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

CASES'08, October 19–24, 2008, Atlanta, Georgia, USA.

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].

```
#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);
            for (i = 0; i < STEPS * KERNEL_WIDTH; i++)  {
                dist_x = x - (((double) i + 0.5) / (double) STEPS);
                if ((SQR (dist_x) + SQR (dist_y)) < THRESHOLD)
                    w = 1.0;
                else
```

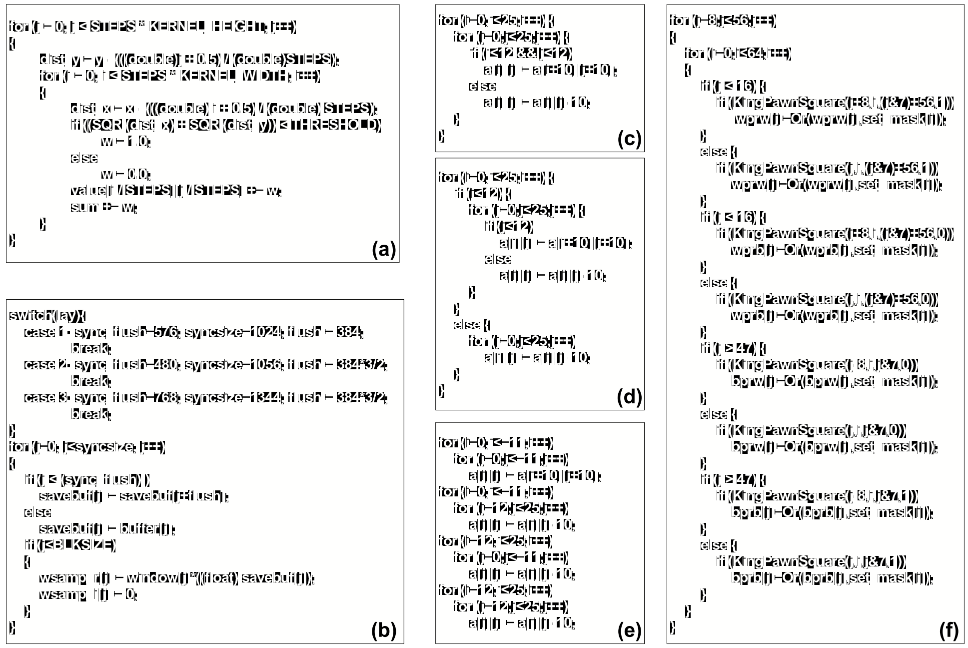

Figure 1: (a) 1st difference - gimp (b) 2nd difference - mp3 (c-e) 3rd difference - synthetic example (f) 3rd difference - 186.crafty

```
                w = 0.0;
            value[i / STEPS][j / STEPS] += w;
        }
    }
  }
}
```

Table 1: Interval analysis result for the expression $(SQR(dist_x) + SQR(dist_y)) < THRESHOLD$

Space([xi][yj][i][j])	Evaluation result(false/true)
[0, 1][0, 1][152, 191][0, 191]	false
[0, 1][2, 2][123, 191][0, 191]	false
[0, 1][3, 4][157, 191][0, 191]	false
[2, 2][0, 1][0, 28][0, 191]	false
[2, 2][0, 1][163, 191][0, 191]	false
[2, 2][2, 2][0, 63][0, 191]	false
[2, 2][2, 2][128, 191][0, 191]	false
[3, 4][0, 1][0, 40][0, 191]	false
[3, 4][2, 2][0, 68][0, 191]	false
[3, 4][3, 4][0, 34][0, 191]	false

Using interval analysis [4] we statically compute information as shown in Table 1 on the conditional expression in the loop nest $((SQR(dist_x) + SQR(dist_y)) < THRESHOLD)$. For example the 2nd row of Table 1 shows that when $(0 \leq xi \leq 1)$ && $(0 \leq yj \leq 1)$ && $(152 \leq i \leq 191)$ && $(0 \leq j \leq 191)$ the expression $(SQR(dist_x) + SQR(dist_y)) < THRESHOLD$ evaluates to `false`. The transformed code, using the 2nd row of Table 1 yields the following optimized code:

```
for (yj = 0; yj <= 1; yj++) {
    y = (double) yj / (double) SUBSAMPLE;
    for (xi = 0; xi <= 1; 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);
            for (i = 0; i < 152; i++)  {
                dist_x = x - (((double) i + 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;
            }
            for (i = 152; i < STEPS * KERNEL_WIDTH; i++)  {
                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

Table 2: Properties which are being compared in Table 3

Property 1	Optimize control 'ow of a loop with nested conditional block
Property 2	Dependence analysis needed
Property 3	Conditional expression depends on loop index
Property 4	Conditional expression is an a¡ne function of loop variables
Property 5	Conditional expression contains logical operators
Property 6	Conditional expression is a function of loop indices and non-loop-index variables
Property 7	Conditional expression has a general polynomial form
Property 8	Conditional expression will be removed completely from loop body of the transformed code

