Enabling Loop Fusion in LLVM by Moving Intervening Code

Julia Aoun, Qingyi Chen, Mohamad Eter, Yile Gu, Musa Haydar

I. Introduction

Loop fusion is a loop optimization technique that attempts to combine multiple loops into a single one. Loop fusion may create slight improvements in overall program performance for some programs, particularly due to the removal of branching instructions. Previous work argues that loop fusion can create opportunities for other loop optimizations, such as loop distribution or software pipelining, to make better decisions when optimizing a program [2]. Furthermore, loop fusion may directly improve data locality if accesses to the same cache line are brought closer together [4].

Currently, a loop fusion pass is implemented for LLVM based on the algorithm presented in [1]. In order for loop fusion performed, the following criteria must be met for the two loops [1]:

1) The loops must be *adjacent* (i.e. there are no instructions between the two loops).
2) The loops must be *conforming* (i.e. they must execute the same number of iterations).
3) The loops must be *control flow equivalent* (i.e. both loops must be guaranteed to execute together).
4) The loops must have *no negative distance dependencies* between them.

Previous work discusses code transformations that seek to alleviate these requirements for loop fusion [1], [2]. However, the current pass does not perform these code transformations to enable loop fusion when these conditions are not met. One particular transformation is *Moving Intervening Code* (MIC), where instructions and basic blocks that occur between two *fusion candidate* loops are moved to adjacency between the two loops. Intervening code may be moved above the first fusion candidate or below the second, contingent on the dependencies between the intervening code and the two loops, respectively. The intervening code itself must also be considered safe to move to ensure the correctness of the resulting program. Once the intervening code has been moved, the loop fusion pass can run as before on the now-adjacent loops.

Our main contribution is an implementation of a Moving Intervening Code algorithm in the LLVM loop fusion pass[2]. In §II, we discuss the existing LLVM loop fusion pass and the necessary conditions to run it in more detail. Then, in §III, we discuss the design and implementation of our Moving Intervening Code algorithm. Finally, in §IV, we discuss our evaluation of this algorithm.

[1]The LLVM implementation of the loop fusion pass is available at https://llvm.org/doxygen/LoopFuse_8cpp_source.html or on the LLVM GitHub repository. The criteria for loop fusion are listed as described in the source code.

[2]Our implementation of the extended loop fusion pass is available on Github: https://github.com/musahaydar/eecs583-loop-fusion

II. Loop Fusion in LLVM

A. Implementation

In this section, we describe the current implementation of loop fusion as an LLVM pass. The implementation is based on the algorithm presented in [1]. It begins by considering the set of loops for a function. Control equivalent loops are considered *fusion candidates* and put into control-flow-equivalent sets sorted by dominance. Only loops with a single exit are considered for fusion. The loops in this set are then considered as pairs and the conditions for loop fusion are checked.

First, the loop fusion pass checks if the two fusion candidates are conforming (i.e. that they have the same trip count). If the trip count for either loop cannot be determined, the pair is skipped. The loop fusion pass will also attempt to peel the first loop (i.e. unroll some iterations) to make the loops conforming; the maximum allowed peel count is specified as a flag to the pass. Next, the pass checks that the fusion candidates are adjacent. Specifically, it ensures that the exit block of the first fusion candidate is the preheader of the second. Finally, if any negative distance dependencies are found between the loops, the pair is skipped.

Once it is determined that the fusion candidates meet all these criteria, the loop fusion can begin. First, any instructions in the preheader of the second loop must be hoisted into the first loop's preheader or sunk below the loop. If neither of these transformations is legal, the pair is skipped. The second loop's empty preheader will be removed. Then, the latch (the block with a branch to the header) of the first loop is modified to jump to the header of the second loop, and the latch of the second loop is modified to jump to the header of the first. Finally, all blocks are moved from the second to the first loop.

B. Running the Loop Fusion Pass

In order to run the loop fusion pass on LLVM bitcode, a few conditions must be met. First, the loops must be in rotated form so that the exit block of the first fusion candidate is equal to its latch. This can be accomplished by running the `loop-rotate` before loop fusion. If the loops are not in rotated form, the loop fusion pass cannot proceed.

