Generalized Index-Set Splitting

Christopher Barton[1], Arie Tal[2], Bob Blainey[2], and José Nelson Amaral[1]

[1] Department of Computing Science,
University of Alberta, Edmonton, Canada
{cbarton, amaral}@cs.ualberta.ca
[2] IBM Toronto Software Laboratory, Toronto, Canada
{arietal, blainey}@ca.ibm.com

Abstract. This paper introduces *Index-Set Splitting* (ISS), a technique that splits a loop containing several conditional statements into several loops with less complex control flow. Contrary to the classic *loop unswitching* technique, ISS splits loops when the conditional is loop variant. ISS uses an *Index Sub-range Tree* (IST) to identify the structure of the conditionals in the loop and to select which conditionals should be eliminated. This decision is based on an estimation of the code growth for each splitting: a greedy algorithm spends a pre-determined code growth budget. ISTs separate the decision about which splits to perform from the actual code generation for the split loops. The use of ISS to improve a loop fusion framework is then discussed. ISS opportunity identification in the SPEC2000 benchmark suite and three other suites demonstrate that ISS is a general technique that may benefit other compilers.

1 Introduction

This paper describes *Index-Set Splitting* (ISS), a code transformation motivated by the implementation of loop fusion in the commercially distributed IBM XL Compilers. ISS is an enabling technique that increases the code scope where other optimizations, such as software pipelining, loop unroll-and-jam, unimodular transformations, loop-based common expression elimination, can be applied.

A loop that does not contain branch statements is a Single Basic Block Loop (SBBL). A loop that contains branches is a Multi-Basic Block Loop (MBBL). SBBLs are easier to optimize than MBBLs. For instance, MBBLs with complex control flow are not candidates for conventional software pipelining. Loop unswitching is a transformation that can convert a MBBL into two non-control flow equivalent SBBLs by moving a branch statement out of the original loop [1]. Loop unswitching is applicable only to loop invariant branches.

ISS recursively splits a loop with several branches into loops with smaller index ranges and fewer branches. Contrary to loop unswitching, ISS splits loops based on loop variant branches. In order to minimize its impact on compilation time and code growth, ISS performs a profitability analysis to control the number of loops that are generated. ISS is effective in removing branches that are found

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 102–116, 2005.

in the original code as well as branches that are inserted into the code by the compiler.

Loop fusion is a code transformation that may insert branches into a loop. Barton *et al.* list three fusion-preventing conditions that, if present, must be dealt with before two control flow equivalent loops can be fused: (1) intervening code; (2) non-identical loop bounds; and (3) negative distance dependencies between loop bodies [2]. The classical solution to deal with the second and third conditions requires the generation of compensatory code outside of the loops. This compensatory code will contain one or more iterations of the loop. If this code is generated during the loop fusion process, it becomes intervening code between other fusion candidates. This new intervening code has, in turn, to be moved elsewhere. Thus a cumbersome loop fusion code transformation is created.

The proliferation of intervening code during the loop fusion process can be avoided by inserting guard branches within the loops. Guards are conditional statements that prevent a portion of the loop code from being executed on certain iterations. Once the loop fusion process has completed, ISS can be run to remove the guard branches from inside the fused loops, thereby turning a single MBBL into many SBBLs.

The main contributions of this paper are:

- A description of the new index-set splitting technique that selectively eliminates loop variant branches from a loop.
- An example of the use of guards followed by index-set splitting to improve the loop fusion framework.
- An example of the use of index-set splitting to enable other optimizations.
- Measurements indicating the changes caused by ISS in the compilation time of applications in the development version of the IBM XL compiler.
- Run-time measurements indicating the impact of ISS on the performance of the code generated.

The paper is organized as follows. Section 2 presents an example to motivate ISS. Section 3 introduces the Index Sub-range Tree that is used to handle loops with multiple split points. Section 4 describes how code growth is controlled by the ISS algorithm. Section 5 describes how ISS is used to produce a cleaner framework for loop fusion. Section 6 shows the use of guards for run-time bounds checks in loop fusion. These guards are then split points for the ISS algorithm. A discussion of how ISS can be used to enable other optimizations is provided in Section 7. An experimental evaluation of ISS is presented in Section 8.