Table 3: Comparison with other loop optimization techniques

		Property(Table 2)							
Optimization technique	Summary of technique	1	2	3	4	5	6	7	8
Loop fusion	Fuse two adjacent countable loops with the same loop limits		✓						
Loop ↓ssion	Broke a single loop into two or more smaller loops		✓						
Loop interchanging	For two nested loops, switch the inner and outer loop		✓						
Loop skewing	For two nested loops, change the indices in a way that remove the dependence from the inner loop		✓						
Strip-mining	Decompose a single loop into an outer loop which steps between strips of consecutive iterations and an inner loop which steps between single iterations within a strip		✓						
Loop tiling	Same as strip-mining for nested-loops and convex shaped iteration space		✓						
Loop collapsing	Two nested loops that refers to arrays be collapsed into single loop		✓						
Loop coalescing	Same as Loop collapsing but the loop limits do not match		✓						
Loop unrolling	Duplicate the body of the loop multiple times and reduce the loop count								
Loop unswitching	Remove loop independent conditional from a loop	✓							✓
Loop peeling	Remove the ↓rst or last iteration of the loop into separate code	✓		✓					✓
Index set splitting	Divides the index set of a loop into two portions	✓		✓					✓
Loop nest splitting [3]	For a nested loop, by using polytope model and genetic algorithms, conditions having no e↑ect on control 'ow are removed or moved up in loop nest	✓		✓	✓	✓			
Our work	For a nested loop, by using interval analysis technique [4] and dependence analysis, the nested loop is partitioned into multiple loops with the no condition	✓	✓	✓	✓	✓	✓	✓	✓

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 unswitching 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 conditionals 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 remove 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 $x_1, ..., x_n$. For every index x_k, the value for lower (upper) bounds lb_k (ub_k) 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, st_{cond_expr} computes the value of the branch condition v.

3. PROPOSED TRANSFORMATION

The proposed transformation decomposes the original

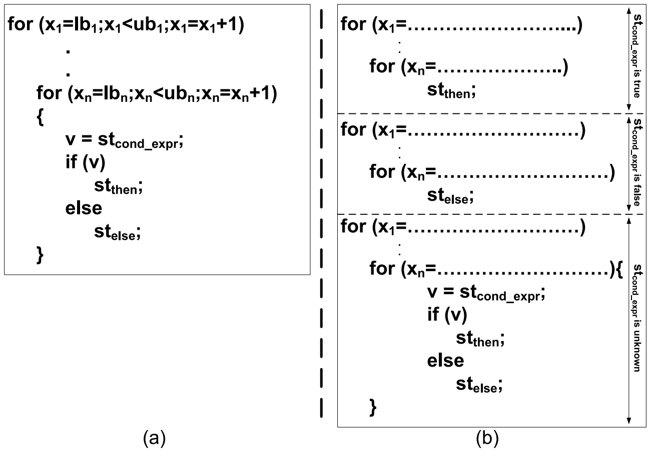

Figure 2: Transformation

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 st_{cond_expr} is known to be `true` at compile time. Likewise, the second part sets up one or more nested loop structures with iteration spaces for which the st_{cond_expr} is known to be `false` at compile time. The third part sets up one or more nested loop structures with an iteration space for which the st_{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 st_{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 *cond_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* (&&, ||, !).

