Control Flow Optimization in Loops using Interval Analysis

Mohammad Ali Ghodrat  
School of Computer Sciences  
University of California  
Irvine, CA  
mghodrat@ics.uci.edu

Tony Givargis  
School of Computer Sciences  
University of California  
Irvine, CA  
givargis@ics.uci.edu

Alex Nicolau  
School of Computer Sciences  
University of California  
Irvine, CA  
nicolau@ics.uci.edu

ABSTRACT

We present a novel loop transformation technique, particularly well suited for optimizing embedded compilers, where an increase in compilation time is acceptable in exchange for significant performance increase. The transformation technique optimizes loops containing nested conditional blocks. 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 using a novel interval analysis technique that can evaluate conditional expressions in the general polynomial form. Results from interval analysis combined with loop dependency information is used to partition the iteration space of the nested loop. In such cases, the loop nest is decomposed such as to eliminate the conditional test, thus substantially reducing the execution time. Our technique completely eliminates the conditional from the loops (unlike previous techniques) thus further facilitating the application of other optimizations and improving the overall speedup. Applying the proposed transformation technique on loop kernels taken from Mediabench, SPEC-2000, mpeg4, qsdpcm and gimp, on average we measured a 175% (1.75X) improvement of execution time when running on a SPARC processor, a 336% (4.36X) improvement of execution time when running on an Intel Core Duo processor and a 198.9% (2.98X) improvement of execution time when running on a PowerPC G5 processor.

Categories and Subject Descriptors

D.3.4 [Processors]: Compilers; I.1 [Computing Methodologies]: Symbolic and Algebraic Manipulation

General Terms

Algorithms

Keywords

Interval analysis, Compiler Loop Optimization, Algorithmic Code Transformation

1. INTRODUCTION

Aggressive compiler optimization, in particular those that address loops can significantly improve the performance of the software, thus justifying the additional compilation time requirements. This is in particular true in the embedded system domain where software has become a key element of the design process and performance is of a critical concern. Furthermore, unlike a traditional compiler, intended for desktop computing, it is acceptable for a compiler intended for embedded computing to take longer to compile but perform aggressive optimizations, such as the ones presented in [13]. In our case, the additional compiler execution time was of the order of 10-msec per loop [4].

In contrast to existing work on loop transformation, we present an algorithmic loop transformation technique that substantially restructures the loop using knowledge about the control flow combined with data-dependence information within the body of the loop. The control flow and data-dependences within the loop body is analyzed using a static interval analysis technique previously outlined in [4]. Interval analysis provides information on the true/false paths within the original loop body as a function of the loop indices. The analysis of the loop iteration dependencies is used to establish the possible space of loop restructuring. Combining these two static analysis results, an algorithm is provided that fully partitions the original iteration space (i.e., original loop) into multiple disjoint iteration spaces (i.e., generated loops). The bodies of these generated loops are void of conditional branches and thus (unlike previous techniques which leave branches in loops) our techniques allows for more effective optimizations. Moreover, each of these loops, and the ordering within them, are consistent with the original loop iteration dependencies.

