Des Autom Embed Syst (2009) 13: 193–221
DOI 10.1007/s10617-009-9043-5

Optimizing control flow in loops using interval and dependence analysis

Mohammad Ali Ghodrat · Tony Givargis · Alex Nicolau

Received: 19 January 2009 / Accepted: 16 June 2009 / Published online: 1 July 2009

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 2.34X speedup when running on a UltraSPARC processor, a 2.92X speedup when running on an Intel Core Duo processor, a 2.44X speedup when running on a PowerPC G5 processor and a 2.04X speedup when running on an ARM9 processor. Performance improvement, taking the entire application into account, was also promising: for 3 selected applications *(mpeg-enc, mpeg-dec and qsdpcm)* we measured 15% speedup on best case (5% on average) for the whole application.

Keywords Interval analysis · Compiler loop optimization · Algorithmic code transformation · Control flow optimization

M.A. Ghodrat (✉) · T. Givargis · A. Nicolau
Department of Computer Science, University of California, Irvine, CA, USA
e-mail: mghodrat@uci.edu

T. Givargis
e-mail: givargis@uci.edu

A. Nicolau
e-mail: nicolau@uci.edu

1 Introduction

Software is becoming a larger fraction of engineering effort. 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, 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 [16]. In our case, the additional compiler execution time was of the order of 10 mili seconds per loop [3].

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 are analyzed using a static *interval analysis* technique previously outlined in [3]. 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 [14].

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

Using interval analysis [3] we statically compute information as shown in the table of Fig. 1 on the conditional expression in the loop nest $((SQR(dist_x) + SQR(dist_y)) < THRESHOLD)$. For example the 2nd row of this table 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 $(SQR(dist_x) + SQR(dist_y)) <$

