An Implementation of Loop Fusion Using Simple-SUIF

John Bent
University of Wisconsin, 1210 West Dayton Street, Madison, WI 53703
johnbent@cs.wisc.edu

January 23, 2001

Abstract

Loop fusion is a compiler optimization that merges the bodies of multiple loops into a single loop. Doing so in a serial program can reduce the loop overhead of the program as well as improve data locality and increase opportunities for better cache utilization. Although not explored in this paper, it should be noted that loop fusion has been proven beneficial as well in parallel programs [11].

1 Introduction

This project has been an even split between an implementation project and a survey of some of the available literature about the topic of loop fusion. Although the original intent was an implementation only project, I have done reading as necessary to learn more about the topic. I was originally motivated by a frustration I experienced during the implementation of the third project. While implementing invariant code motion and dead code elimination, I would occasionally construct test programs which contained so much redundancy and loop invariance that the resulting program would execute an empty loop. It was my original desire to eliminate these empty loops from the program entirely.

However, due to the infrequency of empty loops within actual programs, I have broadened my focus and have added loop fusion to my original project three as well as the elimination of empty loops. My current compiler now correctly implements dead code elimination, useless assignment, loop invariant code motion, dead loop elimination and some opportunities for loop fusion.

Although using Simple SUIF [5] greatly simplifies the development of the compiler, we shall show that loop fusion remains a tricky problem. After discussing the implementation of dead loop elimination and of loop fusion, we will briefy consider some performance profiling using James Larus' QPT tool [8]. Although we will see the desired performance benefit to be gained by fusing loops, we will also consider why loop fusion should occasionally be avoided.

2 Dead Loop Elimination

Although found only infrequently[1] in real programs, empty loops can occasionally be created by compiler optimizations such as useless code removal and invariant loop code motion. In these cases, the condition of the loop becomes essentially useless. A data flow analysis of variable faintness [3] might be able to eliminate these useless instructions in the case in which the index is not live after the loop. However, if the index variable is live after the loop, then loop elimination is impossible using faint variable analysis. To eliminate these useless loop condition instructions requires both the ability to identify empty loops and knowledge of the eventual final value of any live index variables.

2.1 Identifying Empty Loops

In order to decide whether a loop[2] is empty, my algorithm first identifies all instructions which belong to the conditional portion of the loop. For a loop to be considered fusable, it must have an initial value which is incremented per iteration and compared against a flag value. To find loop condition instructions, I recognize that the compare instruction must be found at the back edge of the only node in the loop body that has a successor out of the loop. From this compare instruction, I find both the flag and the index variables for the loop.

To distinguish between the flag and the index variable I note that only the flag should be defined within the loop. In order to be a candidate for loop fusion, the index variable of the loop condition should not be defined within the non-conditional body of the loop. Flowing backwards

[1]Apologies to those programmers who use empty loops as an imprecise sleep.

[2]Only natural [2] loops are considered.

from the compare instruction, an increment instruction should be found. Due to algorithmic knowledge of Simple SUIF, we know that the increment value, if constant, should be loaded into a temporary register immediately preceding the increment instruction.

Finding initial values for both the flag and the index variables use similar backward flow analyses. If the flag is a temporary variable, its loaded constant value should immediately precede the compare instruction. To find the constant value upon loop entry for pseudo variables is done using our previously computing reaching information. For a loop to be considered fusable in my algorithm requires that there is only one possible non-loop based reaching definition for pseudo variables used in the loop condition. By reading instructions backwards from the node in which the pseudo variable is defined, the algorithm can find the constant value which reaches the loop entry point. Note that if the variable is given a non-constant definition that my algorithm will not consider that node fusable. Constant propogation would have increased the opportunities for loop fusion but it is not implemented in my compiler. However, by implementing loop fusion as I have done by searching backwards for constant values, I have discovered another benefit of constant propogation as well as developed an appreciation for how it is implemented.

Once instructions related to the loop condition are identified then loop emptiness can be evaluated, assuming that the loop is still considered fusable. Any loop containing only instructions identified as condition instructions is considered empty and a candidate for dead loop elimination.