As an example consider the loop kernel shown below. This loop kernel is taken from gimp benchmark [11].

```c
#define STEPS 64  
#define KERNEL_WIDTH 3  
#define KERNEL_HEIGHT 3  
#define SUBSAMPLE 4  
#define THRESHOLD 0.25

for (yj = 0; yj <= SUBSAMPLE; yj++) {  
y = (double) yj / (double) SUBSAMPLE;  
for (xi = 0; xi <= SUBSAMPLE; xi++) {  
x = (double) xi / (double) SUBSAMPLE;  
x += 1.0; y += 1.0;  
for (j = 0; j < STEPS * KERNEL_HEIGHT; j++) {  
dist_y = y - (((double)j + 0.5) / (double)STEPS);  
dist_x = x - (((double)i + 0.5) / (double)STEPS);  
if (dist_y * dist_y < THRESHOLD) w = 1.0;  
else  
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CfES’08, October 19–24, 2008, Atlanta, Georgia, USA.  
Copyright 2008 ACM 978-1-60558-469-0/08/10 ...$5.00.
w = 0.0;

for (i = 0; i < STEPS * KERNEL_WIDTH; i++) {
    dist_x = x - (((double) i + 0.5) / (double) STEPS);
    for (j = 0; j < STEPS * KERNEL_HEIGHT; j++) {
        dist_y = y - (((double) j + 0.5) / (double) STEPS);
        if ((SQR (dist_x) + SQR (dist_y)) < THRESHOLD)
            w = 1.0;
        else
            w = 0.0;
        value[i / STEPS][j / STEPS] += w;
    }
}

In the transformed code, the evaluation of the conditional expression for part of the most inner loop (i.e., the loop with i as the index variable) is eliminated. Applying our optimization to the rest of the loop kernel, while using the entire information in Table 1, we obtain 16% speed-up on SPARC, 21% on Intel Core Duo and 24% on PowerPC G5 as shown in Section 5.

The rest of this paper is organized as follows. In Section 2, we outline the related work. In Section 3, we formulate the problem, show the overall flow of the proposed transformation and establish some preliminaries. In Section 4, we establish the transformation technique. In Section 5, we show our experimental results. In Section 7, we conclude.

2. PREVIOUS WORK

There are many transformation techniques targeting nested loops. Since our work specifically applies to control flow optimization of loops we primarily focus on related work that target control flow optimization. Of course, data-flow level optimizations can be combined with control flow
optimizations to further improve the generated code (i.e., data-flow optimizations may benefit from simpler control flow within loops).

Table 2 provides a set of properties that are used to compare and contrast loop optimization strategies using control flow analysis. Furthermore, Table 3 summarizes existing loop transformation techniques and provides an analysis of their strength relative to the presented work.

Among all the techniques listed in Table 3, the three most relevant ones are loop unswitching, index-set splitting and loop nest splitting.

Loop unswitching [8], has similarities to our transformation in targeting conditional blocks within loops. Specifically, loop unswiching attempts to replicate the loop inside each branch of the conditional. In contrast, our technique attempts to completely eliminate the conditional block within a loop by decomposing a loop into multiple independent loops. In loop unswitching technique, the conditional expression does not depend on loop indices, hence limiting its applicability to loops containing trivial conditions, but in our technique the conditional expression is a function of loop indices.

Another technique, index-set splitting [12], does a similar transformation but in a much limited way than our method. First index-set splitting only considers affine expressions and there is no discussion on how to handle cases where there are dependences between loop iterations. In our method we consider non-affine conditional expressions within the loop and handle cases where there are dependences between loop iterations and, when dependences allow, we eliminate the conditions from the loops.

A closely related work in control flow loop optimization is suggested by Falk et al. [3]. The loop model used in their work differs from ours. First, they consider conditional expressions that are strictly affine (vs. arbitrary polynomial in our case) functions of the loop indices. Figure 1-a shows a case in gimp [11] benchmark which is optimized by our technique but not by their method. Second, Falk’s loop model assumes that the conditional expression is strictly a function of loop indices, but in our loop model the conditional expression can include other variables computed within the loop body. Figure 1-b shows a case in mp3 benchmark [14] that can be optimized by our technique but not by their method (here the transformed code is not shown to save space). The final important difference between our work and Falk’s is that in our transformed code the conditional block is completely eliminated while in their work it is simplified or hoisted to a higher point in the nested loops, but not eliminated. To show this difference clearly, let’s first consider a synthetic example shown in Figure 1-c. Figure 1-c shows a case in which our technique (Figure 1-e) has removed the condition completely resulting in significant (30% on SPARC and 68% on Intel) speedup while their technique (Figure 1-d) has only partially eliminated the evaluation of the conditional expression. A similar example 186.crafty from SPEC-2000 [1] is shown in Figure 1-f where applying the technique in [3] will not remove the conditions completely.