Fig. 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`

THRESHOLD evaluates to `false`. The transformed code, using the 2nd row of table yields the optimized code shown below:

```
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 Sect. 5.

The rest of this paper is organized as follows. In Sect. 2, we outline the related work. In Sect. 3, we formulate the problem, show the overall flow of the proposed transformation and establish some preliminaries. In Sect. 4, we establish the transformation technique. In Sect. 5, we show our experimental results. In Sect. 6, 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

Table 1 Properties which are being compared in Table 2

Property 1	Optimize control flow 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 affine 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

with control flow optimizations to further improve the generated code (i.e., data-flow optimizations may benefit from simpler control flow within loops).

Table 1 provides a set of properties that are used to compare and contrast loop optimization strategies using control flow analysis. Furthermore, Table 2 summarizes existing loop transformation techniques and provides an analysis of their strength relative to the presented work.

Among all the techniques listed in Table 2, the three most relevant ones are *loop unswitching*, *index-set splitting* and *loop nest splitting*.

Loop unswitching [9], 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 [15], 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. [2]. 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 2a shows a case in *gimp* [14] 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 2b shows a case in *mp3* benchmark [1] 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 Fig. 2c. Figure 2c shows a case in which our technique (Fig. 2e) has remove the condition completely resulting in significant (30% on SPARC and 68% on Intel) speedup while their technique (Fig. 2d) has only partially eliminated the evaluation of the conditional

Table 2 Comparison with other loop optimization techniques

Optimization technique	Summary of technique	Property (Table 1)							
		1	2	3	4	5	6	7	8
Loop fusion	Fuse two adjacent countable loops with the same loop limits		✓						
Loop fission	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 first 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 [2]	For a nested loop, by using polytope model and genetic algorithms, conditions having no effect on control flow are removed or moved up in loop nest	✓		✓	✓	✓			
Our work	For a nested loop, by using interval analysis technique [3] and dependence analysis, the nested loop is partitioned into multiple loops with the no condition	✓	✓	✓	✓	✓	✓	✓	✓

expression. A similar example *186.crafty* from SPEC-2000 [10] is shown in Fig. 2f where applying the technique in [2] will not remove the conditions completely.

3 Proposed transformation

The proposed transformation decomposes the original nested loops of Fig. 3a into three parts, as shown in Fig. 3b. 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.

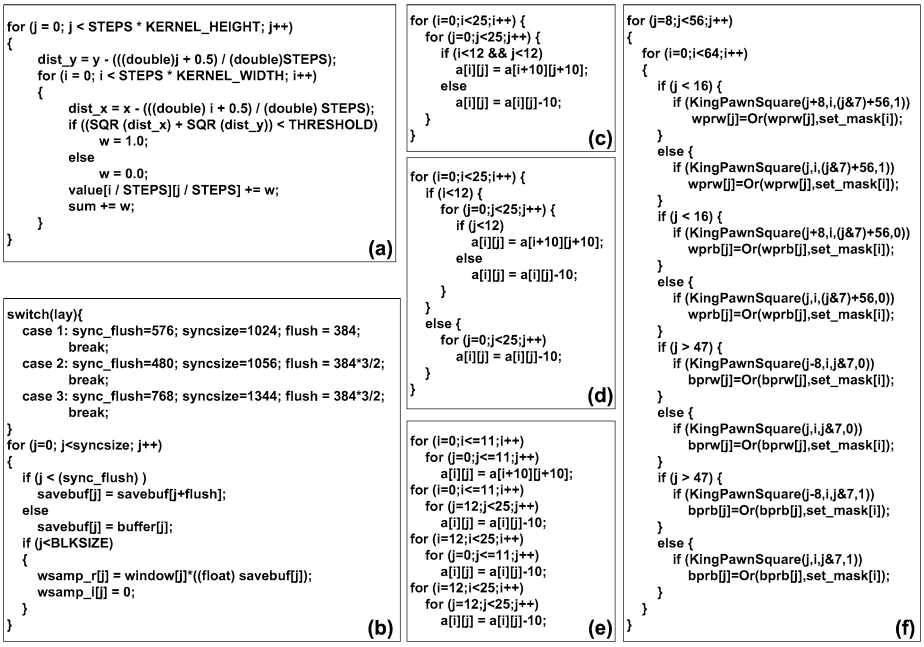

Fig. 2 (**a**) 1st difference—gimp (**b**) 2nd difference—mp3 (**c**–**e**) 3rd difference—synthetic example (**f**) 3rd difference—186.crafty

Our proposed transformation targets loops that follow the normalized template shown in Fig. 3a. Here, there are n loop nests, with n indices $x_1, \ldots, x_n$. 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 re-written in the normalized form of Fig. 3a [9]. Here, st_{cond_expr} computes the value of the branch condition v.

3.1 Preliminaries

In this subsection we summarize the analysis technique developed in [3] 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 [3], 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 com-

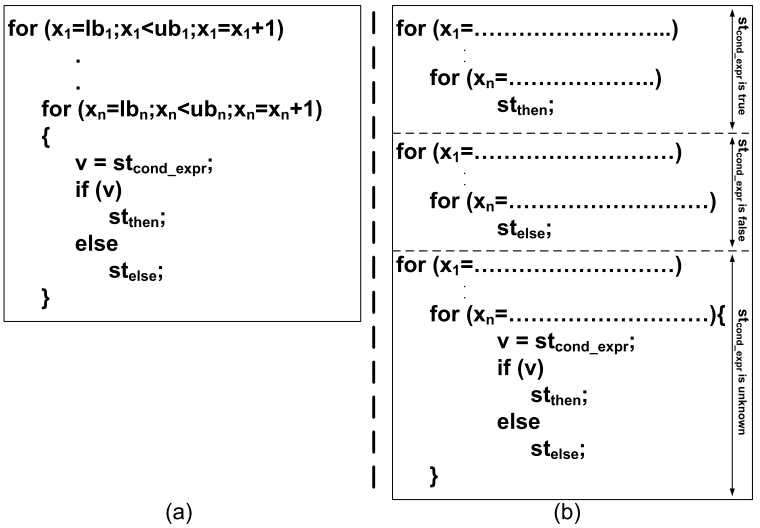

Fig. 3 Transformation

plex 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 [8] for a full coverage of interval arithmetic.

We define an n-dimensional space to be a box-shaped region defined by the Cartesian product $[l_0, u_0] \times [l_1, u_1] \times \cdots \times [l_{n-1}, u_{n-1}]$. Hence, for a given program with n input integer-variables $x_0, x_1, \ldots, x_{n-1}$, the *program domain space* is an n-dimensional space defined by the Cartesian product $[min_0, max_0] \times [min_1, max_1] \times \cdots \times [min_n, max_n]$, 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 $max_i = 127$).

4 Technical approach

We now begin to describe the technique proposed in this paper. A candidate loop L has the structure shown in Fig. 3a. The *iteration space* of L is defined as $[lb_1, ub_1] \times [lb_2, ub_2] \times \cdots \times [lb_n, ub_n]$. The body of L can be decomposed into the *reducible* 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 $x_1, x_2, \ldots, x_n$ 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} (Sect. 4.1).
- Compute the dependence vector of iteration space (Sect. 4.2).
- Partition the iteration space (Sect. 4.3).
- Generate code (Sect. 4.4).

Given the conditional expression *cond_expr* with variables $x_1, x_2, \ldots, x_k$, the domain space partitioning problem [3] is to partition the domain space of *cond_expr* into a mini-

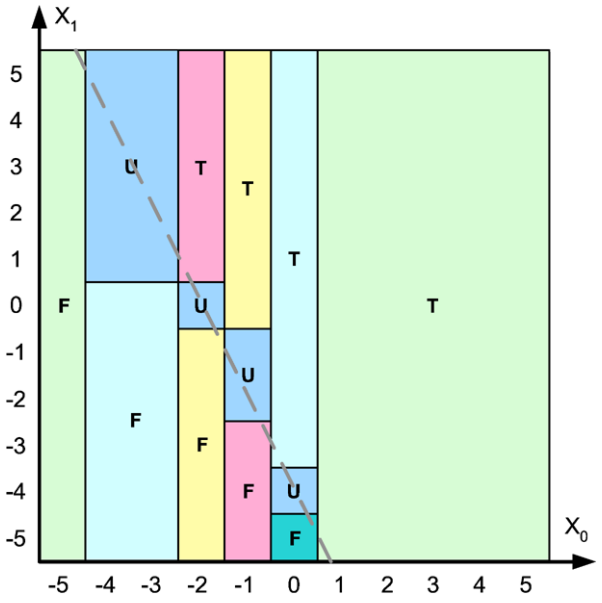

Fig. 4 Partitioned domain of $2x_0 + x_1 + 4 > 0$