2.2 Preserving Live Indices

However, it is not safe to eliminate a loop merely because it is known to be empty as defined above. If the index variable of the loop is truly live following the loop execution than it must contain the value it would contain were the loop not removed. Because my code does not analyze faintness, I initially consider that all loop indices are truly live at the loop exit point.

This is why finding the constant values for the loop condition variables is so important. By identifying a constant value for the flag, the initial value of the index and the increment amount and the increment and compare operators, I am able to compute the final value of the loop index at compile time. By inserting into the loop prehead a new instruction which loads this constant value in a temp and then another instruction to copy the temp into the index variable, an empty loop can then be safely eliminated. Note that any faintness the index variable had due to being incremented in the loop has been removed. If the index variable is not truly live, it can now be removed by useless instruction analysis.

An example of empty loop elimination with the preservation of its live index is shown in figure 1. Note that the index is assigned its final value and placed in the unique preheader of the loop which is then linked to the unique successor of the loop and the loop itself is left dangling and can be removed. In cases in which the loop does not have a unique preheader or a unique successor, I create them in order to simplify the loop removal algorithm.

3 Loop Safety

An interesting side benefit of evaluating the loop condition parameters at compilation time is that “loop safety” can be evaluated for some loops. By loop safety, I mean specifically whether a loop will terminate. If the initial index variable, the increment value and the flag are all known constant terms, then an evaluation can be done at compilation time to determine whether the loop will ever terminate and to discover the eventual value of the index variable if the loop is shown to terminate. For example, if the initial value of the index is set to 0, the increment value is 1, the flag is 10 and the increment operation is an add, then the loop is expected to either complete after 10 iterations or to enter an infinite loop[3] depending on the comparison operator. In the case of an “unsafe” loop of this nature, my compiler will warn the user that they may have inadvertently implemented an infinite loop. Similarly, the compiler warns the user if it is determined that a loop will iterate zero times.[4]

4 Loop Fusion

Loop fusion is a compiler optimization that merges the bodies of multiple loops into one single loop. The loop overhead is thereby reduced by a factor of two for each pair of fused loops. In order to merge loops bodys, it must be both determined that the loops share some common set of iterations and it must be shown to be demonstrably safe to merge the loop bodies.

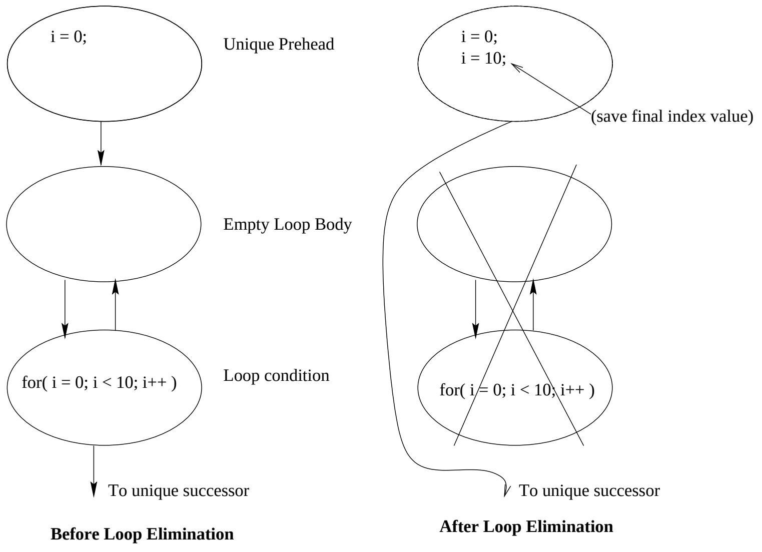

Figure 1: Dead loop elimination

```
for( i = 0; i < 10; i += 3 ) {
    tmp = i * 10;
    a = a + tmp;
}
for( j = 0; j < 10; j += 3 ) {
    b = b * j;
}
```

Figure 2: Iteration sharing

4.1 Iteration sharing

The initial consideration when evaluating the fusability of two loops is whether they share some set of iterations. By using the same algorithms I used in the evaluation of empty loops I can find the loop condition instructions and identify the initial index value of the loop as well as the end flag and increment values and the compare and increment operators. For example in figure 2, it is seen that the two loops can be merged as their condition instructions are identical.