2 A Motivating Example

The code in Figure 1(a) executes a branch in every iteration of the loop. Although in most modern architectures this branch is likely to be predicted correctly, the execution of the branch requires an additional instruction in each loop iteration and has the potential of disrupting the operation of the execution pipeline. Removing a branch from inside a loop by splitting the loop into two separate,

```
for(i=0; i<100; i++) {
  if(i < m)
    A[i] = A[i] * 2;
  else
    A[i] = A[i] * 5;
  B[i] = A[i]*A[i];
}
```

(a) Original loops

```
for(i=0; i<m; i++) {
  A[i] = A[i] * 2;
  B[i] = A[i]*A[i];
}
for(i=m; i<100; i++) {
  A[i] = A[i] * 5;
  B[i] = A[i]*A[i];
}
```

(b) Incorrect ISS

```
for(i=0; i<min(m,100); i++) {
  A[i] = A[i] * 2;
  B[i] = A[i]*A[i];
}
for(i=max(m,0); i<100; i++) {
  A[i] = A[i] * 5;
  B[i] = A[i]*A[i];
}
```

(c) Correct ISS

```
for(i=lb; i<min(m,ub); i++) {
    A[i] = A[i] * 2;
    B[i] = A[i]*A[i];
   }
for(i=max(m,lb); i<ub; i++) {
  A[i] = A[i] * 5;
  B[i] = A[i]*A[i];
}
```

(d) General code generated by ISS

Fig. 1. Example of application of ISS

control flow equivalent, loops is desirable because it results in a reduction of the number of instructions executed. This splitting also produces loops with simpler control flow that are easier to optimize.

However, the code in Figure 1(b) may produce incorrect results. Consider the case in which `m > 100`. The first loop in Figure 1(b) would execute more iterations than the original loop. A similar problem occurs with the second loop if `m < 0`. Thus the correct transformation must replace these loop bounds by `min(m, 100)` and `max(m, 0)`, respectively, as shown in Figure 1(c). In general, for a loop with lower bound `lb`, upper bound `ub`, and split point `m`, the code shown in Figure 1(d) should be produced. This code transformation is called *Index-Set Splitting* (ISS).

ISS is always safe, *i.e.,* no other condition besides the structure of the loop has to be analyzed. ISS can be applied even when the bounds and the split points are not known at compile time. However, if relations between these values can be discovered at compile time, loops may be eliminated or their bodies may be simplified.

3 Index Sub-Range Tree

When a loop contains two split points, ISS could be applied iteratively. For example, ISS could be applied on the original loop creating two new loops, both containing a single split point. ISS would then be applied to each of the new

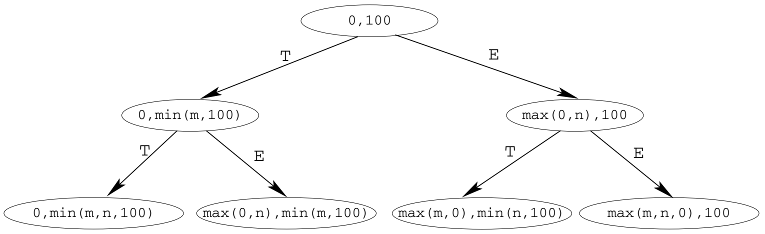


Fig. 2. Index sub-range tree (IST)

loops, creating two new SBBLs. However, iterative ISS would make estimating the potential gain of ISS and controlling the amount of code growth difficult. An alternative solution is to build an *Index Sub-range Tree* (IST). For instance, the following loop contains two split points, `m` and `n`:

```
for(i=0; i < 100; i++) {
    if(i < m)
      A[i] = A[i] * 2;
    else
      A[i] = A[i] * 5;
    B[i] = A[i]*A[i];
  }
```

The IST for the loop above is shown in Figure 2. The root of the IST corresponds to the index range for the original loop. The second level of the tree corresponds to the two loops that are created to eliminate the first test, `(i < m)`, from the loop. If ISS stops at this level of the tree, two loops, each with one branch, are created as shown in Figure 3(a). The nodes in the leaf level in the IST correspond to the four loops that have to be created in order to eliminate all split points, as shown in Figure 3(b).

Edges in the IST labeled with `T` represent the true or "`then`" branch of a test, and edges labeled with `E` represent the "`else`" branch of a test. This labeling is a convenience for the generation of code for the loop representing each node in the tree. The code generation algorithm for a node v_i starts with the original loop code, and traverses the tree from the root to v_i. At each level, if the `then` path is taken, the corresponding branch is eliminated and its `then` code is preserved. If the `else` path is taken, the `else` code is preserved. This process is referred to as the elimination of "dead" inductive branches.

