
2	669	569	14.95	173	124	-28
3	707	410	42.00	237	258	8
4	603	234	61.19	262	367	40
5	658	552	16.03	145	161	11
6	220	170	22.72	86	124	44
7	869	534	38.55	302	782	158
8	412	371	09.95	204	351	72
9	1007	919	08.73	106	130	22
10	24	17	29.16	87	126	44
11	135	61	54.81	164	288	75

88.15%. On average, we observed application speedup of 35.1%. These speedup calculations are based on the ratio of the time to execute the optimized code segment to the time to execute the original code segment. On average 35.1% increase in code size was measured.

B. ARM

The results of experiments for ARM, a popular embedded processor, are summarized in Table V using SimpleScalar/ARM [4] toolset. In Table V, $T_{original}$ and T_{new} columns show the execution time for the selected code segment before and after the proposed transformation, as reported by the cycle-accurate simulator of SimpleScalar/ARM toolset (*sim-outorder*). *Speedup*(%) shows the execution time improvement in each case. The two columns *Power original* and *Power transformed* show the result of power consumption before and after transformation as reported by SimpleScalar/ARM power modeling tool (Sim-Panalyzer) [14]. The *Power reduction* column shows the percentage of power consumption reduction. On average we saw 36.3% speedup for running the code segments on ARM CPU and 36.4% reduction on power consumption.

TABLE V Result of Experiments for ARM

Code seg.	$T_{orig.}$ ($\#cycles$) ($\times 10^3$)	T_{new} ($\#cycles$) ($\times 10^3$)	Speedup (%)	Power orig. ($\times 10^3$)	Power trans. ($\times 10^3$)	Power reduc. (%)
1	8497	572	93	34008	2193	93
2	7876	7757	01	31982	32263	-08
3	966593	382278	60	3221814	1295703	59
4	2490140	188417	92	8062563	595235	92
5	76039	50601	33	274522	204053	25
6	8478	7229	14	30727	24701	19
7	39874	35194	11	124536	117814	05
8	496679	327590	34	2199263	1349631	38
9	12056	11559	4.1	46855	43813	06
10	8574	7230	15	33188	25463	23
11	617702	347367	43	3004423	1527731	49

VI. Conclusion

We have presented the short-circuit code transformation technique, intended for embedded compilers. The transformation technique optimizes conditional blocks in high-level programs. Specifically, the transformation takes advantage of the fact that the Boolean value of the conditional expression, determining the true/false paths, can be statically analyzed to determine cases when one or the other of the true/false paths are guaranteed to execute. In such cases, code is generated to bypass the evaluation of the conditional expression. When the bypass code is faster to evaluate than the conditional expression, a net performance gain is obtained. We are currently considering applying a similar technique to loop optimization. Specifically, we recognize that the condition that is evaluated within the loop control structure can be analyzed, similar to the work shown here, in order to generate a more optimal looping mechanism.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers principles, techniques and tools.* Addison Wesley, 1988.

[2] B.W. Arden, B.A. Galler, and R.M. Graham. An algorithm for translating boolean expressions. *Journal of the ACM*, 9(2):222–239, 1962.

[3] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[4] D.C. Burger and T. M. Austin. The simplescalar tool set, v2.0. *Computer Architecture News*, 25(3):13–25, 1997.

[5] M. Potkonjak C. Lee and W.H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. of Micro-30*, 1997.

[6] M.H. Clifton. A comparison of space requirements for short-circuit and full evaluation of boolean expressions. In *Proc. of ACMSE*, 1998.

[7] K. Cooper and L. Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2004.

[8] P. Faraboschi, J.A. Fisher, and C. Young. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, 89:1638–1659, 2001.

[9] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transaction on Computers*, C-30(7):478–490, 1981.

[10] M.A. Ghodrat, T. Givargis, and A. Nicolau. Equivalence checking of arithmetic expressions using fast evaluation. In *Proc. of CASES*, 2005.

[11] A. Ghosh and T. Givargis. Cache optimization for embedded processor cores: An analytical approach. *ACM TODAES*, 9(4):419–440, 2004.

[12] M.Z. Hanani. An optimal evaluation of boolean expressions in an online query system. *Communications of the ACM*, 20(5):344–347, 1977.

[13] H.D. Huskey and W.H. Wattenburg. Compiling techniques for boolean expressions and conditional statements in algol 60. *Communications of the ACM*, 4(1):70–75, 1961.

[14] N. Kim, T. Kgil, V. Bertacco, T. Austin, and T. Mudge. Microarchitectural power modeling techniques for deep sub-micron microprocessors. In *Proc. of ISLPED*, 2004.

[15] R. Leupers. Code generation for embedded processors. In *Proc. of ISSS*, 2000.

[16] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test*, 18(6):23–33, 2001.

[17] M. Wolfe. How compilers and tools differ for embedded systems. In *Proc. of CASES*, 2005.

[18] www.mp3tech.org. Iso mp3 sources (distribution 10). Available as http://www.mp3-tech.org/programmer /sources/dist10.tgz.