mal set of k-dimensional spaces $s_1, s_2, \ldots, s_n$ 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 4 shows the partitioned domain space and the corresponding Boolean values [3].

4.1 Interval set computation

In the following discussion, the code segment presented in Table 3 is used to demonstrate the interval_set computation. In Table 3, 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 Sect. 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 3). 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.

Table 3 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: $v = 1$;	$[1, 1]$	$(x_1 > 0 \&\& x_2 > 0)$	$[1, 10] \times [1, 5]$

- 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 3). 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 3 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 3). For each conditional expression *cond_expr*, there exists a set of spaces $S_1, S_2, \ldots, S_k$ that collectively defines the part of the domain space for which *cond_expr* evaluates to `true`. For example, line 4 of Table 3 shows the conditional expression $(x_1 > 0)\&\&(x_2 > 0)$ defined as $[1, 10] \times [1, 5]$.

Formally, for a variable v, the *interval_set* (i.e., *v.iset*) is defined as $\{(I_j, S_j) | j \in (1 \ldots m)\}$, 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 reducible 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 [3].
(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:

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

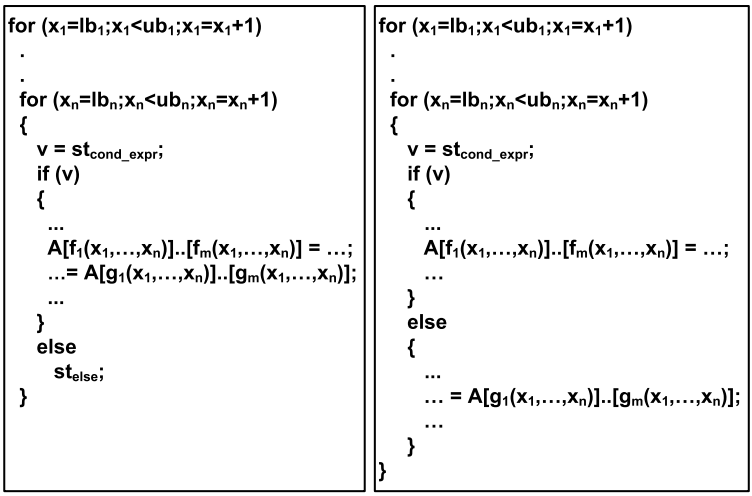

Fig. 5 General memory access model

Furthermore, we define three sets of spaces:

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

For the example of Table 3, 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 at least one of the statements st_1 and st_2 write 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 and at least one of these accesses is a write. 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 $I = \{i_1, \ldots, i_n\}$ of integers showing the corresponding values of the loop indices. If there is a data dependency between statement st_1 during iteration $I = \{i_1, \ldots, i_n\}$ and statement st_2 during iteration $J = \{j_1, \ldots, j_n\}$, then the *dependence vector* is defined as $J - I = \{j_1 - i_1, \ldots, j_n - i_n\}$.

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 5 shows our general m-dimensional memory access model. Figure 5a shows the case when both statements access an array during the execution of the *then* part. Figure 5b 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.

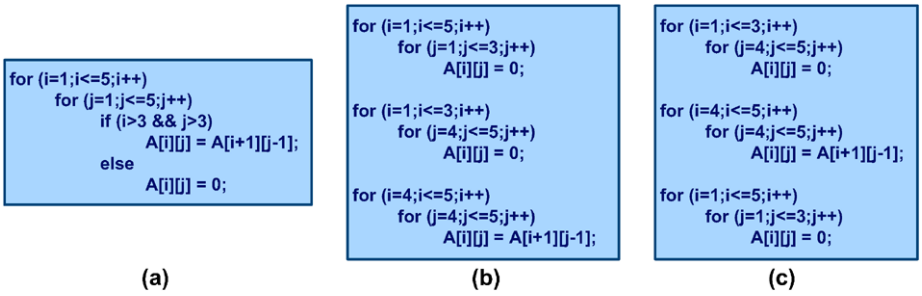

Fig. 6 (**a**) Original code; (**b**) Wrong transformed code; (**c**) Correct transformed code

In the case of Fig. 5a, there exists a data dependence if there are two iteration vectors I and J such that:

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

In the case of Fig. 5b, there exists a data dependence if there are two iteration vectors I and J such that:

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

In the case that both of the accesses are in the *else* part, then $st_{cond\_expr}(I)$ and $st_{cond\_expr}(J)$ in (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 (2).

4.2.1 Example

Before going over the next step of our methodology, an example will be presented which shows the importance of the dependence vector computation and how it effects the result of iteration space partitioning if it is ignored. Intuitively, what we need to ensure is that the order of dependent statements from different spaces is preserved by the execution order of the transformed loops. That is, a space (loop) S_i has to execute before space (loop) S_j if there are dependences between statements in S_i and statements in S_j—e.g., if a statement in S_i produces a values used in S_j. As an example, consider the simple code segment shown in Fig. 6a.

The result of domain space partitioning for the expression $(i > 3 \&\& j > 3)$ is given in Fig. 7. The dependence vector for this code segment is $[1, -1]$ and is shown in Fig. 8.

If the code is transformed without considering the dependence vector, then we may write code for each space shown in Fig. 7 in any order. For example Fig. 6b shows one such transformed code. This code is generated in the incorrect order $[1, 5][1, 3]$, $[1, 3][4, 5]$, $[4, 5][4, 5]$ yielding errornous results when executed. To see why the ordering is incorrect, is illustrated by an example: $A[4][4]$ depends on $A[5][3]$ (write after read) and by generating code for the space $[1, 5][1, 3]$, we are violating this dependence. But considering the spaces using the order $[1, 3][4, 5]$, $[4, 5][4, 5]$, $[1, 5][1, 3]$ will generate correct code as shown in Fig. 6c.

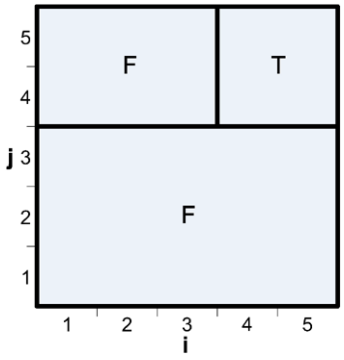

Fig. 7 Domain space partitioning ($i > 3 \&\& j > 3$)

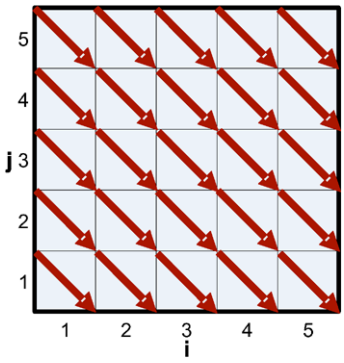

Fig. 8 Dependence vector for the example

4.3 Iteration space partitioning

Recall that sets T, F and U were computed according to Sect. 4.1. Likewise, the dependence vector was computed in Sect. 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 $(S_1, S_2, \ldots, S_p)$ 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 $\forall k \in (1, \ldots, m) f_k(i_1, \ldots, i_n) = g_k(i_1, \ldots, i_n)$ in Fig. 5) 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 Fig. 5 can be expressed as:

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

Fig. 9 Sort the spaces using the dependence vector