One caveat is to notice that the body of the second loop however reads the index value of the loop (i.e. variable *j*). Using the same algorithm as described in empty loop removal, we will add an instruction to the preheader of the second loop setting the value of *j* to its evaluation value which can be determined at compilation time to be 12. However, the loop body instruction, *b* = *b* * *j* will be moved above this into the body of the first loop. As the value of *j* will be the same as *i* for each iteration of the loop, we can safely replaces occurences of *j* with *i* for all moved instructions. Once the loops have been merged, the now empty loop becomes a candidate for empty loop removal.

[3]Due to finite sized data-types, most "infinite" loops will eventually terminate. A loop however which uses an add increment operation and an increment value of 0 will be truly never exit.

[4]After implementing this behavior, I discovered that SUIF actually does this itself . . .

4.1.1 Partially fusable loops

```
for( i = 0; i < 100; i ++ ) {
    a = a + i;\\
}
for( j = 0; j < 99; j ++ ) {
    b = b * j;
}
```

Figure 3: Partially fusable loops

Although it is possible to partially merge loops which share a large, but not complete, subset of the total iterations, my code does not attempt to do so. In this case the reduction in loop overhead will not be a factor of two but will be weighted by the overlap in the loop iterations.

Figure 3 shows an example of partially fusable loops in which the loop overhead reduction would be ninety-nine percent. To partially fuse these two loops would require a contraction of the first loop's iterations to ninety-nine such that the body of the second loop could then be inserted. The second loop could then be removed, but we must add

```
for( i = 1; i < 100; i ++ ) {
    A[i] = i;
}
for( j = 1; j < 99; j ++ ) {
    B[j] = A[j] + j;
}
```

Figure 4: Fusability with array references

also a new loop with iterations equal to one (one hundred minus ninety-nine) to complete the necessary iterations of the first loop. A copy of the first loop's body would need to be inserted into this new loop at the cost of increased code size. [5]

4.2 Safety and legality of fusion

Once it is known that two loops share iterations and condition values, it must be shown that their bodies can be safely merged. To do so informally, is relatively straightforward: Two loops can be fused if their nested depths are the same, and if doing so does not break any dependencies between them.

A formal examination of loop fusability is much more complicated however. Any dependencies between loop bodies are loop independent; when merged however they lose this independence because fusion places both the source and sink together in the same iteration body. Array references in particular make this dependence analysis more difficult. In figure 4, we see that it is legal to fuse the two loops as the dependence between the arrays is forward and occurs at the same iteration depth. Substantial gains might be seen in this case because of the locality of array reference. *A[i]* would need to be loaded only once instead of twice for each iteration of the loops.

If however the instruction in the second loop is changed to *B[j] = A[j-1]*, the loops are no longer safely mergable as the second loop would attempt to reference an array index in *A* which has not yet itself been assigned the correct value that it should have at the point in which the non-fused instruction would normally be executed. Some analyses [9] [1] have been developed to allow shifting the iterations by the necessary number in order to allow legal fusion. In the above example, it is clear that the loop would need to be shifted once, but a formal examination of this problem is beyond the scope of this paper.[6]

[5] An alternative implementation would be to merge the loops as if they were totally mergable, set the condition to iterate to the maximum value of both loops and then protect the smaller loop within another conditional statement. This reduces the redundant code of the first approach by introducing more runtime instructions.

[6] And probably beyond my programming skills as well.

4.2.1 My implementation of loop fusion

For my compiler, I consider loops fusable if one follows the other in a straight sequential block of code. More formally, if the first dominates the second and the second postdominates the first and there is only possible path between the two. Further, I require that there is no write-read dependencies from the first to the second loop. If these requirements hold true and the loops have been demonstrated to show conditional equivalence, then my compiler will consider the two loop bodies fusable.

For the simplicity of the algorithm I further require that the tail of each loop (its condition node) has only one predecessor within the loop body and that the head of each node has only one unique preheader (this is also required for loop invariant code motion as well). If this is not the case, I create these nodes to simplify the loop fusion. The first loop is "cut" between its tail node and the loop body predecessor of the tail node. The second loop is removed, excluding its condition node, and is inserted into the "cut" in the first loop.