Figure 4 shows the elimination of dead inductive branches to generate the loop body for the leaf node `max(0,n), min(m,100)` in the IST of Figure 2 (the second loop in Figure 3(b)). Starting at the root, to reach this leaf node, the algorithm first follows the `then` path, thus the text `if (i < m)` is eliminated but its `then` code is preserved. At the next level the `else` path is taken. Because the `else` code of the test `if(i < n)` is empty, the entire `if` statement is eliminated.

The IST correctly models nested branches. In the case of a nested branch, the inner level branch only splits the range of the nodes for which they apply. The IST for the loop with nested branch of Figure 5 is shown in Figure 6.

```
for(i=0; i < min(m,100); i++) {
    A[i] = A[i] * 2;
    if(i < n)
      A[i] = A[i] * 5;
    B[i] = A[i]*A[i];
   }
for(i=max(m, 0); i < 100; i++) {
    if(i < n)
      A[i] = A[i] * 5;
    B[i] = A[i]*A[i];
   }
```

(a) Elimination of first split point

```
for(i=0; i < min(m,n,100); i++) {
    A[i] = A[i] * 2;
    A[i] = A[i] * 5;
    B[i] = A[i]*A[i];
   }
for(i=max(0,n); i < min(m,100); i++) {
    A[i] = A[i] * 2;
    B[i] = A[i]*A[i];
   }
for(i=max(m, 0); i < min(n,100); i++) {
    A[i] = A[i] * 5;
    B[i] = A[i]*A[i];
   }
for(i=max(m,n,0); i < 100; i++) {
    B[i] = A[i]*A[i];
   }
```

(b) Elimination of second split point

Fig. 3. Handling loops with multiple split points

```
for(i=max(0,n); i < min(m,100); i++) {
    ~~if(i < m)~~
       A[i] = A[i] * 2;
    ~~if(i < n)~~
       ~~A[i] = A[i] * 5;~~
    B[i] = A[i]*A[i];
   }
```

Fig. 4. Elimination of dead inductive branches

```
for(i=0; i < 100; i++) {
    if(i < m)
       A[i] = A[i] * 2;
    else
       if(i < n)
          A[i] = A[i]*5;
    B[i] = A[i]*A[i];
   }
```

Fig. 5. A loop with nested branches

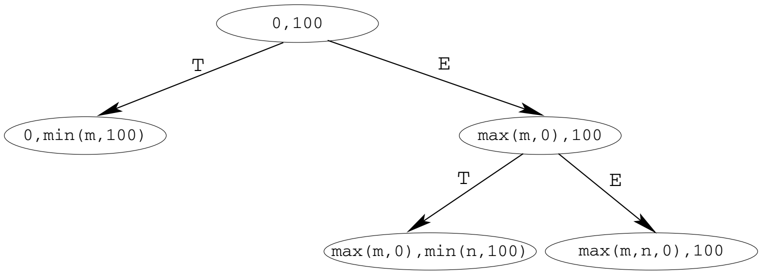


Fig. 6. Index sub-range tree for nested branches

4 Controlling Code Growth

Each index splitting requires the duplication of the loop that it splits. Therefore, there is a potential for significant code growth. If this code growth is left unchecked it may (1) prohibitively slow down the compiler by consuming compilation time that would be put to better use elsewhere and (2) generate negative instruction cache effects at run time.

To control code growth the ISS algorithm marks the root of the sub-range tree with the code size estimate for the original loop. The code size estimate is based on the number of machine instructions that would have been generated for the loop being analyzed. Each node of the subtree is annotated with an estimate of the code size that would be produced by ISS. This estimate is based on doubling the size of the loop at the current level and subtracting the code that is removed from each loop because of the splitting.

In the resulting IST each node is annotated with a code size estimate for its children. The ISS is a greedy algorithm that executes a top-down breadth first traversal of this annotated tree until either all the leaves are processed or a specified code growth budget is consumed. If the budget is exhausted, the lowest nodes that were visited in each branch of the tree represent the loops that are generated by ISS.

5 Applying ISS to Loop Fusion

A loop is normalized if it has a lower bound of 0, and an increment of 1. Thus all normalized loops have the same lower bound, increment, and direction (both loops increase their indexes). If L_i and L_j are normalized and their upper bounds are not the same, the loops are *non-conforming.* Non-conforming loops can be fused if iterations are peeled from the longer loop. However peeling iterations from a loop is not desirable in a loop fusion framework because the peeled iterations may become intervening code that, in turn, has to be moved to allow future loop fusions. For instance, to fuse loops L1 and L2 of Figure 7(a), two iterations of L2 have to be peeled as shown in Figure 7(b). Once L1 and L2 are fused (forming L4) the code for the peeled iterations becomes intervening code

```
L1: for(i=0; i<n-2; i++)
       A[i] = A[i] * 2;
L2: for(j=0; j<n; j++)
       A[j] = A[j] + 3;
L3: for(k=0; k<n-2; k++)
       A[k] = A[k] - 5;
```