Our proposed transformation targets loops that follow the normalized template shown in Figure 2-a. Here, there are n loop nests, with n indices x₁, ..., xₙ. For every index xᵢ, the value for lower (upper) bounds lbᵢ (ubᵢ) is assumed to be statically computable signed integer constants. When unknown bounds exists, an estimate (possibly profile-based) can be used without affecting the correctness of the transformed code. In particular, the closer the estimated bounds to the actual, the higher the efficiency of the transformation. The body of the inner most loop contains at least one conditional block, called the target conditional block.

A large number of arbitrary loop structures can be rewritten in the normalized form of Figure 2-a [8]. Here, slcond_expr computes the value of the branch condition v.

3. PROPOSED TRANSFORMATION

The proposed transformation decomposes the original

<table>
<thead>
<tr>
<th>Optimization technique</th>
<th>Property(Table 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop unswitching</td>
<td>✓</td>
</tr>
<tr>
<td>Loop interchanging</td>
<td>✓</td>
</tr>
<tr>
<td>Loop skewing</td>
<td>✓</td>
</tr>
<tr>
<td>Strip-mining</td>
<td>✓</td>
</tr>
<tr>
<td>Loop collapsing</td>
<td>✓</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>✓</td>
</tr>
<tr>
<td>Loop unswitching</td>
<td>✓</td>
</tr>
<tr>
<td>Loop peeling</td>
<td>✓</td>
</tr>
<tr>
<td>Loop tiling</td>
<td>✓</td>
</tr>
<tr>
<td>Loop nest splitting</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 3: Comparison with other loop optimization techniques

<table>
<thead>
<tr>
<th>Optimization technique</th>
<th>Property(Table 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop unswitching</td>
<td>✓</td>
</tr>
<tr>
<td>Loop interchanging</td>
<td>✓</td>
</tr>
<tr>
<td>Loop skewing</td>
<td>✓</td>
</tr>
<tr>
<td>Strip-mining</td>
<td>✓</td>
</tr>
<tr>
<td>Loop collapsing</td>
<td>✓</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>✓</td>
</tr>
<tr>
<td>Loop unswitching</td>
<td>✓</td>
</tr>
<tr>
<td>Loop peeling</td>
<td>✓</td>
</tr>
<tr>
<td>Loop tiling</td>
<td>✓</td>
</tr>
<tr>
<td>Loop nest splitting</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2: Properties which are being compared in Table 3

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property 1</td>
<td>Optimize control flow of a loop with nested conditional block</td>
</tr>
<tr>
<td>Property 2</td>
<td>Minimize dependence analysis needed</td>
</tr>
<tr>
<td>Property 3</td>
<td>Conditional expression depends on loop index</td>
</tr>
<tr>
<td>Property 4</td>
<td>Conditional expression is not affine function of loop variables</td>
</tr>
<tr>
<td>Property 5</td>
<td>Conditional expression contains logical operators</td>
</tr>
<tr>
<td>Property 6</td>
<td>Conditional expression is a function of loop indices and non-loop-index variables</td>
</tr>
<tr>
<td>Property 7</td>
<td>Conditional expression has a general polynomial form</td>
</tr>
<tr>
<td>Property 8</td>
<td>Conditional expression will be removed completely from loop body of the transformed code</td>
</tr>
</tbody>
</table>
nested loops of Figure 2-a into three parts, as shown in Figure 2-b. The first part sets up one or more nested loop structures with iteration spaces for which the \( sl_{\text{cond.expr}} \) is known to be \texttt{true} at compile time. Likewise, the second part sets up one or more nested loop structures with iteration spaces for which the \( sl_{\text{cond.expr}} \) is known to be \texttt{false} at compile time. The third part sets up one or more nested loop structures with an iteration space for which the \( sl_{\text{cond.expr}} \) can not be statically evaluated. The three parts combined cover the entire iteration space of the original nested loops. Since the evaluation of \( sl_{\text{cond.expr}} \) is eliminated in parts one and two, the decomposed code executes substantially fewer instructions than the original code.

### 3.1 Preliminaries

In this subsection we summarize the analysis technique developed in [4] and used for our transformation. Without loss of generality, the remaining discussions in the paper will use C/C++ notation. Every program can be represented as a Control Data Flow Graph (CDFG) intermediate form. A CDFG is a graph that shows both data and control flow in a program. The nodes in a CDFG are basic blocks. Each basic block contains straight lines of statements with no branch except for the last statement and no branch destination except for the first statement. The edges in a CDFG represent the control flow in the program.

As defined in [4], a conditional expression \( \text{cond.expr} \) is either a simple condition or a complex condition. A simple condition is in the form of \( \text{expr}_1 \text{ROP} \text{expr}_2 \). Here, \( \text{expr}_1 \) and \( \text{expr}_2 \) are arithmetic expressions and \( \text{ROP} \) is a relational operator \( (=, \neq, <, \leq, >, \geq) \). An arithmetic expression is formed over the language \( (+, -, \times, \text{constant}, \text{variable}) \). A complex condition is either a simple condition or two complex conditions merged using logical operators \( (\&\&\!, ||\!, !) \).

An integer interval of the form \([a, b]\) represents all possible integer values in the range \(a\) to \(b\), inclusively. The operations \((+, -, \times, /)\) can be defined on two intervals \([a, b]\) and \([c, d]\). We refer the interested reader to [7] for a full coverage of interval arithmetic.