The control flow graph is then patched to connect the preheader of the second loop to the successor of its tail node. Note that this completely removes the second loop's condition node from the program. Using the technique described in empty loop elimination, the index variable of the second loop is set to its final value and placed in the preheader of the second loop so that it is available in its correct location in reference to the control flow graph. Finally, the body of the second loop, which is now included in the first loop, is searched to replace any instances of the second loop's condition variables with the equivalent variables of the first loop.

4.2.2 Using dot

An example of this can be seen in figure5; these graphs were actually generated by my compiler which can be made to write a dot [7] input file both before and after optimizing. These dot files represet the control flow graph of the program's procedures and provided an invaluable debugging aid during the development of the compiler. Note how the algorithm has created unique condition predecessors for each of the loops.

4.3 When to fuse

Loop fusion can result in significant performance boosts by reducing loop overhead, by allowing more instruction overlap, by reducing cache misses and by allowing a better awareness of associativity between instructions within the merged loop bodies. Many studies [10] [4] [11] [9] have shown the gains to be had by loop fusion.

Using QPT, I have also seen these performance gains using my program. Please refer to table 4.3 for this profiling information. In addition, the source code for the test programs and the test scripts to run them the test programs as a batch can be found in their respective directories in the handin directory.

4.3.1 When not to fuse

Loop fusion is not always a good idea. Very large granularities can induce register spilling that would not occur in independent loops as well as increasing the instruction size beyond the size of the instruction cache. In fact, some authors have proposed the opposite approach, loop distribution. [6] Splitting large loops may resolve some of the problems that can be associated with loop fusion as well as introduce more opportunities for parallelizing code in a distributed environment.

5 Conclusion

We have explained what loop fusion is in a general sense. From there, we described our specific implementation. By attempting to develop loop fusion algorithms, we have discovered that there are many different classes of loop fusion with varying degrees of complexity. Performance measurements indicate that loop fusion is often a significant optimization. However, loops of large granularities and parallel programs may actually suffer under a strict policy of loop fusion and should therefore be evaluated differently. In conclusion, loop fusion is recommended for small loops in serial programs. The size of the instruction cache and the number of available registers need to be considered when deciding whether to merge or distribute loop bodies.

References

[1] Alain Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[2] Charles Fischer. Finding loops in control flow graphs.

[3] Charles Fischer. Cs701 final exam, December 1996.

[4] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. Technical report, School of Computer Science, McGill University, March 1992.

[5] Stanford Compiler Group. The simple-suif compiler guide.

[6] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, New York, NY, 1990.

[7] Elefherios Koutsofios and Stephen C. North. Drawing graphs with dot.

[8] James Larus. A quick program profiling and tracing system.

[9] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. 1995.

[10] Phillip Mucci and Kevin London. Performance optimization for the origin 2000, September 1998.

[11] S. Singhai and K. McKinley. Loop fusion for data locality and parallelism. In *Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems*, New Paltz, April 1996.

Test Program	With fusing	Without fusing	With no optimizations
arrays.c	2140	2940	2940
bigLoops.c	20000177	38000171	38000171
do_while.c	252	252	252
empty_loop.c	8154	8154	12152
empty_loops.c	16010175	16010175	24014173
fusable.c	276	306	316
infinite_loop.c	–	–	–
loops.c	12571	12571	12571
non_const_loop.c	1576332214	1576332214	1576332214
replaceIndexVars.c	324	404	404
simple_loops.c	307	307	375
smallLoops.c	365	521	521
while.c	257	257	257
zero_itr.c	148	148	150

Table 1: Reducing dynamic instruction counts using loop fusion

```
arrays.c    - this tests fusion for array references
bigLoops.c  - this tests fusion for big loops
do_while.c  - although the do_while is very similar to the for loop, and I
              originally meant to restructure its internal contents in order
              to make it fit the for loop model so that I could then
              consider it for fusion, I did not get to do this
empty_loop.c    - tests dead loop elimination
fusable.c       - a simple fusability test
infinite_loop.C - tests whether compiler correctly identifies and warns about
                  infinite loops.  Is a .C and not a .c so that the Makefile
                  will ignore it.  It is an infinite loop and would mess up
                  my group testing of these files.
loops.c         - nested loops - should not be fused
non_const_loop.c    - the flag is not a constant, this cannot be fused
replaceIndexVars.c  - tests whether index vars are correctly replaced when
                      appropriate when loops are fused
simple_loops.c  - non-fusable loops
while.c     - a while loop.  Suif changes this into a do_while and I
              originally wanted to be able fuse these as well as fuse for loops
              but I didn't get around to implementing this.  Although I'm
              confident that I now know how to do it.
zero_itr.c  - intended to test whether the compiler warns about loops that
              will execute zero times, but it turns out that SUIF beats me to
              to the punch and removes the loop itself
```