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 $[l_0, u_0] \times [l_1, u_1] \times ... \times [l_{n-1}, u_{n-1}]$. 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 $[min_0, max_0] \times [min_1, max_1] \times ... \times [min_n, max_n]$, where min_i and max_i are defined based on the type of the variable x_i (e.g. if x_i is of type *signed character* then $min_i = -128$ and $max_i = 127$).

Given the conditional expression *cond_expr* with variables $x_1, x_2, ..., x_k$, the domain space partitioning problem [4] is to partition the domain space of *cond_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 *cond_expr* evaluates to `true` for every point in space s_i. If space s_i has a Boolean value of `false`, then *cond_expr* evaluates to `false` for every point in space s_i. If space s_i has a Boolean value of `unknown`, then *cond_expr* may evaluate to `true` for some points in space s_i and `false` for others.

For example, consider *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].

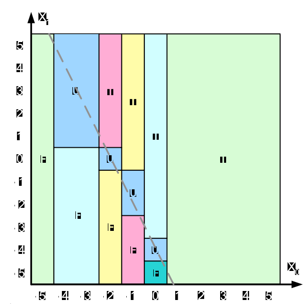

Figure 3: Partitioned Domain of $2x_0 + x_1 + 4 > 0$

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 st_{cond_expr}, st_{then}, and st_{else}. The variable v, computed by st_{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 st_{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 st_{cond_expr} CDFG) and Boolean variable v is assumed to be live on exit (i.e., output of the st_{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 a unary arithmetic expression in the form of $v = OP\ x_i$, then v's interval is defined to be the corresponding arithmetic operation OP applied to x_i's interval.
- If v is assigned a binary arithmetic expression in the form of $v = x_i\, OP\, x_j$, then v's interval is defined to be the corresponding arithmetic operation OP applied to x_i's and x_j's intervals.
- 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, than 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 assignments 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 $S_1, S_2, ..., S_k$ 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

Code (st_{cond_expr})	Interval	Condition	Space
// loop var: x_1	$[-10, 10]$		
// loop var: x_2	$[-5, 5]$		
1: bool v;	$[0, 1]$	true	$[-10, 10] \times [-5, 5]$
2: v=0;	$[0, 0]$	true	$[-10, 10] \times [-5, 5]$
3: if(x_1>0&&x_2>0)			
4: v=1;	$[1, 1]$	$(x_1 > 0\&\& x_2 > 0)$	$[1, 10] \times [1, 5]$

Formally, for a variable v, the *interval_set* (i.e., *v.iset*) is defined as $\{(I_j, S_j)|j \in (1...m)\}$, where I_j is an integer interval and S_j a space. Furthermore, $\bigcup_{j=1}^{m} S_j =$ *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 $b_0, b_1, ..., b_n$, 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:

$$v.iset = \{([1,1], S_{T1}), ([1,1], S_{T2}), ..., ([1,1], S_{Tn_1}), \\ ([0,0], S_{F1}), ([0,0], S_{F2}), ..., ([0,0], S_{Fn_2}), \\ ([0,1], S_{U1}), ([0,1], S_{U2}), ..., ([0,1], S_{Un_3})\}.$$

Furthermore, we define three sets of spaces:

$$T = \{S_{T1}, S_{T2}, ..., S_{Tn_1}\}, \\ F = \{S_{F1}, S_{F2}, ..., S_{Fn_2}\}, U = \{S_{U1}, S_{U2}, ..., S_{Un_3}\}.$$

For the example of Table 4, the *interval_set* of the Boolean variable v is:

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

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 st_1 and st_2 access the memory location M during the same loop iteration. Loop-carried dependency occurs when statement st_1 accesses the memory location M in one iteration and st_2 accesses it in some iteration later. In this discussion, statements st_1 and st_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 $I = \{i_1, ..., i_n\}$ of integers showing the corresponding values of the loop indices. If there is a data dependency between statement st_1 during iteration $I = \{i_1, ..., i_n\}$ and statement st_2 during iteration $J = \{j_1, ..., j_n\}$, 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:

$$f_k(I) = g_k(J)\ \forall k,\ 1 \le k \le m\ \&\& \\ st_{cond\_expr}(I) = true\ \&\&\ st_{cond\_expr}(J) = true \quad (1)$$

```
for (x_1=lb_1;x_1<ub_1;x_1=x_1+1)
 .