The loop fusion pass must also be able to compute the trip counts of all the fusion candidates. By default, when

```
move_intervening_code:
   initialize vectors move_up and move_down
   for each intervening BasicBlock B:
      movable = false
      if !dependant(B, FC0):
         add to move_up
         movable = true
      if !dependant(B, FC1):
         add to move_down
         movable = true
      if !movable:
         return false //loops not fusible
   if move_up == set of intervening BBs:
      move intervening code above LC0
      return true
   else if move_down == set of intervening BBs:
      move intervening code below LC1
      return true
   else:
      return false //loops not fusible
```

Listing 1. High-level psuedocode for our implementation of the Moving Intervening Code algorithm.

compiling a benchmark in Clang with all LLVM optimizations disabled, the loop increment variable will be stored in memory. However, it must be in a register in order to be computable. The variable can be moved to a register in the bitcode by running the mem2reg LLVM pass. The mem2reg pass may introduce unnecessary phi nodes into the bitcode (i.e. phi nodes with only one value), which may prevent the pass from proceeding in the case where this node is detected as intervening code by the loop fusion pass. These phi nodes can be eliminated by running the instcombine pass.

It is also possible that, once these instructions are removed, empty basic blocks (i.e. basic blocks containing only a single branch to a single successor block) remain in the program, which similarly inhibits the loop fusion pass since they are detected as intervening code. To enable loop fusion in this case, we run the simplifycfg pass, which eliminates the empty basic blocks.

III. Moving Intervening Code

In this section, we discuss our main contribution: an implementation of the Moving Intervening Code algorithm within the LLVM loop fusion pass. An overview of the algorithm is shown in Listing 1, and is explained here in more detail. The function move_intervening_code is called after the pass detects that the loops are not adjacent. It attempts to move the intervening code at the basic block level. If the intervening code is successfully moved, the function returns true, and the pass proceeds to fuse the loops.

We initialize two vectors, move_up and move_down, that will contain the blocks which we've determined can be moved above the first fusion candidate FC0 or below the second fusion candidate FC1, respectively. We say a basic block is dependent on a loop if any of its instructions depend on any of the instructions in the loop candidates. For each basic block, we iterate through the instructions and check if any of them depend on the instructions in FC0 and FC1.

We consider three types of data dependencies between the instructions in the fusion candidates and instructions in the intervening code: read-write, write-write and write-read dependencies. If no such dependencies exist between the intervening block and FC0, we add it to the move_up vector. Similarly, if no dependencies exist between the intervening block and FC1, we add it to the move_down vector. If any basic block cannot be moved, then we've determined that cannot move all of the intervening code blocks, so the loops cannot be fused.

Once we've determined if all of the basic blocks can be moved, we compare the move_up and move_down vectors with the set of intervening basic blocks. If all of the intervening basic blocks can be moved either above the first fusion candidate or below the second, then we are free to move them. If not, we say that the intervening code cannot be moved.

As our approach moves all the intervening code together, we take the following approach. We assume that the exit block of FC0 contains nothing but intervening code. To move the code up:

1) The successor of the preheader of FC0 is set to be the exit block of FC0 (the first block of intervening code)
2) The preheader of FC1 (the last block of intervening code) has its successor updated to be the entry block of FC1
3) The successor of the exit block of FC0 is set to be entry block of FC1

For each of these steps, the phi nodes must also be updated appropriately. Once these transformations have been completed, the intervening code will have been entirely moved above FC0, take the place of it's preheader. At this point, the two loop bodies are adjacent, so that loop fusion can proceed. Moving code below the second loop is a similar transformation.

A. Limitations

Some programs may introduce intervening code such that some portion of it may only be moved above the first loop while another portion may only be moved below the second loop, due to dependencies between the intervening code and both loops respectively [2]. While our implementation moves all the intervening code together, a potential improvement to our algorithm would support this case. To do so, it must consider the dependencies not only between the fusion candidates and the intervening code, but also among the blocks of intervening code themselves. Control dependencies present a particular challenge here. Consider the intervening code in Listing 2. Here, the statement on line 14 can only be moved if the if-statement which guards it is moved along with it.