(a) Original loops

```
L1: for(i=0; i<n-2; i++)
       A[i] = A[i] * 2;
L2: for(j=0; j<n-2; j++)
       A[j] = A[j] + 3;
     A[n-2] = A[n-2] + 3;
     A[n-1] = A[n-1] + 3;
L3: for(k=0; k<n-2; k++)
       A[k] = A[k] - 5;
```

(b) After peeling second loop

Fig. 7. Loop peeling example

```
L4: for(i=0; i<n-2; i++) {
      A[i] = A[i] * 2;
      A[i] = A[i] + 3;
    }
    A[n-2] = A[n-2] + 3;
    A[n-1] = A[n-1] + 3;
L3: for(k=0; k<n-2; i++)
      A[k] = A[k] - 5;
```

(a) First fusion

```
L5: for(i=0; i<n-2; i++) {
      A[i] = A[i] * 2;
      A[i] = A[i] + 3;
      A[i] = A[i] - 5;
    }
    A[n-2] = A[n-2] + 3;
    A[n-1] = A[n-1] + 3;
```

(b) Last fusion

Fig. 8. Loop fusion and movement of intervening code example

```
L1: for(i=0; i<n-2; i++)
       A[i]=A[i]*2;
L2: for(j=0; j<n; j++)
       A[j]=A[j]+3;
L3: for(k=0; k<n-2; k++)
       A[k]=A[k]-5;
```

(a) Original loops

```
L4: for(i=0; i<n; i++)
      if (i < n-2)
        A[i]=A[i]*2;
        A[j]=A[j]+3;
      else
        A[j]=A[j]+3;
L3: for(k=0; k<n-2; k++)
      A[k]=A[k]-5;
```

(b) First fusion

```
L5: for(i=0; i<n; i++)
      if (i < n-2)
        if (i < n-2)
          A[i]=A[i]*2;
          A[i]=A[i]+3;
        else
          A[i]=A[i]+3;
        A[k]=A[k]-5;
      else
        if (i<n-2)
          A[i]=A[i]*2;
          A[j]=A[j]+3;
        else
          A[j]=A[j]+3;
```

(c) Second fusion

Fig. 9. Loop fusion using guards

between L4 and L3, as shown in Figure 8(a). This new intervening code has to be moved before the next fusion, as shown in Figure 8(b).

An alternative to iteration peeling is to introduce *guards* in the fused loop, as shown in Figure 9. The introduction of guards prevents the generation of additional intervening code. However, it creates fused loops with complex control flow. These complex control structures: (1) cause the dynamic execution of more branch operations, (2) may prevent future optimizations such as software pipelining, and (3) make instruction scheduling and register allocation more difficult. Thus once all fusions are performed, ISS separates loops fused with guards into individual simpler loops.

6 Runtime Bounds Check

When the relationship between the upper bounds of the two loops cannot be determined at compile time, a run-time bounds check must be performed. The

```
for(i=0; i<n; i++)
   A[i] = A[i] * 2;


for(j=0; j<m; j++)
   A[j] = A[j] * 3;
```

(a) Original loops

```
S = max(n,m);
T = min(n,m);
for(i=0; i<S; i++) {
  if (i<T) {
   A[i] = A[i] * 2;
   A[i] = A[i] * 3;
  }
  else {
   if(i<n)
     A[i] = A[i] * 2;
   else
     A[i] = A[i] * 3;
  }
}
```

(b) After Fusion

```
S=max(n,m);
T=min(n,m);
for (i=0; i < T; i++) {
   A[i] = A[i] * 2 ;
   A[i] = A[i] * 3 ;
}
for (i=max(T,0); i < n; i++)
   A[i] = A[i] * 2 ;
for (i=max(n,0); i < S; i++)
   A[i] = A[i] * 3 ;
```

(c) After ISS

Fig. 10. Run time bounds check example

fused loop combines the bodies of the two original loops for the minimum iteration count. Residuals of the two loops can then be executed depending on the iteration counts of the original loops.

For instance, assume that n and m in Figure 10(a) are not known at compile time. During loop fusion we want to generate the code shown in Figure 10(b). The upper bound of the fused loop is the maximum of the two original upper bounds. The execution of the composition of the bodies of the two loops is guarded by a test comparing with the minimum of the original bounds. Finally, the remainder iterations of the longer loop are executed. Applying ISS results in the code shown in Figure 10(c). The max(T,0) and max(n,0) in the resulting loops are necessary to preserve program semantics.