 .
 for (x_n=lb_n;x_n<ub_n;x_n=x_n+1)
 {
   v = st_cond_expr;
   if (v)
   {
     ...
     A[f_1(x_1,...,x_n)]..[f_m(x_1,...,x_n)] = ...;
     ...= A[g_1(x_1,...,x_n)]..[g_m(x_1,...,x_n)];
     ...
   }
   else
     st_else;
 }
```
(a)

```
for (x_1=lb_1;x_1<ub_1;x_1=x_1+1)
 .
 .
 for (x_n=lb_n;x_n<ub_n;x_n=x_n+1)
 {
   v = st_cond_expr;
   if (v)
   {
     ...
     A[f_1(x_1,...,x_n)]..[f_m(x_1,...,x_n)] = ...;
     ...
   }
   else
   {
     ...
     ... = A[g_1(x_1,...,x_n)]..[g_m(x_1,...,x_n)];
     ...
   }
}
```
(b)

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:

$$f_k(I) = g_k(J)\,\forall k,\ 1 \le k \le m\ \&\&$$
$$st_{cond\_expr}(I) = true\ \&\&\ st_{cond\_expr}(J) = false \qquad (2)$$

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 `false`. 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 $\forall k \in (1, ..., m) f_k(i_1, ..., i_n) = g_k(i_1, ..., i_n)$ 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:

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

If $\forall k\ \alpha_{k,1} = \alpha_{k,2}$ 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 dependency from S_i to S_j if $i < j$.

Algorithm 1 shows the proposed solution for Problem 2. Algorithm 1 first expand 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:** $dependency_{vector} = \{\beta_{1,1} - \beta_{1,2}, ..., \beta_{n,1} - \beta_{n,2}\}$
3: **Output:** $Sorted\{T, F, U\}$
4: $relationSet \leftarrow \phi$
5: **for all** Spaces $S_i \in \{T, F, U\}$ **do**
6: $\quad expanded_{space} \leftarrow expandSpace(S_i, dependency_{vector})$
7: $\quad overlapped_{spaces} \leftarrow findOverlap(expanded_{space})$
8: $\quad$ **for all** Spaces $S_j \in overlapped_{spaces}$ **do**
9: $\qquad relationSet \leftarrow relationSet \cup (S_i < S_j)$
10: $\quad$ **end for**
11: **end for**
12: $sortedSpaces \leftarrow RelationalSort(relationSet, \{T, F, U\})$
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_3 by expanding the boundaries of S_3 using the dependence vector β. 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 dependency vector β.

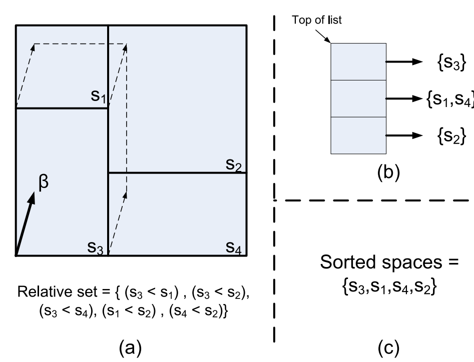

Figure 5: Example run of Algorithms 1 and 2

4.4 Code Generation

Given the sorted spaces $(S_1, S_2, ..., S_p)$, code generation entails emitting a loop for the S_is. We note that, $S_i = [l_1, u_1] \times [l_2, u_2] \times ... \times [l_n, u_n]$. Hence, the loop control segment would be generated according to the following template:

for($x_1 = l_1$; $x_1 \le u_1$; x_1++)
  for($x_2 = l_2$; $x_2 \le u_2$; x_2++)

...
for($x_n = l_n$; $x_n \le u_n$; x_n++)
`body`

Moreover, the `body` of the generated loops contains only st_{then} if $S_i \in T$, only st_{else} if $S_i \in F$, or the original loop body if $S_i \in U$.