We define an \(n\)-dimensional space to be a box-shaped region defined by the Cartesian product \([l_0, u_0] \times [l_1, u_1] \times ... \times [l_n, u_n]\). Hence, for a given program with \(n\) input integer-variables \(x_0, x_1, ..., x_{n-1}\), the program domain space is an \(n\)-dimensional space defined by the Cartesian product \([l_0, u_0] \times [l_1, u_1] \times ... \times [l_n, u_n]\), where \(l_0\) and \(u_0\) are defined based on the type of the variable \(x_i\) (e.g., if \(x_i\) is of type signed character then \(l_0 = -128\) and \(u_0 = 127\)).

Given the conditional expression \(\text{cond.expr}\) with variables \(x_1, x_2, ..., x_k\), the domain space partitioning problem [4] is to partition the domain space of \(\text{cond.expr}\) into a minimal set of \(k\)-dimensional spaces \(s_1, s_2, ..., s_n\) with each space \(s_i\) having one of \texttt{true(T)}, \texttt{false(F)}, or \texttt{unknown(U)} Boolean value.

If space \(s_i\) has a Boolean value of \texttt{true}, then \(\text{cond.expr}\) evaluates to \texttt{true} for every point in space \(s_i\). If space \(s_i\) has a Boolean value of \texttt{false}, then \(\text{cond.expr}\) evaluates to \texttt{true} for every point in space \(s_i\). If space \(s_i\) has a Boolean value of \texttt{unknown}, then \(\text{cond.expr}\) may evaluate to \texttt{true} for some points in space \(s_i\) and \texttt{false} for others.

For example, consider \(\text{cond.expr} : 2 \times x_0 + x_1 + 4 > 0\) (domain space \([-5, 5] \times [-5, 5]\)). Figure 3 shows the partitioned domain space and the corresponding Boolean values [4].

4. TECHNICAL APPROACH

We now begin to describe the technique proposed in this paper. A candidate loop \(L\) has the structure shown in Figure 2-a. The iteration space of \(L\) is defined as \([lb_1, ub_1] \times [lb_2, ub_2] \times ... \times [lb_n, ub_n]\). The body of \(L\) can be decomposed into the CDFGs corresponding to \(sl_{\text{cond.expr}}\), \(sl_{\text{then}}\), and \(sl_{\text{else}}\). The variable \(v\), computed by \(sl_{\text{cond.expr}}\), is defined in terms of the loop variables \(x_1, x_2, ..., x_n\) and all other variables which are alive when computing the value of \(v\). The transformation technique consists of a number of steps, specifically:

- Compute the interval set of \(v\) by processing the CDFG corresponding to \(sl_{\text{cond.expr}}\) (Section 4.1).
- Compute the dependence vector of iteration space (Section 4.2).
- Partition the iteration space (Section 4.3).
- Generate code (Section 4.4).

4.1 Interval Set Computation

In the following discussion, the code segment presented in Table 4 is used to demonstrate the interval set computation. In Table 4, loop variables \(x_1\) and \(x_2\) are assumed to be live on entry (i.e., inputs to the \(sl_{\text{cond.expr}}\) CDFG) and Boolean variable \(v\) is assumed to be live on exit (i.e., output of the \(sl_{\text{cond.expr}}\) CDFG). We refer the reader to Section 3.1 for a
review of integer intervals, spaces and program domain space used here.

At any given point in the CDFG, a variable \( v \) has an interval, defining the range of possible values it may have. At the point of declaration, the type of a variable \( v \) gives the upper and lower bounds of such an interval (e.g., line 1 of Table 4). Along each path in the CDFG, originating from the point of declaration of \( v \), we recompute \( v \)'s interval when \( v \) is redefined according to the following rules:

- If \( v \) is assigned a constant value \( C \) (or, expression evaluating to a constant value), then \( v \)'s interval is defined to be \([C, C]\).
- If \( v \) is assigned an arithmetic expression in the form of \( v = OP_x \), then \( v \)'s interval is defined to be the corresponding arithmetic operation \( OP \) applied to \( x \)'s interval.
- If \( v \) is assigned an arithmetic expression in the form of \( v = x \), then \( v \)'s interval is the corresponding arithmetic operation \( OP \) applied to \( x \)'s interval.
- If \( v \) is assigned a complex arithmetic expression, then the complex arithmetic expression is decomposed into a set of unary or binary operations as defined above.
- If \( v \) is assigned a statically undeterminable function, then \( v \)'s interval is defined according to its type.

Let us extend the notion of \( v \)'s interval by associating a conditional expression with \( v \)'s interval (third column in Table 4). The goal is to capture the fact that \( v \)'s interval takes on different values along different paths (forks based on conditional expression) in the CDFG. For example, line 4 of Table 4 shows a conditional assignment to variable \( v \), based on the values of the input variables \( x_1 \) and \( x_2 \). In this example, when \((x_1 > 0) \&\& (x_2 > 0)\) \( v \)'s interval is defined to be \([1, 1]\), otherwise, \( v \)'s interval is defined to be \([0, 0]\).

Let us establish an equivalence between a conditional expression and a set of spaces (fourth column in Table 4). For each conditional expression \( cond_{expr} \), there exists a set of spaces \( S_1, S_2, \ldots, S_i \) that collectively defines the part of the domain space for which \( cond_{expr} \) evaluates to \( true \). For example, line 4 of Table 4 shows the conditional expression \((x_1 > 0) \&\& (x_2 > 0)\) defined as \([1, 10] \times [1, 5]\).

Table 4: Interval-set example