7 ISS as an Enabling Technique

The previous sections showed that ISS can be used to simplify code generated by optimizations such as loop fusion. ISS also enables optimizations that could not be performed in the presence of dynamic branches. For example, consider the loop in Figure 11(a).

This loop initializes the first 25 columns of each row in the two dimensional array A to zero and doubles all other entries in the array. However, A is traversed in column-major order while multidimensional arrays are stored in row-major order in the C programming language. Thus the data reference in this loop is extremely inefficient as it will result in a cache miss for every iteration of the inner loop (provided that the dimensions of A are larger than a cache line). Loop interchange, is an optimization that detects this type of memory access and interchanges the outer and inner loops to improve cache performance [3]. Un-

```
for (int j=0; j < 10000; j++) {
  if (j < 25) {
    for (int i=0; i < 10000; i++) {
      A[i][j] = 0;
    }
  }
  else {
    for (int i=0; i < 10000; i++) {
      A[i][j] += A[i][j];
    }
  }
}
```

(a) Original Loop

```
for (int j=0; j < 25; j++) {
  for (int i=0; i < 10000; i++) {
    A[i][j] = 0;
  }
}
for (int j=25; j < 10000; j++) {
  for (int i=0; i < 10000; i++) {
    A[i][j] += A[i][j];
  }
}
```

(b) After ISS-enabled interchange

Fig. 11. Loop interchange enabled by ISS

fortunately, these loops cannot be interchanged because of the dynamic branch guarding the innermost loop. After ISS has removed the dynamic branch, the code shown in Figure 11(b) is generated. Loop interchange will then be able to interchange the outer loop with the inner loop, resulting in a more efficient traversal of `A`.

Using a small test program containing the above code example, the runtime went from 12.88 seconds without Index-Set Splitting to 0.40 seconds using Index-Set Splitting.[3] This performance improvement is a result of the two loops being interchanged, resulting in increased cache performance. However, this transformation would not be possible if ISS did not eliminate the dynamic branch guarding the inner loops, thereby creating perfect loop nests. This demonstrates the ability of ISS to enable other optimizations, resulting in improved performance.

8 Experimental Evaluation

This section presents an experimental evaluation of a robust implementation of ISS in the development version of the IBM XL compiler suite. When introduced by itself in a compiler suite, ISS has the potential to degrade both compilation time and execution time. The appeal of ISS is its integration with other loop optimizations, as discussed in Section 7. Compile time degradation can be attributed to the processing of additional loops by later optimizations. Runtime degradation will occur if ISS creates many loops with small iteration counts or loops that are not executed at all. When control flow reaches a loop that is not executed, it still has to execute a test for the loop terminating condition. Also, if the compiler is not able to eliminate *min* and *max* computations introduced by ISS in hot paths, performance may also degrade. A careful implementation

[3] This test program was run on the same machine used to collect results in Section 8.

of ISS should have only minor impact on compilation and execution time, and thus enable subsequent optimizations to profit from a simpler loop structure in the code. The results of this experimental study can be summarized as follows:

- A total of 107 opportunities for ISS are found in several benchmark suites before loop fusion is applied. With the application of loop fusion, the number of ISS opportunities increased to 133.
- ISS does not increase compilation time. For the SPEC 2000 suite the compilation time is reduced by 17 seconds (0.3%). For a combination of benchmarks from Perfect, Quetzal and NAS, this reduction is of 34 seconds (1.6 %).
- Execution time variations due to ISS alone are very small for the SPEC 2000 benchmark suite (less than 3%). For benchmarks in the Perfect suite this variation can be larger (from 8% slower to 8% faster), but these benchmarks have very short runtimes (less than 5 seconds).

We prototyped ISS in the development version of the IBM XL compiler suite. Benchmarks were compiled using this development compiler and run on an IBM p630 machine, equipped with two POWER4$^{\text{TM}}$processors, 2048 MB of memory and running AIX$^{®}$5.1.

8.1 Opportunities for ISS

Table 1 shows the number of opportunities to apply ISS in standard benchmark suites. These opportunities were counted using compile-time instrumentation. The benchmark suites listed on Table 1 were tested in their entirety. The benchmarks not shown had no opportunities for ISS. An opportunity to apply ISS is a loop that contains a loop variant branch that splits the range of the loop index. The table shows that in some benchmarks there is a significant number of loops to which ISS applies even when loop fusion is not performed. This empirical result is evidence that ISS is a general technique that may benefit the implementation of optimizations in a compiler beyond the loop restructuring framework. The results also show that loop fusion creates additional ISS opportunities that can be detected and handled by our implementation.