Algorithm 2 Relational Sort

Input: T, F, U
Input: $relationSet$
Output: $Sorted\{T, F, U\}$
$sortedList \leftarrow \phi$
for all Relation $r_k = (S_i < S_j) \in relationSet$ **do**
  if $(S_i \notin sortedList) and (S_j \notin sortedList)$ **then**
    $sortedList.push(S_i)$
    $sortedList.push(S_j)$
  else if $(S_i \in sortedList) and (S_j \notin sortedList)$ **then**
    $i_{index} \leftarrow sortedList.find(S_i)$
    $sortedList.insert(S_j, i_{index})$
  else if $(S_i \notin sortedList) and (S_j \in sortedList)$ **then**
    $j_{index} \leftarrow sortedList.find(S_j)$
    $sortedList.insert(S_i, j_{index} - 1)$
  else
    $i_{index} \leftarrow sortedList.find(S_i)$
    $j_{index} \leftarrow sortedList.find(S_j)$
    if $i_{index} >= j_{index}$ **then**
      $sortedList.remove(S_i)$
      $sortedList.insert(S_i, j_{index} - 1)$
    end if
  end if
end for
return(sortedSpaces)

5. EXPERIMENTS

To evaluate the proposed code transformation technique, several *loop kernels* from *MediaBench* [2] application suite and *SPEC-2000* [1] were chosen. We also experimented with an *mp3* encoder implementation obtained from [14], an *mpeg4* 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 speedups are 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: $Improvement(\%) = (1 - T_o/T_n) \times 100$. 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 a 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.

Table 5: Selected Application List

Application	Function desc.	Conditional expressions	Properties(Table2)							
			1	2	3	4	5	6	7	8
mpeg4	Motion estimation	$(x3 < 0 \| \| x3 > 35 \| \| y3 < 0 \| \| y3 > 48)$ $(x4 < 0 \| \| x4 > 35 \| \| y4 < 0 \| \| y4 > 48)$	✓	✓	✓	✓	✓			✓
qsdpcm	Motion estimation	$((4*x+vx-4+x4<0) \| \|$ $(4*x+vx-4+x4>(N/4-1)) \| \|$ $(4*y+vy-4+y4<0) \| \|$ $(4*y+vy-4+y4>(M/4-1)))$	✓	✓	✓	✓	✓			✓
gimp	Create Kernel	$(32*x-2*i+1)^2+(32*y-2*j+1)^2 < 4096$	✓	✓	✓				✓	✓
122.tachyon (SPECMPI-2007)	Parallel ray tracing (Generate Noise Matrix)	$(x == NMAX-1)$, $(y == NMAX-1), (z == NMAX-1)$	✓	✓	✓	✓	✓			✓
186.crafty (SPEC-2000)	Chess program (Generate Piece Masks)	$(j < 16), (j > 47)$	✓	✓	✓	✓				✓
175.vpr (SPEC-2000)	FPGA Circuit Placement and Routing (Check architecture ↓le)	$i! = 4\&\&i! = DETAILED\_START + 5\&\&$ $i! = 5\&\&i! = DETAILED\_START + 6$	✓	✓	✓	✓				✓
252.eon (SPEC-2000)	Computer Visualization	$(i == 0), (j == 0), (k == 0)$	✓	✓	✓	✓				✓
253.perlbmk (SPEC-2000)	PERL Programming Language	$((c >=' A'\&\&c <=' Z') \| \|$ $(c >=' a'\&\&c <=' z') \| \|$ $(c >=' 0'\&\&c <=' 9') \| \| c ==' \_')$	✓	✓	✓	✓	✓			✓
Synthetic graphics	Collision detection	$(x*x+y*y == x*x*y)$	✓	✓	✓				✓	✓
mpgdec-initdec	Initialize Decoder	$(i < 0), (i > 255)$	✓	✓	✓	✓				✓