```
1:  Input: T, F, U
2:  Input: dependency_vector = {β1,1 − β1,2, ..., βn,1 − βn,2}
3:  Output: Sorted{T, F, U}
4:  relationSet ← φ
5:  for all Spaces Si ∈ {T, F, U} do
6:      expanded_space ← expandSpace(Si, dependency_vector)
7:      overlapped_spaces ← findOverlap(expanded_space)
8:      for all Spaces Sj ∈ overlapped_spaces do
9:          relationSet ← relationSet ∪ (Si < Sj)
10:     end for
11: end for
12: sortedSpaces ← RelationalSort(relationSet, {T, F, U})
13: return(sortedSpaces)
```

Fig. 10 Relational sort

```
1:  Input: T, F, U
2:  Input: relationSet
3:  Output: Sorted{T, F, U}
4:  sortedList ← φ
5:  for all Relation rk = (Si < Sj) ∈ relationSet do
6:      if (Si ∉ sortedList) and (Sj ∉ sortedList) then
7:          sortedList.push(Si)
8:          sortedList.push(Sj)
9:      else if (Si ∈ sortedList) and (Sj ∉ sortedList)
        then
10:         iindex ← sortedList.find(Si)
11:         sortedList.insert(Sj, iindex)
12:     else if (Si ∉ sortedList) and (Sj ∈ sortedList)
        then
13:         jindex ← sortedList.find(Sj)
14:         sortedList.insert(Si, jindex − 1)
15:     else
16:         iindex ← sortedList.find(Si)
17:         jindex ← sortedList.find(Sj)
18:         if iindex >= jindex then
19:             sortedList.remove(Si)
20:             sortedList.insert(Si, jindex − 1)
21:         end if
22:     end if
23: end for
24: return(sortedSpaces)
```

If $\forall k\ \alpha_{k,1} = \alpha_{k,2}$ then the dependence vector can be expressed as $\{\beta_{1,1} - \beta_{1,2}, \ldots, \beta_{n,1} - \beta_{n,2}\}$. 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 $\{\beta_{1,1} - \beta_{1,2}, \ldots, \beta_{n,1} - \beta_{n,2}\}$ we are interested in $p = |T| + |F| + |U|$ sorted spaces $(S_1, S_2, \ldots, S_p)$ in a way that there is no loop-carried data dependency from S_i to S_j if $i < j$.

Algorithm shown in Fig. 9 shows the proposed solution for Problem 2. This algorithm first expand the boundaries of all the spaces using the dependence vector (line 6). It 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 in Fig. 10 is used as a subroutine to sort the spaces (line 12).

Algorithm shown in Fig. 10 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).

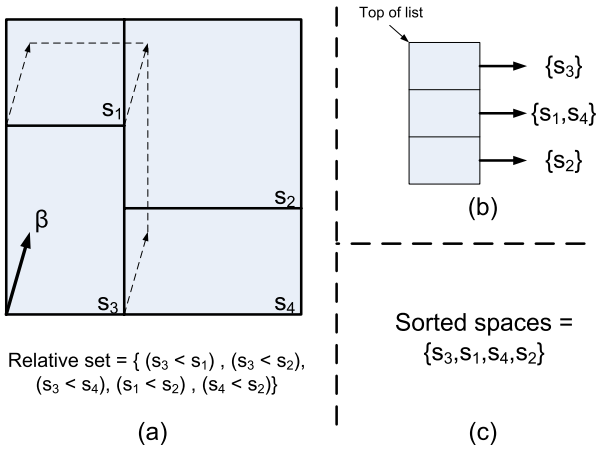

Fig. 11 Example run of Algorithms 9 and 10

Figure 11 shows an example run of algorithms in Figs. 9 and 10. Figure 11a 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 9 on all the spaces. Figure 11b shows the result of executing Algorithm 10 on the relative set shown in Fig. 11a and finally Fig. 11c shows the sorted spaces under the dependency vector β.

4.4 Code generation

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

$$
\begin{array}{l}
\text{for}(x_1 = l_1; x_1 \leq u_1; x_1 + +) \\
\quad \text{for}(x_2 = l_2; x_2 \leq u_2; x_2 + +) \\
\quad\quad \vdots \\
\quad\quad\quad \text{for}(x_n = l_n; x_n \leq u_n; x_n + +) \\
\quad\quad\quad\quad \texttt{body}
\end{array}
$$

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 $S_i \in U$.

5 Experiments

To evaluate the proposed code transformation technique, several *loop kernels* from *MediaBench* [7] application suite and *SPEC-2000* [10] were chosen. We also experimented with an *mp3* encoder implementation obtained from [1], an *mpeg4* full motion estimation obtained from [2], GNU Image Manipulation Program (*gimp*) [14] and also *qsdpcm* [11] 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 4. In Table 4 *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 4, *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 (Sect. 4) at the source level to each of the chosen benchmarks by hand (except the first step in Sect. 4 which is automated), compiled the original and the transformed code, and measured the improvement. We did this experiment for four types of instruction sets: SPARC, x86, PowerPC and ARM. For all the instruction sets, we measured the speedup together with code size increase.

Each loop kernel (original and transformed) was compiled using different optimization levels of *gcc* [13], 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 original loop kernel to the time to execute the optimized 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 has been calculated using the following formula: $Speedup = (T_o / T_n)$. Each bar in Figs. 12, 14, 16 and 18 shows the speedup after applying our code transformation. For each benchmark there are 5 bars, the first 4 representing the speedup for 4 cases of optimizations mentioned above, the fifth bar gives the average speedup. Likewise we have calculated the code size ratio, which is the transformed code size divided by the original code size.