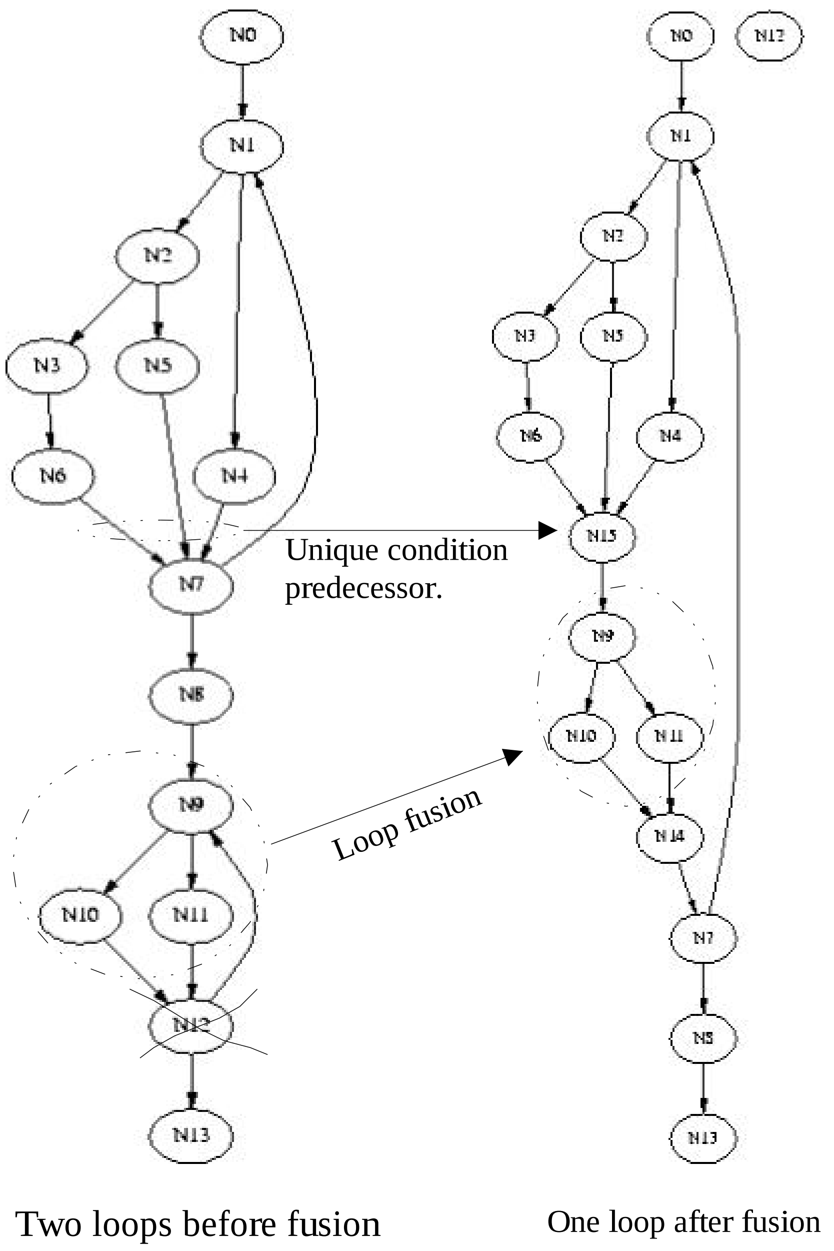

Figure 5: Loop fusion

These two CFG's are created by my compiler. Notice how the compiler will create unique condition predecessors for each loop (nodes 14 and 15 in the right side CFG). Creating this unique condition predecessors simplifies the loop movement algorithm. The second loop's condition node has been entirely removed from the graph and the body of the second loop (nodes 9, 10, and 11), which can itself by an arbitrarily complicated CFG, has been inserted at the tail of the first loop.