mpgenc-vh↓lter	Ver./Hor. Filter,2:1 Subsample	$(i < 5), (i < 4), (i < 3), (i < 2), (i < 1)$	✓	✓	✓	✓				✓
mp3-psych	Layer 3 Psych. Analysis	$j < sync\_flush, j < BLKSIZE$	✓	✓	✓	✓		✓		✓
mp3-align	Read and align audio data	$j < 64$	✓	✓	✓	✓				✓
mpgenc-idct	IDCT Initialize	$(i < -256), (i > 255)$	✓	✓	✓	✓				✓
mpgdec-vh↓lter	Ver./Hor. Interpolation Filter	$(i < 2), (i < 1)$	✓	✓	✓	✓				✓

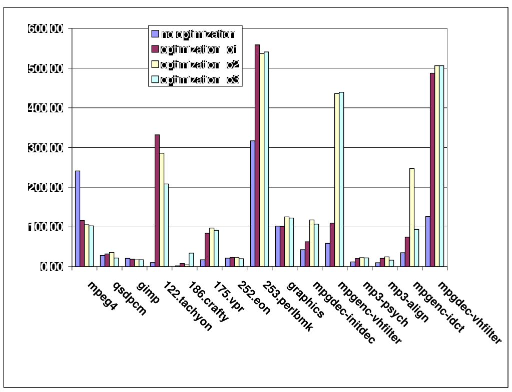

Figure 6: **Effect of transformation on time for SPARC**

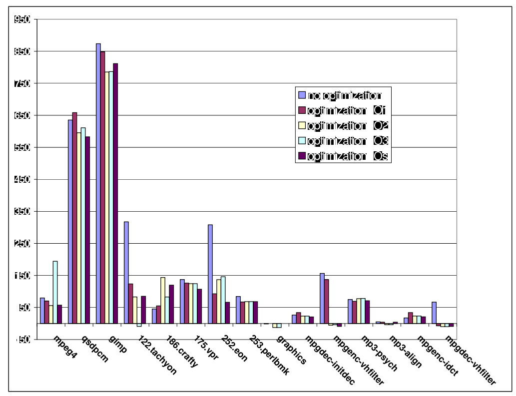

Figure 7: **Effect of transformation on code size for SPARC**

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 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*, *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. 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 6: Result of Experiments for Sparc-Time and code size(Shaded: Original;White: Transformed)

	Time (Original and Transformed)								Code size (Original and Transformed)									
Benchmark	No		-O1		-O2		-O3		No		-O1		-O2		-O3		-Os	
mpeg4	228638	67032	98920	45740	91782	44674	91608	45208	362	651	178	303	196	306	196	577	180	283
qsdpcm	138730	108332	26234	19884	19602	14446	17408	14304	253	1860	135	1024	141	981	153	1088	128	874
gimp	114054	94422	45870	38664	44960	38368	44928	38274	265	2580	158	1498	149	1319	149	1321	140	1276
122.tachyon	177416	161306	44714	10348	38294	9928	30614	9932	166	693	80	179	76	139	153	139	74	137
186.crafty	216310	212102	51436	47782	44636	42702	23584	17602	380	552	198	307	117	285	238	435	143	314
175.vpr	14050	11972	5902	3204	5990	3038	5846	3054	148	351	84	190	94	211	94	211	87	180
252.eon	590	487	594	484	591	481	586	489	350	1428	139	268	81	192	78	192	101	168
253.perlbmk	10474	2512	7138	1084	6798	1068	6806	1062	108	199	61	102	60	101	60	101	60	101
graphics	4982	2466	2320	1152	1338	594	1308	588	82	81	44	44	48	42	48	42	43	43
mpgdec-initdec	3438	2408	670	412	670	308	642	310	72	91	38	51	39	48	39	48	39	47
mpgenc-vh↓lter	12706	8010	5040	2406	4234	790	4238	786	295	756	151	358	130	123	128	123	120	109
mp3-psych	6184	5532	3930	3270	3834	3128	3764	3088	186	325	127	215	120	213	119	212	107	183
mp3-align	15106	13740	3604	2980	2972	2386	2736	2352	99	104	49	51	52	50	52	50	48	50
mpgenc-idct	3486	2582	718	412	1152	332	748	386	241	100	44	59	43	53	43	53	43	52
mpgdec-vh↓lter	2034	900	552	94	582	96	582	96	157	262	82	76	71	64	71	64	66	60

Table 7: Result of Experiments for Intel-Time and code size(Shaded: Original;White: Transformed)

	Time (Original and Transformed)								Code size (Original and Transformed)									
Benchmark	No		-O1		-O2		-O3		No		-O1		-O2		-O3		-Os	
mpeg4	36638	8366	16906	2866	16600	2738	17862	2846	365	636	211	305	193	270	212	290	204	288