5.1 SPARC

The results of experiments on SPARC are summarized in Table 5. The first half of Table 5 shows the result of measured time before and after transformation for 4 different optimization options. The second half of Table 5 shows the result of code size before and after transformation for the same 4 optimization options plus another optimization for code size (*-Os*). The speedup and the code size ratio have been shown graphically in Figs. 12 and 13.

The experiments were run on a Sun workstation, with two 1503 MHz SUNW, UltraSPARC-IIIi CPU's 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 6.58X. On average, we observed application speedup of 2.34X. On average we observed 2.51X increase on code size.

Note that there are cases where we measured decrease in code size (e.g., *B9* or *B15*), this is due to removal of the conditional expression evaluation from the code combined with the small number of partitions that are generated. The same result is observed X86, PowerPC and ARM as shown in the following sections.

5.2 Intel x86

The results of experiments on Intel x86 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 speedup and the code size ratio have been shown graphically in Figs. 14 and 15.

The experiments were run on a MacBook with a Intel Dual Core 1.8 GHz and 1 GB of memory. We used GCC compiler version 3.4.1 in order to generate executables. In the

Table 4 Selected application list

Benchmark#	Application	Function desc.	Conditional expressions	Properties (Table 1)							
				1	2	3	4	5	6	7	8
1	mpeg4	Motion estimation	$(x3 < 0 \| x3 > 35 \| y3 < 0 \| y3 > 48)$ $(x4 < 0 \| x4 > 35 \| y4 < 0 \| y4 > 48)$	✓	✓	✓	✓	✓			✓
2	qsdpcm	Motion estimation	$((4 * x + vx - 4 + x4 < 0) \|$ $(4 * x + vx - 4 + x4 > (N/4 - 1)) \|$ $(4 * y + vy - 4 + y4 < 0) \|$ $(4 * y + vy - 4 + y4 > (M/4 - 1)))$	✓	✓	✓	✓	✓			✓
3	gimp	Create Kernel	$(32 * x - 2 * i + 1)^2 + (32 * y - 2 * j + 1)^2 < 4096$	✓	✓	✓				✓	✓
4	122.tachyon (SPECMPI-2007)	Parallel ray tracing (Generate Noise Matrix)	$(x == NMAX - 1)$, $(y == NMAX - 1)$, $(z == NMAX - 1)$	✓	✓	✓	✓	✓			✓
5	186.crafty (SPEC-2000)	Chess program (Generate Piece Masks)	$(j < 16)$, $(j > 47)$	✓	✓	✓	✓				✓
6	175.vpr (SPEC-2000)	FPGA Circuit Placement and Routing (Check architecture file)	$i! = 4 \&\& i! = DETAILED\_START + 5 \&\&$ $i! = 5 \&\& i! = DETAILED\_START + 6$	✓	✓	✓	✓				✓
7	252.eon (SPEC-2000)	Computer Visualization	$(i == 0)$, $(j == 0)$, $(k == 0)$	✓	✓	✓	✓				✓

Table 4 (*Continued*)

Benchmark#	Application	Function desc.	Conditional expressions	Properties (Table 1)							
				1	2	3	4	5	6	7	8
8	253.perlbmk (SPEC-2000)	PERL Programming Language	$((c >=' A' \&\& c <=' Z') \|\| (c >=' a' \&\& c <=' z') \|\| (c >=' 0' \&\& c <=' 9') \| c ==' \_')$	✓	✓	✓	✓	✓			✓
9	Synthetic graphics	Collision detection	$(x * x + y * y == x * x * y)$	✓	✓	✓				✓	✓
10	mpgdec-initdec	Initialize Decoder	$(i < 0), (i > 255)$	✓	✓	✓	✓				✓
11	mpgenc-vhfilter	Ver./Hor. Filter,2:1 Subsample	$(i < 5), (i < 4), (i < 3), (i < 2), (i < 1)$	✓	✓	✓	✓				✓
12	mp3-psych	Layer 3 Psych. Analysis	$j < sync\_flush$, $j < BLKSIZE$	✓	✓	✓	✓		✓		✓
13	mp3-align	Read and align audio data	$j < 64$	✓	✓	✓	✓				✓
14	mpgenc-idct	IDCT Initialize	$(i < -256), (i > 255)$	✓	✓	✓	✓				✓
15	mpgdec-vhfilter	Ver./Hor. Interpolation Filter	$(i < 2), (i < 1)$	✓	✓	✓	✓				✓

Table 5 Result of experiments for sparc-time and code size (*shaded*: original; *white*: transformed)

Benchmark	Time (original and transformed)								Code size (original and transformed)									
	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-vhfilter	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-vhfilter	2034	900	552	94	582	96	582	96	157	262	82	76	71	64	71	64	66	60

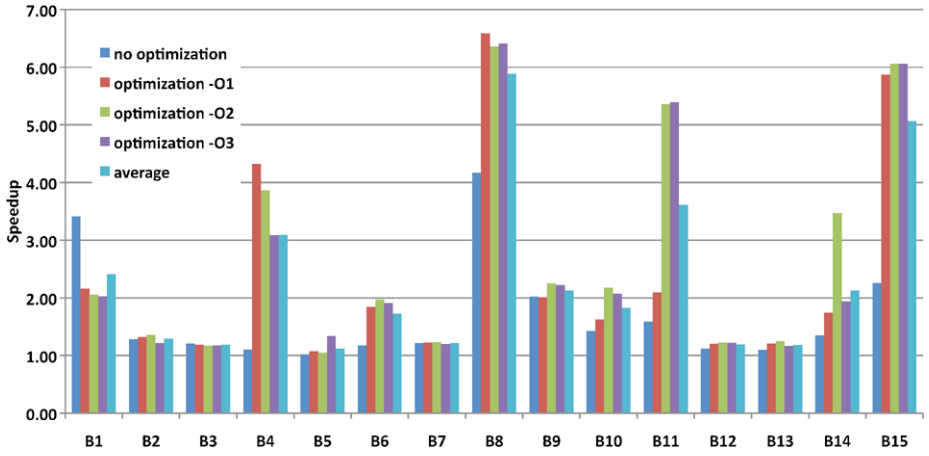

Fig. 12 Effect of transformation on time for SPARC

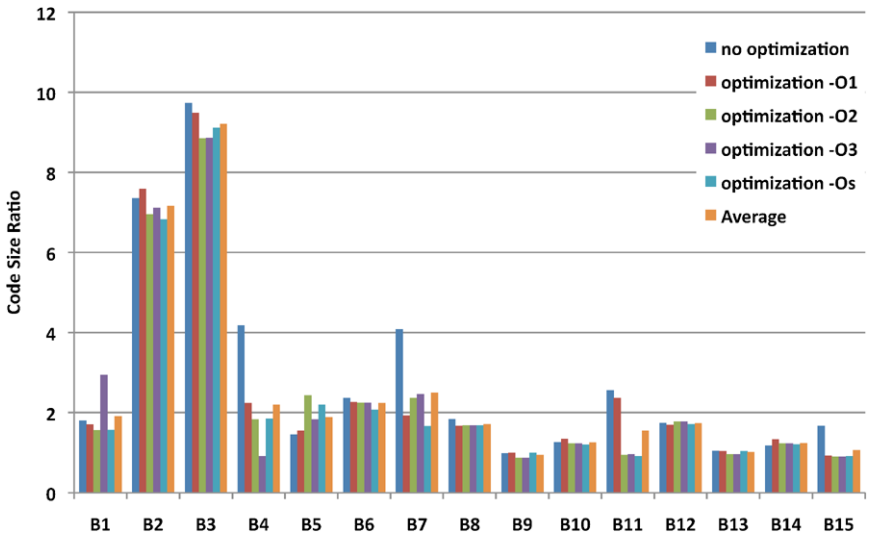

Fig. 13 Effect of transformation on code size for SPARC