<table>
<thead>
<tr>
<th>Code (cond_{expr})</th>
<th>Interval (2)</th>
<th>Location (2)</th>
<th>Space (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ loop var: x_2</td>
<td>([-5, 5])</td>
<td>true</td>
<td>([-10, 10] \times [-5, 5])</td>
</tr>
<tr>
<td>1: $b_0$, v := 0;</td>
<td>[0, 1]</td>
<td>true</td>
<td>([-10, 10] \times [-5, 5])</td>
</tr>
<tr>
<td>2: if(x_1 &gt; 0) &amp;&amp; x_2 &gt; 0</td>
<td>([1, 1])</td>
<td>((x_1 &gt; 0) &amp;&amp; (x_2 &gt; 0))</td>
<td>([1, 10] \times [1, 5])</td>
</tr>
</tbody>
</table>

Formally, for a variable \( v \), the interval set (i.e., \( v \).iset) is defined as \( \{(I_j, S_j)\}_{j = 1}^{\text{iteration space}} \), where \( I_j \) is an integer interval and \( S_j \) a space. Furthermore, \( \bigcup_{j=1}^{\text{iteration space}} S_j \) = iteration_space. Intuitively, the interval set captures the range of values that a variable may receive during the execution of a program, taking the control flow into account.

A procedure for computing the output interval-set of a CDFG follows:

1. Topologically sort the CDFG's basic blocks and obtain \( b_0, b_1, \ldots, b_n \), repeat steps 2-5 for each basic block in sorted order.

2) Compute the interval set(s) for every DFG in \( b_i \).
3) Perform domain space partitioning analysis on the conditional expression at the exit of \( b_i \) [4].
4) Use the true and unknown spaces to compute the interval set(s) of the input variables of \( b_i \)'s jump-through basic block.
5) Use the false and unknown spaces to compute the interval set(s) of the input variables of \( b_i \)'s fall-through basic block.

Applying the above algorithm on the st_{cond,expr} CDFG would yield the interval set of the Boolean variable \( v \):

\[
v.iset = \{ (\{1, 1\}, S_{T1}), (\{1, 1\}, S_{T2}), \ldots, (\{1, 1\}, S_{Tn}) \},
(\{0, 0\}, S_{F1}), (\{0, 0\}, S_{F2}), \ldots, (\{0, 0\}, S_{Fn})
(\{0, 1\}, S_{U1}), (\{0, 1\}, S_{U2}), \ldots, (\{0, 1\}, S_{Un}) \}.
\]

Furthermore, we define three sets of spaces:

\[
T = \{ S_{T1}, S_{T2}, \ldots, S_{Tn} \},
F = \{ S_{F1}, S_{F2}, \ldots, S_{Fn} \}, U = \{ S_{U1}, S_{U2}, \ldots, S_{Un} \}.
\]

For the example of Table 4, the interval set of the Boolean variable \( v \) is:

\[
v.iset = \{ (\{1, 1\}, [1, 10] \times [1, 5]), (\{0, 0\}, [-10, 10] \times [-5, 5]),
(\{0, 0\}, [1, 10] \times [-5, 0]) \}
\]

4.2 Dependence Vector Computation

Data dependency in a loop is either of type loop-carried or of type loop-independent. Loop-independent dependency occurs when statements \( s_1 \) and \( s_2 \) access the memory location \( M \) during the same loop iteration. Loop-carried dependency occurs when statement \( s_1 \) accesses the memory location \( M \) in one iteration and \( s_2 \) accesses it in some iteration later. In this discussion, statements \( s_1 \) and \( s_2 \) may belong to any of \( st_{cond,expr}, st_{then} \) or \( st_{else} \).

For each iteration of the nested loop structure, we define a vector \( I = \{i_1, \ldots, i_n\} \) of integers showing the corresponding values of the loop indices. If there is a data dependency between statement \( s_1 \) during iteration \( I = \{i_1, \ldots, i_n\} \) and statement \( s_2 \) during iteration \( J = \{j_1, \ldots, j_n\} \), then the dependence vector is defined as \( J - I = \{j_1 - i_1, ..., j_n - i_n\} \).

The notion of dependence vector is well established in the compiler literature [6]. The existing dependence vector analysis techniques make the conservative assumption that any pair of statements within a loop body may execute during the same iteration. For the proposed transformation, we extend the analysis of dependence vector to account for control flow dependency between a pair of statements with the loop body, as described below.

Figure 4 shows our general \( m \)-dimensional memory access model. Figure 4-a shows the case when both statements access an array during the execution of the then part. Figure 4-b shows the case when one statement accesses an array during the execution of the then part and the other statement accesses an array during the execution of the else part.

In the case of Figure 4-a, there exists a data dependence if there are two iteration vectors \( I \) and \( J \) such that:

\[
f_k(I) = g_k(J) \land \forall k, 1 \leq k \leq m\ &\&
st_{cond,expr}(I) = true \land \&\& st_{cond,expr}(J) = true \quad (1)
\]
Figure 4: General Memory Access Model

In the case of Figure 4-b, there exists a data dependence if there are two iteration vectors I and J such that:

\[ f_k(I) = g_k(J) \forall k, 1 \leq k \leq m \quad \& \quad st_{cond,expr}(I) = true \quad \& \quad st_{cond,expr}(J) = false \quad (2) \]

In the case that both of the accesses are in the else part, then \( st_{cond,expr}(I) \) and \( st_{cond,expr}(J) \) in Equation 1 are equal to true. Similarly, the case when the write access is in the else part and the read access is in the then part, \( st_{cond,expr}(I) = false \) and \( st_{cond,expr}(J) = true \) in Equation 2.

4.3 Iteration Space Partitioning

Recall that sets T, F and U were computed according to Section 4.1. Likewise, the dependence vector was computed in Section 4.2. We define the first problem of iteration space partitioning as below:

**Problem 1:** Given T, F and U and the dependence vector between the points in that space we are interested in \( p = |T| + |F| + |U| \) sorted spaces \( S_1, S_2, ..., S_p \) in a way that there is no loop-carried data dependence from \( S_i \) to \( S_j \) if \( i < j \).

In general, solving Problem 1 requires finding the dependencies for the whole iteration space (i.e., solving equations \( \forall k \in (1, m)f_k(i_1, i_2, ..., i_n) = g_k(i_1, i_2, ..., i_n) \) in Figure 4) for arbitrary equations, which is a known NP-hard [6] problem. However, in two special cases, the problem can be solved efficiently. The first obvious case is when it is known (e.g., via a pragma directive) that there is no loop-carried data dependence. Here, the spaces can be sorted in any arbitrary way. The second case is when the dependency relationship is expressed as a linear equation of a special form. Specifically, if \( f_k \)’s and \( g_k \)’s in Figure 4 can be expressed as:

\[ \forall k \in (1, n)f_k(i_1, i_2, ..., i_n) = f_k(i_k) = a_{k,1} \times i_k + \beta_{k,1} \]
\[ \forall k \in (1, n)g_k(i_1, i_2, ..., i_n) = g_k(i_k) = a_{k,2} \times i_k + \beta_{k,2} \]

If \( a_{k,1} = a_{k,2} \) then the dependence vector can be expressed as \( \{\beta_{1,1} - \beta_{1,2}, ..., \beta_{n,1} - \beta_{n,2}\} \). Hence, Problem 1 can be re-defined as Problem 2 below:

**Problem 2:** Given T, F and U and the dependence vector in the form of \( \{\beta_{1,1} - \beta_{1,2}, ..., \beta_{n,1} - \beta_{n,2}\} \) we are interested in \( p = |T| + |F| + |U| \) sorted spaces \( S_1, S_2, ..., S_p \) in a way that there is no loop-carried data dependence from \( S_i \) to \( S_j \) if \( i < j \).

Algorithm 1 shows the proposed solution for Problem 2. Algorithm 1 first expands the boundaries of all the spaces using the dependence vector (line 6). Algorithm 1 then, finds all the spaces which have overlap with the expanded region, which gives, for each space, the set of dependent spaces (line 7). Using these dependencies, a set of relations between spaces is built (lines 8-10). Finally, Algorithm 2 is used as a subroutine to sort the spaces (line 12).

Algorithm 2 works as follows. In a partially sorted list of spaces, if it reads a relation \( S_i < S_j \) and if \( S_i \) is located after \( S_j \) in the list, their locations in the list are exchanged (lines 16-21). If any of \( S_i \) and \( S_j \) is not in the list, it is added to the list in a way to preserve the precedence relation (i.e., \( S_i \) before \( S_j \) if \( S_i < S_j \) and etc.) (lines 6-15).

Algorithm 1 Sort the spaces using the dependence vector

1: Input: T, F, U
2: Input: dependencyVector = \{\beta_{1,1} - \beta_{1,2}, ..., \beta_{n,1} - \beta_{n,2}\}
3: Output: Sorted(T, F, U)
4: relationSet = \phi
5: for all spaces \( S_i \in \{T, F, U\} \) do
6: expandedSpace = expandSpace(\( S_i \), dependencyVector)
7: overlappedSpaces = findOverlap(expandedSpace)
8: for all Spaces \( S_j \in overlappedSpaces \) do
9: relationSet = relationSet \cup \{S_i < S_j\}
10: end for
11: end for
12: sortedSpaces = RelationalSort(relationSet, \{T, F, U\})
13: return(sortedSpaces)

Figure 5 shows an example run of Algorithms 1 and 2. Figure 5-(a) shows the spaces that are dependent on the space \( S_1 \) by expanding the boundaries of \( S_1 \) using the dependence vector \( \beta \). It also shows the relative set which is built by applying Algorithm shown in Figure 1 on all the spaces. Figure 5-(b) shows the result of executing Algorithm 2 on the relative set shown in Figure 5-(a) and finally Figure 5-(c) shows the sorted spaces under the dependence vector \( \beta \).

4.4 Code Generation

Given the sorted spaces \( S_1, S_2, ..., S_p \), code generation entails emitting a loop for the \( S_i \)s. We note that, \( S_i = [l_i, u_i] \times [l_{i2}, u_{i2}] \times ... \times [l_{in}, u_{in}] \). Hence, the loop control segment would be generated according to the following template:

\[ \begin{align*}
&\text{for}(x_1 = l_1; x_1 \leq u_1; x_1++) \\
&\text{for}(x_2 = l_2; x_2 \leq u_2; x_2++) \\
&\text{...}
\end{align*} \]
Moreover, the body of the generated loops contains only sthen if \(S_i \in T\), only stelse if \(S_i \in F\), or the original loop body if \(S_i \in U\).

Algorithm 2 Relational Sort

1: Input: \(T, F, U\)
2: Output: relationSet
3: 4: sortedList \(= \emptyset\)
5: for all Relation \(r_n = (S_i, S_j) \in \text{relationSet}\) do
6: if \((S_i \notin \text{sortedList}) \land (S_j \notin \text{sortedList})\) then
7: sortedList.push\((S_i)\)
8: sortedList.push\((S_j)\)
9: else if \((S_i \in \text{sortedList}) \land (S_j \notin \text{sortedList})\) then
10: \(i_{\text{index}} \leftarrow \text{sortedList}.\text{find}(S_i)\)
11: sortedList.insert\((S_i, i_{\text{index}})\)
12: else if \((S_i \notin \text{sortedList}) \land (S_j \in \text{sortedList})\) then
13: \(j_{\text{index}} \leftarrow \text{sortedList}.\text{find}(S_j)\)
14: sortedList.insert\((S_j, j_{\text{index}} - 1)\)
15: else
16: \(i_{\text{index}} \leftarrow \text{sortedList}.\text{find}(S_i)\)
17: \(j_{\text{index}} \leftarrow \text{sortedList}.\text{find}(S_j)\)
18: if \(i_{\text{index}} > j_{\text{index}}\) then
19: sortedList.remove\((S_i)\)
20: sortedList.insert\((S_i, j_{\text{index}} - 1)\)
21: end if
22: end if
23: end for
24: return(sortedSpaces)

5. EXPERIMENTS

To evaluate the proposed code transformation technique, several loop kernels from Mediapack [2] application suite and SPEC-2000 [1] were chosen. We also experimented with an mp3 encoder implementation obtained from [14], an mpegq full motion estimation obtained from [3], GNU Image Manipulation Program (gimp) [11] and also qsdpcm [9] video compression algorithm which is obtained from [5].