qsdpcm	34452	27848	9460	7272	14850	10530	15930	11290	219	1486	179	1192	173	1091	218	1282	179	1125
gimp	18138	15936	16262	13380	16060	13448	16060	13448	210	2032	157	1372	133	1130	133	1130	131	1063
122.tachyon	40854	34438	15472	2502	15548	3246	8474	2486	143	649	82	190	73	135	95	138	73	135
186.crafty	37736	39652	19346	18116	21740	21110	8400	6840	346	508	272	422	250	420	362	488	314	494
175.vpr	2130	1400	1462	976	1272	732	1276	744	110	244	80	230	104	253	106	253	93	223
252.eon	126	123	24	11	35	25	28	25	285	1265	108	210	144	236	82	236	126	236
253.perlbmk	1850	462	762	152	762	150	756	150	83	158	59	100	64	106	64	106	60	100
graphics	250	122	58	22	52	30	52	30	60	60	48	48	53	48	53	48	53	48
mpgdec-initdec	540	448	158	52	140	60	140	60	57	68	49	53	54	54	57	54	49	54
mpgenc-vh↓lter	3012	1482	1000	158	980	92	980	92	254	653	175	126	176	129	176	129	171	129
mp3-psych	900	800	610	482	550	492	538	510	117	203	108	186	93	184	93	184	110	183
mp3-align	2770	2572	1240	644	846	614	786	640	74	71	59	54	59	61	59	61	59	51
mpgenc-idct	560	430	158	52	180	52	182	60	63	74	56	61	61	62	64	62	56	62
mpgdec-vh↓lter	438	170	216	30	110	12	110	20	136	216	99	82	103	80	103	80	97	80

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

[1] Standard Performance Evaluation Corporation. Spec cpu2000. Available as http://www.spec.org/cpu2000/.

[2] C. Lee et. al. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[3] H. Falk and P. Marwedel. Control flow driven splitting of loop nests at the source code level. In *Proceedings of DATE*, pages 410– 415, 2003.

[4] M.A. Ghodrat, T. Givargis, and A. Nicolau. Equivalence checking of arithmetic expressions using fast evaluation. In *Proceedings of the CASES*, pages 147 – 156, 2005.

[5] I. Issenin and N. Dutt. Data reuse driven energy-aware mpsoc co-synthesis of memory and communication architecture for streaming applications. In *CODES-ISSS 2006*, pages 294–299, 2006.

[6] K. Kennedy and R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[7] R.E. Moore. *Interval analysis*. Prentice-Hall, Englewood Cliffs, N. J., 1966.

[8] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[9] P. Stobach. A new technique in scene adaptive coding. In *Proceedings of EUSIPCO*, 1988.

[10] The GCC Team. Gnu compiler collection. Available as http://gcc.gnu.org/.

[11] The GIMP Team. Gnu image manipulation program. Available as http://www.gimp.org/.

[12] M. Wolfe. *High-Performance Compilers for Parallel Computing*. Addison Wesley, 1995.

[13] M. Wolfe. How compilers and tools differ for embedded systems. In *Proceedings of the CASES*, page 1, 2005.

[14] www.mp3tech.org. Iso mp3 sources. Available as http://www.mp3-tech.org/programmer /sources/dist10.tgz.