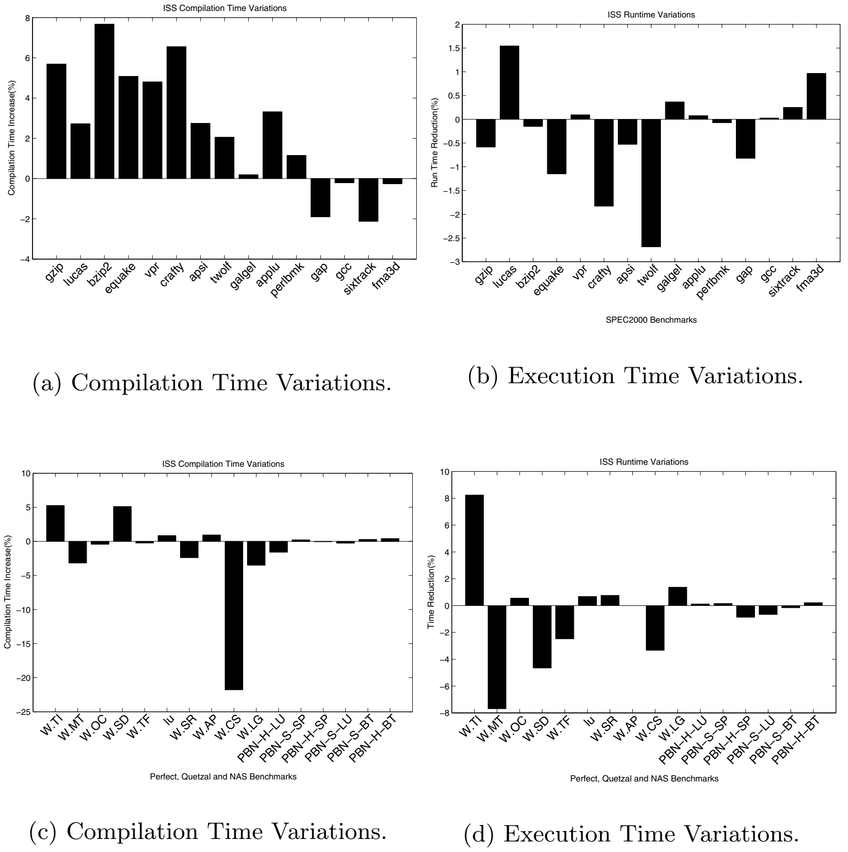
8.2 Variations in Compilation and Execution Time.

The variations in compilation time and execution time are presented in Figure 12. The bar graphs show the percentage *increase* in compilation time and the percentage *reduction* in execution time. Thus, a negative number in Figures 12(a) and 12(c) means the compilation process is taking less time when ISS is applied (*i.e.,* a larger-magnitude negative number is better). Similarly, a positive number in Figures 12(b) and 12(d) means the program execution time is lower when ISS is applied (*i.e.,* a higher positive number is better). The baseline for the comparison is an optimized compilation (at level *-O3 -qhot*) without ISS. In both the baseline and the ISS versions of the compiler all standard, and most advanced, optimizations found in a commercial compiler are performed. ISS has complex interactions with other optimizations.

Table 1. Number of times that an opportunity to apply ISS was identified

Suite	Bench-	No Loop Fusion	Loop Fusion	
	mark	ISS Opportunities	Loops Fused	ISS Opportunities
SPEC2000	bzip2	1	4	2
	crafty	2	7	4
	eon	1	0	1
	gap	7	4	9
	gzip	0	4	3
	perlbmk	2	0	2
	twolf	4	0	4
	vpr	1	0	1
	applu	2	4	3
	apsi	8	4	8
	equake	1	0	1
	fma3d	1	36	3
	galgel	6	21	6
	lucas	1	2	1
	sixtrack	16	26	24
Perfect	W.CS	2	1	2
	W.LG	0	2	1
	W.MT	1	1	2
	W.SR	2	4	3
	W.OC	2	0	2
	W.TF	1	4	2
	W.AP	8	4	8
	W.SD	3	4	3
	W.NA	2	4	4
	W.TI	3	4	4
Quetzal	lu	17	15	17
	rnflow	6	8	6
NAS PBN-S	BT	1	8	1
	LU	1	11	1
	SP	1	9	1
NAS PBN-H	BT	1	9	1
	FT	1	0	1
	LU	1	7	1
	SP	1	12	1
	Total	**107**		**133**

The normalization to the baseline times in the presentation of percentage variations may be misleading. Thus, for convenience, the benchmarks in Figure 12 are sorted from left to right based on their baseline compilation time. In Figure 12(a) benchmarks located to the left of `apsi` have a compilation time of less than one minute. `apsi` and `twolf` have a compilation time of less than two minutes. Similarly, in Figure 12(c) all benchmarks to the left of `W.TF` have a compilation time of less than one minute and all benchmarks to the left of



(a) Compilation Time Variations.