By loop kernel, 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. The characteristics of the loop kernels selected for our experiments are listed in Table 5. In Table 5 conditional expressions column shows the particular conditional expression(s). If there are more than one conditional expression in a loop kernel, 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 5, Application column shows where we picked the loop kernel and Function description column shows the functionality of the code where the kernel is taken from. 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 three types of processors: SPARC, Intel and PowerPC. For all processors, we measured the performance improvement together with code size increase.

Note that there are cases where we measured decrease in code size (e.g., mpegdec-vhfilter), this is due to removal of the conditional expression evaluation from the code and small number of partitions that are generated. Note that since there are real runtime results on real machines, they naturally factor in any possible performance effects of code size increase on caching. Thus the speedup is the real effect of the transformation on actual running code.

Experiments with GCCs increasing levels of optimizations (none, -O1, -O2, -O3) show that the proposed optimization techniques yields additional performance improvements when applied in conjunction with existing compiler optimizations in vast majority of cases. In the few cases where this is not true (e.g., 186. crafty in Intel or PowerPC or qsdpcm in PowerPC), the difference is within measurement noise. Furthermore, this is a well known effect of interactions between compiler optimizations and is indeed also visible without our transformations (e.g., 175.vpr for SPARC and qsdpcm for Intel and PowerPC) as shown in Tables 6, 7 and 8.

Each loop kernel (original and transformed) was compiled using different optimization levels of gcc [10], namely: no optimization (shown as no in the following sections); using -O1 switch; using -O2 switch and finally using -O3 switch. In the following sections, the speedup calculations are based on the ratio of the time to execute the optimized loop kernel to the time to execute the original loop kernel. In each case the execution time before code transformation \(T_o\) and the execution time after code transformation \(T_n\) are measured and speedup improvement has been calculated using the following formula: Improvement\(\%\) = \((1 - T_n/T_o) \times 100\). Each bar in Figure 6, 8 and 10 shows the time improvement after applying our code transformation. For each benchmark there are 4 bars, representing the time improvement for 4 cases of optimizations mentioned above.

5.1 SPARC

The results of experiments on SPARC CPU are summarized in Table 6. The first half of Table 6 shows the result of measured time before and after transformation for 4 different optimization options. The second half of Table 6 shows the result of code size before and after transformation for the same 4 optimization options plus another optimization for code size (-Os). The percentage of time and code size change has been shown graphically in Figure 6 and Figure 7.

The experiments were run on a Sun workstation, with 2 SPARC CPUs (1503 MHz SUNW, UltraSPARC-IIIi) and 2 GB of memory, but the code ran for all experiments on a single CPU. We used GCC compiler version 3.4.1 in order to generate executables. In the best case, we observed application speedup of 551\% (6.51X). On average, we observed application speedup of 175\% (2.75X). On average we observed 150.9\% increase on code size.

5.2 Intel X86

The results of experiments on Intel CPU are summarized in Table 7. The first half of Table 7 shows the result of measured time before and after transformation for 4 different optimization options. The second half of Table 7 shows the result of code size before and after transformation for the same 4 optimization options plus another optimization for code size (-Os). The percentage of time and code size change has been shown graphically in Figure 8 and Figure 9.

The experiments were run on a MacBook with an Intel Dual Core 1.8GHz and 1 GB of memory. We used GCC compiler version 3.4.1 in order to generate executables. In the best case, we observed application speedup of 965\% (10.65X). On average, we observed application speedup of 336\% (4.36X). On average we observed 134.2\% increase on code size.
5.3 PowerPC

The results of experiments on ppc CPU are summarized in Table 8. The first half of Table 8 shows the result of measured time before and after transformation for 4 different optimization options. The second half of Table 8 shows the result of code size before and after transformation for the same 4 optimization options plus another optimization for code size (-Os). The percentage of time and code size change has been shown graphically in Figure 10 and Figure 11.

The experiments were run on a Apple PowerMac G5 with a 1.6 GHz PowerPC G5 and 768 MB of memory. We used GCC compiler version 4.0.1 in order to generate executables. In the best case, we observed application speedup of 812% (8.12X). On average, we observed application speedup of 198.9% (2.98X). On average, we observed a 136.2% increase on code size.