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

	Time (Original and Transformed)								Code size (Original and Transformed)									
Benchmark	No		-O1		-O2		-O3		No		-O1		-O2		-O3		-Os	
mpeg4	111968	19876	29638	4530	27166	4532	19562	4534	346	614	154	233	163	242	264	344	157	236
qsdpcm	71244	72052	9802	10412	9622	9346	16100	9188	240	1694	124	772	128	781	145	821	123	766
gimp	81894	67692	44642	35746	42344	34818	42320	34878	253	2361	141	1172	137	1156	154	1180	136	1140
122.tachyon	77394	64740	18878	6960	17888	8570	13286	7966	147	552	75	144	78	145	153	145	75	144
186.crafty	105844	103300	25782	27160	26090	26034	8590	8130	357	523	202	293	194	295	235	398	227	383
175.vpr	11986	9296	3500	2380	2974	2076	2972	2080	152	351	76	204	79	202	79	202	75	191
252.eon	423	417	67	51	62	44	62	44	239	955	92	184	92	197	84	197	86	196
253.perlbmk	8572	1748	1442	400	1452	390	1450	390	107	206	58	102	60	107	60	107	60	102
graphics	1756	1080	140	90	160	80	140	60	74	74	42	42	44	44	44	44	43	43
mpgdec-initdec	3828	2604	360	242	410	202	360	198	79	100	47	54	48	57	48	57	47	54
mpgenc-vhfilter	7112	3724	1772	190	1670	190	1670	190	265	628	142	100	154	98	154	98	141	97
mp3-psych	4410	4828	2840	3084	3020	3192	2862	2950	192	350	132	214	135	223	135	223	130	219
mp3-align	17062	16092	2150	1162	1902	1104	2158	1162	121	126	69	54	71	56	71	56	69	54
mpgenc-idct	2960	1936	370	212	370	192	410	200	87	108	53	61	54	62	54	62	52	59
mpgdec-vhfilter	1126	484	270	60	250	60	250	60	155	252	82	68	84	70	84	70	79	68

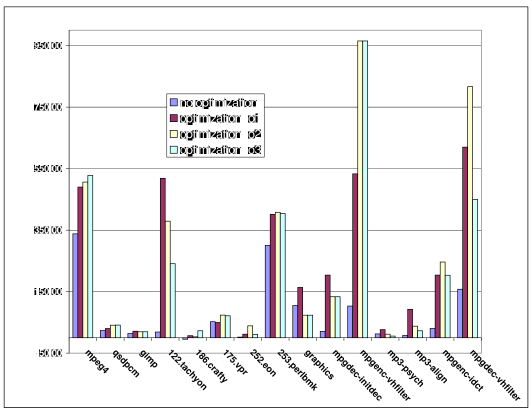

Figure 8: Effect of transformation on time for Intel

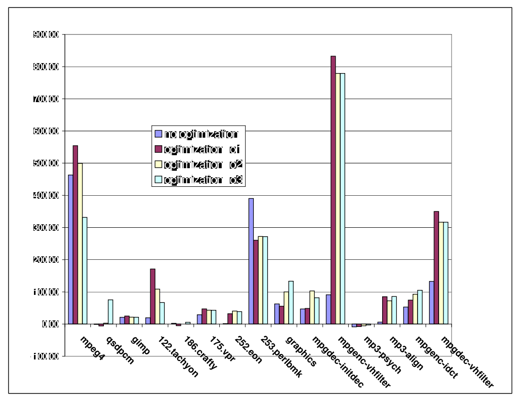

Figure 10: Effect of transformation on time for PowerPC

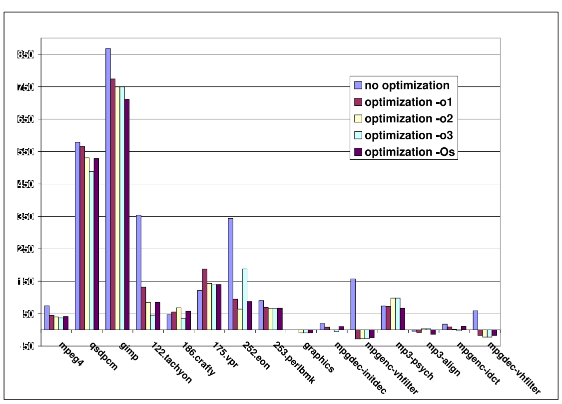

Figure 9: Effect of transformation on code size for Intel

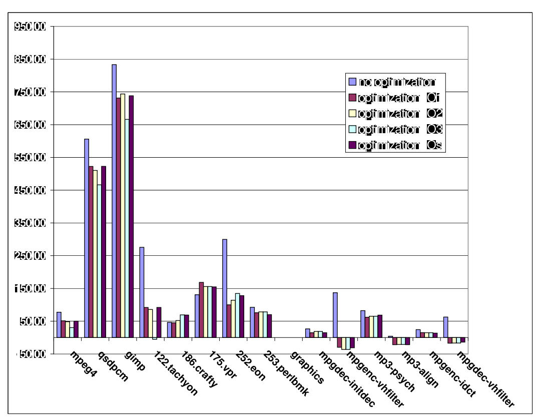

Figure 11: Effect of transformation on code size for PowerPC