best case, we observed application speedup of 10.65X. On average, we observed application speedup of 2.92X. On average we observed 2.34X increase on code size.

One interesting point here (and also for SPARC in Fig. 12) is the huge constant speedup for benchmark B4 with -O1, -O2 or -O3 optimization. In this benchmark there is an access to a 3 dimensional array. In the non-optimized version the address to the beginning of the array is always computed for every access inside loop. In the optimized version (-O1, -O2 and -O3), the beginning of the array computation is hoisted out of the loop and is kept in a register, yielding a huge performance gain. Finally, the timing result for all -O1, -O2 and -O3 is almost the same as can be seen in Tables 5 and 6.

Table 6 Result of experiments for intel x86-time and code size (*shaded*: original; *white*: transformed)

Benchmark	Time (original and transformed)								Code size (original and transformed)									
	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-vhfilter	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-vhfilter	438	170	216	30	110	12	110	20	136	216	99	82	103	80	103	80	97	80

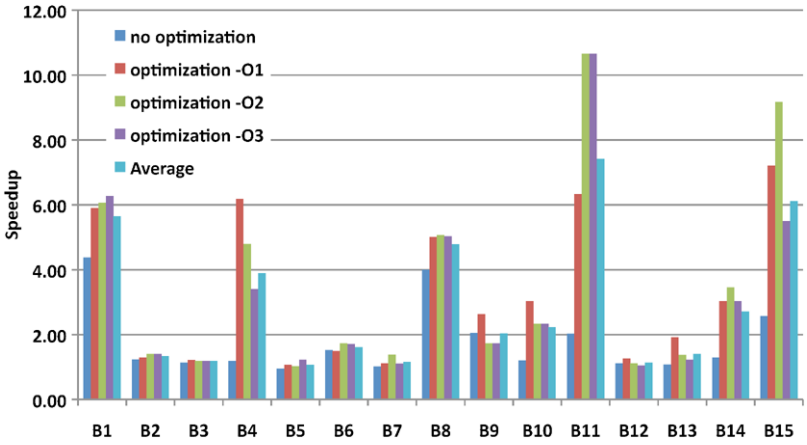

Fig. 14 Effect of transformation on time for x86

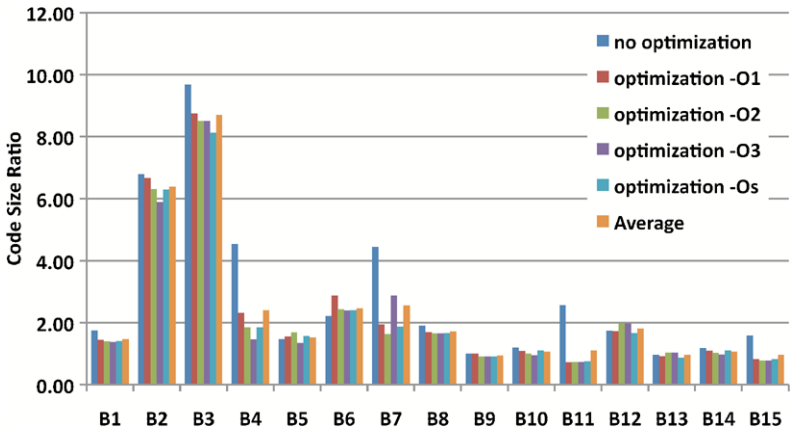

Fig. 15 Effect of transformation on code size for x86

5.3 PowerPC

The results of experiments on PowerPC 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 speedup and the code size ratio have been shown graphically in Figs. 16 and 17. 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 9.33X. On average, we observed application speedup of 2.44X. On average we observed 2.31X increase on code size.

Table 7 Result of experiments for PowerPC-time and code size (*shaded*: original; *white*: transformed)

Benchmark	Time (original and transformed)								Code size (original and transformed)									
	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

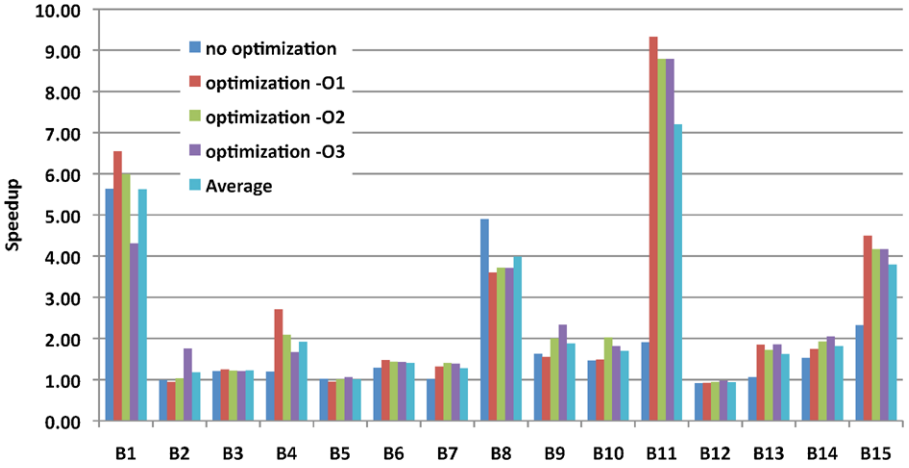

Fig. 16 Effect of transformation on time for PowerPC

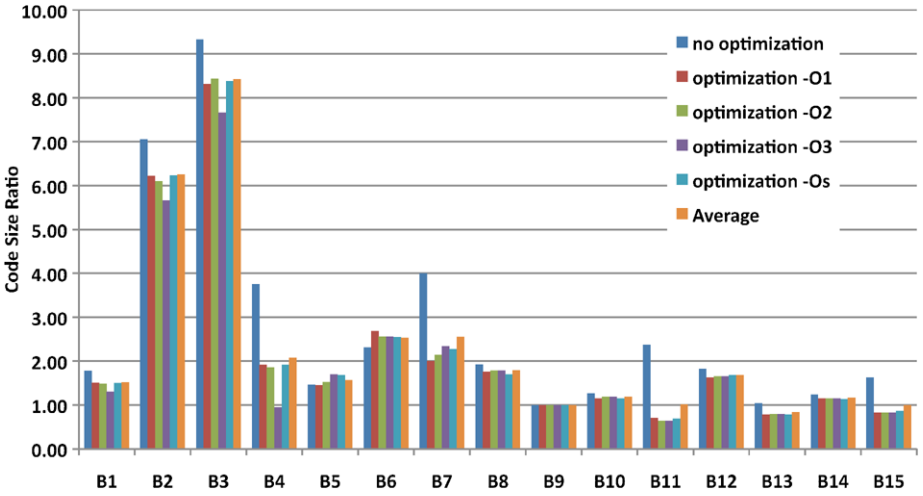

Fig. 17 Effect of transformation on code size for PowerPC

5.4 ARM

The results of experiments on ARM 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 speedup and the code size ratio have been shown graphically in Figs. 18 and 19.

The experiments were run on an ARM evaluation board TS-7250 from Technologic Systems [12] with a 200 MHz ARM9 which its characteristics are shown in Table 9. We used *arm-linux-gcc* compiler version 3.4.4 in order to generate executables. In the best case, we observed application speedup of 5.77X. On average, we observed application speedup of 2.04X. On average we observed 3.23X increase on code size.

Table 8 Result of experiments for ARM—time and code size (*shaded*: original; *white*: transformed)

Benchmark	Time (original and transformed)								Code size (original and transformed)									
	No		-O1		-O2		-O3		No		-O1		-O2		-O3		-Os	
mpeg4	2426	637	1380	476	1265	458	1278	457	335	2817	174	1287	187	1291	186	1243	162	1108
qsdpcm	1660	1402	3533	2950	3460	2750	3236	2513	232	2641	119	1252	139	1436	147	1545	108	1106
gimp	8523	7028	5922	4604	6329	4947	6329	4956	203	2168	105	979	102	956	101	927	118	906
122.tachyon	2760	2350	9400	5766	8566	5733	8566	5766	167	761	71	216	78	193	78	187	77	184
186.crafty	1746	1696	7366	6833	7900	7300	7866	7300	173	321	98	180	102	183	102	183	96	172
186.crafty	1526	1453	5533	5200	5333	5100	3233	2800	304	446	171	255	118	302	222	442	110	304
300.twolf	1410	1403	1190	1190	1356	1300	1356	1300	70	75	37	42	37	41	37	41	40	49
175.vpr	3700	3500	2666	2500	2700	2500	2533	2400	422	1400	232	743	200	633	257	1114	195	626
175.vpr	7166	5266	3966	3000	3800	3000	3600	3000	111	249	57	123	68	156	68	156	60	125
253.perlbmk	1137	283	4386	913	5023	873	5023	870	100	196	46	82	47	83	46	82	45	81
graphics	1758	879	6080	3040	4666	2330	4666	2333	74	73	35	34	38	33	38	33	30	30
mpgdec-initdec	3600	2033	1400	696	1196	660	1200	660	68	92	35	44	33	41	33	41	29	38
mpgenc-vhfilter	9993	4980	2680	1390	2600	716	2590	713	274	713	112	267	94	104	94	104	86	86
mp3-psych	6310	7030	6113	6863	6150	10130	N/A	N/A	169	322	130	227	120	220	115	220	111	209
mp3-align	2850	2800	2656	2626	3396	3373	2470	2446	123	128	57	58	61	57	61	57	56	56
mpgenc-idct	1820	1026	7066	3633	6066	3400	6066	3400	72	96	40	50	38	46	38	46	34	43