(b) Execution Time Variations.

(c) Compilation Time Variations.

(d) Execution Time Variations.

Fig. 12. Variation in the compilation time and run time using ISS on SPEC2000 (a and b) and on the Perfect, Quetzal and NAS (c and d) benchmark suite [5]

W.LG have a compilation time of less than two minutes. The compilation time of most benchmarks is not significantly impacted by ISS. applu's compile time increases from 207 seconds to 214 seconds. Furthermore, compilation is faster for the benchmarks with the longest compilation times: gap, gcc, sixtrack and fma3d. The total aggregated compilation time for the SPEC2000 suite does not change significantly: it is reduced by 17 seconds (or 0.3%) when ISS is applied. Thus the simplified loop structure provided to later optimizations compensates for the time spent on ISS. Similarly, the aggregated compilation time for benchmarks listed in Figure 12(c) is reduced by 34 seconds (1.6 %) with ISS.

The variations on execution time because of ISS are very small. As shown in Figure 12(b) execution time variations are under 3% (reductions of 3.3 seconds in

[5] Measurements did not use the official SPEC tools.

lucas and fma3d and additional 3.5 seconds in crafty and 12.5 seconds in twolf are the largest time variations). While the percentage variation in run times for the benchmarks in Figure 12(d) are larger, the W.* benchmarks from the Perfect suite have very short running time. The largest runtime variation in the W.* benchmarks is 0.11 seconds. The largest variation in runtime in Figure 12(d) is for the PBN-H-SP benchmark whose runtime increases by 1.7 seconds.

The small variations in execution time is evidence that the implementation of ISS in this industry-strong compiler is robust. Further improvements to loop optimizations, currently underway, that were enabled by ISS should produce overall performance improvements.

8.3 Micro-architecture Study

ISS does not have a significant impact on the runtime performance of the benchmarks tested. However, a large number of loops contained ISS opportunities. Thus, the question still arises as to the effects that ISS code changes have on the execution of the program. Since ISS removes loop variant branches from loops, one metric that should be affected by ISS is the number of branch mispredictions incurred during the execution of a program. By monitoring hardware performance counters, we examined the execution of several benchmarks to determine the number of target address branch mispredictions.

The study revealed that crafty has a 30% increase in the number of branch mispredictions (from 5.7 billion to 7.4 billion), while twolf's branch mispredictions increased from approximately 122 million without ISS to 1.1 billion with ISS. These additional mispredictions should contribute to the increased running time of these benchmarks. An analysis of the code generated for twolf and crafty reveals that the values of the *min* and *max* statements inserted by ISS could not be computed at compile time. The runtime execution of these *min* and *max* statements should be the cause of the performance degradation.

Significant reductions in branch mispredictions occur in apsi (82%, from 630 million to 111 million) and fma3d (31%, from 15 billion to 10 billion). However, these reductions did not translate into improved running times. A possible explanation is that the hardware was able to recover effectively from these branch mispredictions in the code generate by the baseline compiler.

9 Related Work

Loop unswitching is a similar technique to index-set splitting in the sense that a loop with a condition is converted into two non-control flow equivalent simpler loops [1]. However, as defined by Frances Allen and John Cocke, unswitching only does the conversion when the test's conditional is loop-independent [4]. In contrast index-set splitting performs multiple unswitches of tests on the value of the index variable of the loop. Another distinction between loop unswitching and ISS is that the separate loops created by unswitching are not control flow equivalent, while ISS creates control flow equivalent loops.

Loop fusion has been implemented in compilers for over twenty years [5]. Optimizations to loop fusion have been proposed by Gao [6], Ding [7, 8], McKinley [9, 10], Allen, and Kennedy [11] among others. Most research papers on loop transformations prescribe selective fusion of loops, *i.e.,* a decision about the profitability of fusing two or more loops is made during the loop fusion phase. Placing the decision about loop groupings in the fusion leads to several graph-based optimization algorithms. The IBM XL compilers take a different approach to loop restructuring: maximal loop fusion is applied first and then selective loop distribution, using several heuristics, takes place.