6. ACKNOWLEDGEMENT

This work was in part supported by grant #0749508 from the National Science Foundation.

7. CONCLUSION

Given the stringent design constraints and performance requirements of embedded systems, as software becomes more dominant, the importance of aggressive compiler optimizations also increases. Hence, it is acceptable for a compiler intended for embedded computing to take longer to execute but perform aggressive compiler optimizations.

We have presented a new loop transformation technique, intended for embedded compilers. The transformation technique optimizes loops with nested conditional blocks and it decomposes the loop nests in a way that conditional testing is eliminated. Applying the proposed transformation technique on the loop kernels taken from Mediabench, SPEC2000, mpg4, gsdpcm and gimp, on average we measured a 175% (1.75X) improvement of execution time when running on a SPARC processor, a 336% (4.36X) improvement of execution time when running on an Intel Core Duo processor and a 198.9% (2.98X) improvement of execution time when running on a PowerPC G5 processor. We used high-end processors because better compilers are available, so as to avoid the possibility that our technique looks better than it should because of poor optimizations done by the compiler. Also, these processors are representative of high-end embedded processors (Intel Core-duo has an embedded version, so do PowerPC and SPARC). We measured a code size increase of 150.9% for SPARC, 134.2% for Intel and 136.2% for PowerPC. Note that despite the size increase, the overall

Table 5: Selected Application List

<table>
<thead>
<tr>
<th>Application</th>
<th>Function desc.</th>
<th>Conditional expressions</th>
<th>Properties(Table2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg4</td>
<td>Motion estimation</td>
<td>(x &lt; 0) (x &gt; 35) (y &lt; 0) (y &gt; 46)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>gsdpcm</td>
<td>Motion estimation</td>
<td>(x &gt; 0) (x &gt; 35) (y &lt; 0) (y &gt; 46)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>gimp</td>
<td>Create Kernel</td>
<td>(32 * x - 2 * i + 1) * 132 * y + 2 * j + 1)^2 &lt; 4096</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>122.tachyon</td>
<td>Parallel ray tracing</td>
<td>(y &gt;= N - 1), (z &gt;= N - 1)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>186.candy</td>
<td>Chess program</td>
<td>(x &lt; 10), (y &gt; 47)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>175.vippe</td>
<td>FPGA Circuit Placement and Routing</td>
<td>(i &gt;= 4k &amp;&amp; i &gt;= DETAIL,START + 5 &amp;&amp; i = 5 &amp;&amp; k &gt; 1)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>252.som</td>
<td>Computer Visualization</td>
<td>(t == 0), (t == 0), (t == 0)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>253.peribank</td>
<td>PERL Programming Language</td>
<td>(x &lt; 4 &amp;&amp; k &lt; c &amp;&amp; c &lt; y)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>mpgdec-initdec</td>
<td>Initial Decoder</td>
<td>(x &lt; 0), (x &gt; 255)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>mpgdec-filter</td>
<td>Ver. / Hor. Filter for 2D Subsample</td>
<td>(x &lt; 1), (y &lt; 1), (x &lt; 1), (y &lt; 1)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>mpgdec-luma</td>
<td>Layer 3 Psych. Analysis</td>
<td>(x = x &amp;&amp; y = y &amp;&amp; z = z &amp;&amp; k = k)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>mpgdec-mpeg10</td>
<td>Head and signs audio data</td>
<td>(x &lt; 16)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>mpgdec-vhfilter</td>
<td>Ver. / Hor. Interpolation Filter</td>
<td>(x &lt; 256), (y &lt; 255)</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>
performance is still improved by the above factors, i.e., cache performance degradation, if any, due to the increased code size is already factored into the results, since we measured actual runtime of the original and transformed code.

8. REFERENCES


Table 8: Result of Experiments for PowerPC-Time and code size (Shaded: Original; White: Transformed)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (Original and Transformed)</th>
<th>Code size (Original and Transformed)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>-O1</td>
</tr>
<tr>
<td>mpeg4</td>
<td>111968</td>
<td>19876</td>
</tr>
<tr>
<td>qdpcm</td>
<td>71244</td>
<td>72052</td>
</tr>
<tr>
<td>gimp</td>
<td>81994</td>
<td>67692</td>
</tr>
<tr>
<td>122 tachyon</td>
<td>77034</td>
<td>64740</td>
</tr>
<tr>
<td>186 crafty</td>
<td>105844</td>
<td>103300</td>
</tr>
<tr>
<td>175 vpr</td>
<td>11986</td>
<td>9296</td>
</tr>
<tr>
<td>252 com</td>
<td>423</td>
<td>417</td>
</tr>
<tr>
<td>253 perlbufk</td>
<td>8572</td>
<td>1748</td>
</tr>
<tr>
<td>graphics</td>
<td>1756</td>
<td>1080</td>
</tr>
<tr>
<td>mpgdec-initdec</td>
<td>3828</td>
<td>2604</td>
</tr>
<tr>
<td>mpgenc-vh↓lter</td>
<td>7112</td>
<td>3724</td>
</tr>
<tr>
<td>mpg3-psy3</td>
<td>4410</td>
<td>4828</td>
</tr>
<tr>
<td>mpg3-align</td>
<td>17062</td>
<td>16092</td>
</tr>
<tr>
<td>mpgdec-vh↓lter</td>
<td>2960</td>
<td>1936</td>
</tr>
<tr>
<td>mpg3-psy3</td>
<td>1126</td>
<td>484</td>
</tr>
</tbody>
</table>

Figure 8: Effect of transformation on time for Intel

Figure 9: Effect of transformation on code size for Intel

Figure 10: Effect of transformation on time for PowerPC

Figure 11: Effect of transformation on code size for PowerPC