mpgdec-vhfilter	3030	1280	6500	1900	7100	1900	7100	1600	146	243	69	71	55	54	55	53	51	49

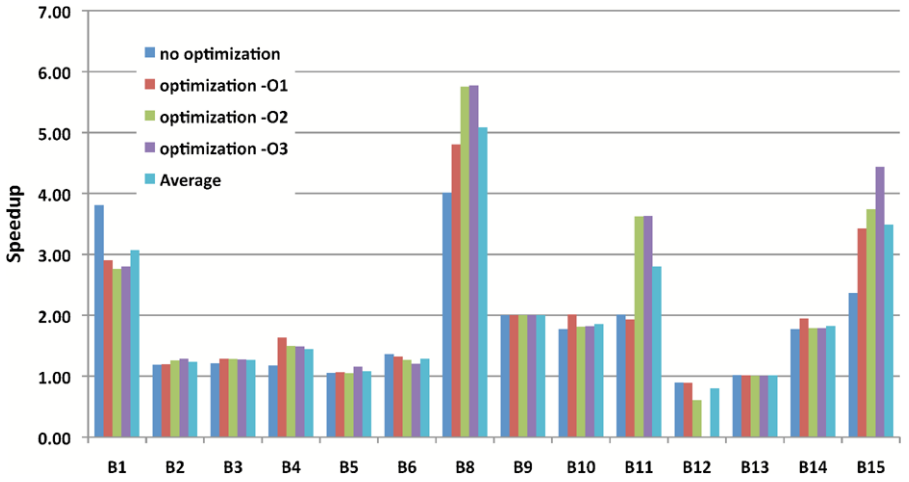

Fig. 18 Effect of transformation on time for ARM

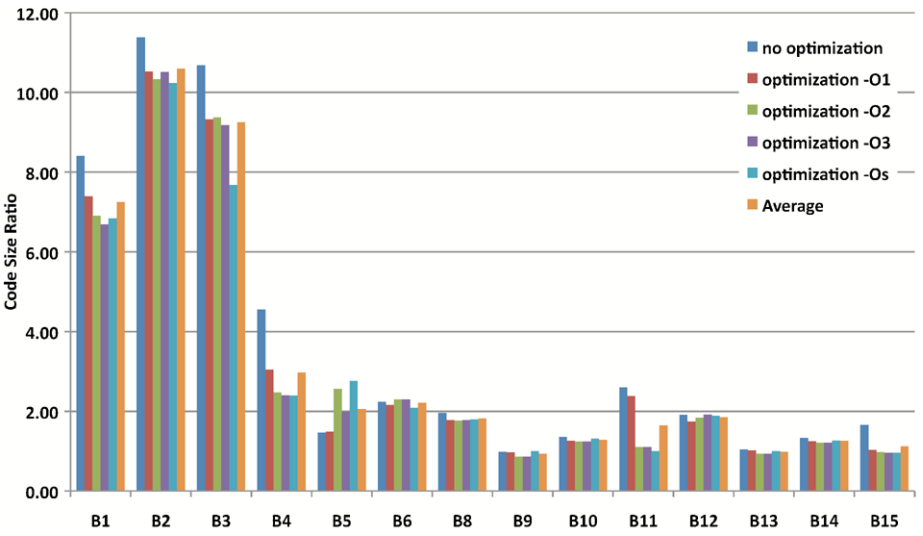

Fig. 19 Effect of transformation on code size for ARM

This technique is not always rewarding as can be seen from the result for *B12* on ARM9 processor (Fig. 18). This is due to high number of memory array accesses inside the original loop. In the original loop, each array access is loaded once through out the loop. When partitioned in a series of loops (after transformation), each loop partition will have its own memory loads, which makes it more expensive compared to the original code.

5.5 Full application speed-up

To investigate the benefit of our proposed transformation on a whole application, we tested our transformation on 3 applications, namely *mpeg decoder*, *mpeg encoder* and *qsdpcm* on

Table 9 ARM9 board specification

200 MHz ARM9 processor with MMU
32 MB of High Speed SDRAM
32 MB Flash disk used for RedBoot boot-loader, Linux kernel and root file system
Linux Kernel 2.6.20
USB Flash drive supported
10/100 Ethernet interface
2 USB 2.0 Compatible OHCI ports (12 Mbit/s Max)

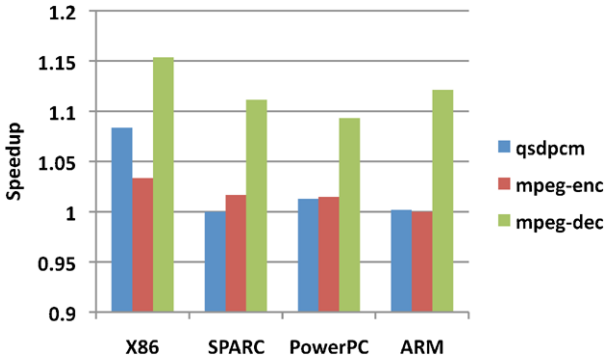

Fig. 20 Full application performance improvement

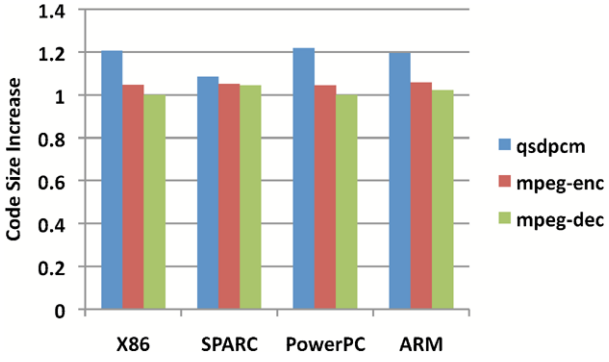

Fig. 21 Full application code size increase

all the 4 previously mentioned instruction sets: *SPARC*, *x86*, *PowerPC* and *ARM*. For this experiment we only used *gcc -O3* for compilation and as before we hand transformed all the places in code which our transformation could be applied. Figure 20 shows the result of the performance improvement for the three mentioned application. In the best case we get 15% speedup for the *mpeg-dec* on X86. On average we get 5% performance improvement. Figure 21 shows the result of the code size increase for the three mentioned application. In the worst case we get 21% code size increase for the *qsdpcm* on PowerPC. On average we get 6% code size increase.

It's worth mentioning that like any other compiler optimization (e.g. most of the compiler optimization techniques mentioned in Table 4), our technique is applicable to certain part of code, which may or may not be part of the hot spots. So, the gain that we can get on the full application speedup varies a lot. Even though, there are compiler optimization techniques

that by enabling some hardware techniques can gain a lot (like all the loop parallelization techniques), no single software-only compiler optimization can give us huge speed-ups on all the benchmarks, when taking the entire application into account.

5.6 Additional remarks

1. 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. This is due to dataflow optimizations which will be enabled because of the removal of the conditional expressions from loop bodies. 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. Or sometimes, it is because more optimization (i.e. -O2, -O3) will not give more benefits comparing to -O1. This was shown for example in *122.tachyon* in both Figs. 12 and 14.

   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 x86 and PowerPC) as shown in Tables 5, 6 and 7.
2. 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.
3. The code size increase reported in Sects. 5.1, 5.2, 5.3 and 5.4 are only for the loop kernels.
4. The domain space partitioning algorithm might produce lots of spaces for a given nested conditional expression. If the loop is partitioned based on all these spaces, then the loops overhead will create a diminish return for performance gain. A similar argument has been presented in [4]. What should be done in these cases is to drop all those spaces which have a size smaller than a constant threshold, in this way they will be merged with *unknown* spaces and the original code will be generated for them in the code generation step.
5. Given profiling information, our method can be applied to a general loop that has variable lower and upper bounds: (1) Separate out the part of domain space with the profiled upper/lower bounds, (2) apply the method presented in this paper and generate optimized code for this carved out space, and finally (3) for the remaining part of the domain space, use the original code. As an example, consider the loop shown here:

```
for (i=lb;i<ub;i++)
{
    if (Cexpr)
        st_then;
    else
        st_else;
}
```

   If profiling information shows that there is an interval $[Lp, Up]$ (Lp and Up are two constant integers) such that $lb \leq Lp \leq Up \leq ub$, we can transform the code as follows:

```
if ( (lb <= Lp) && (Up <= ub) )
{
    for (i=lb;i<Lp;i++) {
        // Original loop body
    }
```

```
        for (i=Lp;i<Up;i++) {
            // Transformed loop body based on the constant values of
               [Lp,Up] and Cexpr
        }
        for (i=Up;i<ub;i++) {
            // Original loop body
        }
    }
    else
    {
        for (i=lb;i<ub;i++) {
            // Original loop body
        }
    }
```

6 Conclusion

Given the stringent design constraints, performance requirements of embedded systems and as software is becoming a larger fraction of engineering effort, 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 2.34X speedup when running on a UltraSPARC processor, a 2.92X speedup when running on an Intel Core Duo processor, a 2.44X speedup when running on a PowerPC G5 processor and a 2.04X speedup when running on an ARM9 processor. In addition to ARM9 which is a representative of an embedded 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). On average, we measured a code size increase of 2.51X for SPARC, 2.34X for Intel x86, 2.31X for PowerPC and 3.23X for ARM. Note that despite the size increase, the overall 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. Performance improvement, taking the entire application into account, was also promising: for 3 selected applications (*mpeg-enc, mpeg-dec and qsdpcm*) we measured 15% speedup on best case (5% on average) for the whole application.

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

References

1. Iso mp3 sources. Available as http://www.mp3-tech.org/programmer/sources/dist10.tgz
2. Falk H, Marwedel P (2003) Control flow driven splitting of loop nests at the source code level. In: Proceedings of DATE, pp 410–415

3. Ghodrat MA, Givargis T, Nicolau A (2005) Equivalence checking of arithmetic expressions using fast evaluation. In: Proceedings of the CASES, pp 147–156
4. Ghodrat MA, Givargis T, Nicolau A (2007) Short-circuit compiler transformation: Optimizing conditional blocks. In: Proceedings of the 12th Asia and South Pacific design automation conference (ASP-DAC 2007), pp 504–510
5. Issenin I, Dutt N (2006) Data reuse driven energy-aware mpsoc co-synthesis of memory and communication architecture for streaming applications. In: CODES-ISSS 2006, pp 294–299
6. Kennedy K, Allen R (2001) Optimizing compilers for modern architectures: A dependence-based approach. Morgan Kaufmann, San Mateo
7. Lee C et al (1997) Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In: International symposium on microarchitecture, pp 330–335
8. Moore RE (1966) Interval analysis. Englewood Cliffs, Prentice-Hall
9. Muchnick SS (1997) Advanced compiler design and implementation. Morgan Kaufmann, San Mateo
10. Standard Performance Evaluation Corporation Spec cpu2000. Available as http://www.spec.org/cpu2000/
11. Stobach P (1988) A new technique in scene adaptive coding. In: Proceedings of EUSIPCO
12. Technologic systems http://www.embeddedarm.com/products/board-detail.php?product=TS-7250
13. The GCC Team. Gnu compiler collection. Available as http://gcc.gnu.org/
14. The GIMP Team. Gnu image manipulation program. Available as http://www.gimp.org/
15. Wolfe M (1995) High-performance compilers for parallel computing. Addison-Wesley, Reading
16. Wolfe M (2005) How compilers and tools differ for embedded systems. In: Proceedings of the CASES, p 1