Allen, Callahan, and Kennedy described loop alignment as a solution to eliminate synchronization in the execution of parallel loops [5]. Alignment is used to describe the Global Alignment Network (GAN) by Padua *et al.* GAN distributes data in a multiprocessor system. For instance GAN could partition a vector and distribute its elements to several processors in the system to eliminate cross-iteration dependencies when creating fully parallel loops [12].

Yang *et al.* propose a technique to improve the order of branches based on run-time profile [13]. However, their technique does not reverse the order of loops and conditionals.

10 Conclusions

This paper introduced a new code transformation that enables the unswitching of loops that contain conditionals that are loop-dependent. Index-set splitting was implemented in the development version of the commercial IBM XL compilers and tested with four benchmark suites, including the industry standard SPEC2000 suite. The use of ISS as a convenient tool to implement a cleaner loop fusion transformation was also discussed.

ISS removes loop variant branches from inside a loop body, splitting the original loop into several loops with varying ranges. The compiler can then remove ranges that it can prove will never execute. ISS significantly impacts the generated code: the resulting loop bodies are smaller, making it easier to perform resource allocation and instruction scheduling (including modulo scheduling). ISS enables loop interchange, resulting in improved cache performance. ISS can also benefit other loop optimizations, such as loop parallelization, by removing loop-carried dependencies. On architectures where predicated instructions are available, the removal of the loop variant branch will remove the necessity of predicating the instructions that are control dependent on the branch. This will prevent aborted predicated instructions from polluting execution streams.

The static evaluation of ISS discovered opportunities for application of ISS even when loop fusion is not performed, thus indicating that ISS is a general technique that may benefit other compilers. The dynamic measurements of performance indicate that there is no significant variation in compile time and run time due to ISS alone. Thus downstream optimizations enabled by ISS shall produce overall performance improvements.

Acknowledgments

This research was supported by the IBM Center for Advanced Studies (CAS), and by a grant from the Collaborative Research Development (CRD) Grants program of the National Sciences and Engineering Council of Canada (NSERC) of Canada. Some of the infrastructure used for the experimental evaluation was acquired through a grant from the Canadian Foundation for Innovation (CFI).

Trademarks

References

1. Cooper, K.D., Torczon, L., *Engineering a Compiler.* Morgan Kaufmann (2004)
2. Blainey, B., Barton, C., Amaral, J.N., Removing impediments to loop fusion through code transformations. *Workshop on Languages and Compilers for Parallel Computing*, College Park, MD (2002)
3. Wolfe, M., *High Performance Compilers for Parallel Computing.* Addison Wesley, Longman (1994)
4. Allen, F.E., Cocke, J., A catalogue of optimizing transformations. In Rustin, R., ed., *Design and Optimization of Compilers.* Prentice-Hall (1972) 1–30
5. Allen, R., Callahan, D., Kennedy, K., Automatic decomposition of scientific programs for parallel execution. *Symposium on Principles of Programming Languages*, Munich, Germany (1987) 63–76
6. Gao, G.R., Olsen, R., Sarkar, V., Thekkath, R., Collective loop fusion for array contraction. *Workshop on Languages and Compilers for Parallel Computing*, New Haven, Conn., Berlin: Springer Verlag (1992) 281–295
7. Ding, C., Kennedy, K., The memory bandwidth bottleneck and its amelioration by a compiler. *International Parallel and Distributed Processing Symposium*, Cancun, Mexico (2000) 181–189
8. Ding, C., Kennedy, K., Improving effective bandwidth through compiler enhancement of global cache reuse. *International Parallel and Distribute Processing Symposium*, San Francisco, CA (2001)
9. Kennedy, K., McKinley, K.S., Maximizing loop parallelism and improving data locality via loop fusion and distribution. *Workshop on Languages and Compilers for Parallel Computing*, Portland, Ore., (1993) 301–320
10. Singhai, S., McKinley, K., A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, **40** (1997) 340–355
11. Allen, R., Kennedy, K., *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers (2002)
12. Padua, D.A., Kuck, D.J., Lawrie, D.H., High-speed multiprocessors and compilation techniques, *IEEE Transactions on Computers*, **29** (1980) 763–776
13. Yang, M., Uh, G.R., Whalley, D.B., Improving performance by branch reordering. *Programming Language Design and Implementation* (PLDI), Montreal, Canada, (1998) 130–141