Another potential improvement would be the inclusion of a profitability analysis. It has been shown that loop fusion does not improve program performance in all cases, and in some cases may even hinder other optimizations. This may be due to increased register pressure, cache misses, or the creation of additional control flow [2], [5]. Currently, LLVM's loop fusion pass has a function stub for determining if fusion

```
1  int main() {
2     int x = 1;
3     int y = 1;
4     int z = 0;
5
6     scanf("%d", &z);
7
8     for (int i = 0; i < 5; ++i) {
9        x = x + i;
10    }
11
12    y = z;
13    if (z) {
14       z = y * 5;
15    }
16
17    for (int j = 0; j < 5; ++j) {
18       y = y * (1 + j);
19    }
20
21    printf("x: %d y: %d\n", x, y);
22    printf("z: %d \n", z);
23 }
```

Listing 2. The example with which we tested the improved loop fusion pass. The intervening code on lines 13-16 should be moved above the first loop, and then the loop should be fused.

```
12 if (z) {
13    z = x * 5;
14 }
```

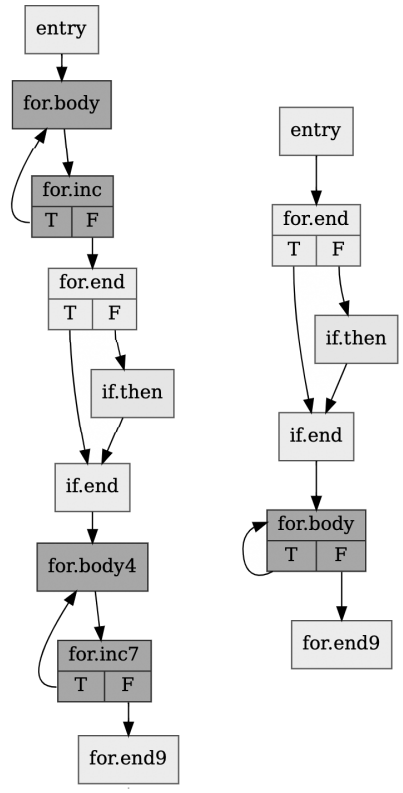

Fig. 1. The control-flow graph for the example program in Listing 2 before moving intervening code and loop fusion (left) and after (right).

Benchmark	Time	# Instr. Executed
MIC Up (Not Fused)	4.28 sec.	25 billion
MIC Up (Fused)	2.59 sec.	22 billion
MIC Down (Not Fused)	4.35 sec.	25 billion
MIC Down (Fused)	2.62 sec.	22 billion

Fig. 2. The results of running our contrived example for moving intervening code up (Listing 2) or down (Listing 3), with the loop iterating one billion times.

between two candidates is profitable, which always returns true, such that the loop fusion pass tries to fuse as many loops as possible. However, by moving intervening code, we may block the fusion of other, more profitable fusion candidates, and the potential for performance decreases still exists.

Finally, not all intervening code may be moved. In particular, instructions that may have side effects (e.g. function calls, volatile memory accesses, I/O operations) are considered unsafe to move in prior work [2]. A potential improvement is to broaden the criteria for movable intervening code by detecting which of these operations are actually safe despite their side effects.

IV. Evaluation

We evaluated our improved loop fusion pass with the example program presented in Listing 2. Here, there are two loops that would be fusible since they are conforming, control-flow equivalent, and do not contain any negative distance dependencies. However, there is intervening code on lines 13-16, such that the loops are not adjacent. The intervening code depends on an input unknown until runtime (the `scanf` call on line 7), making this example less trivial—using only constant values can result in the intervening code being sunk after constant value propagation when running the benchmark with LLVM optimizations enabled.

In Listing 2, the intervening code modifies the value of `y` such that, if the code is moved below the second loop, the print statement on line 21 would output a different value than expected. However, since the intervening code does not depend on the variable `x` which is produced by the first loop, the code can be raised above the first loop safely, and the loops may be fused. Listing 3 presents a slight modification to the intervening code such that `y` is no longer overwritten by the intervening code, and it instead depends on the value of `x`. In this case, moving it above the first loop is unsafe since it depends on the `x` produced by it. However, it can be safely moved below the second loop.

As the intervening code in this example makes the fusion candidates non-adjacent, LLVM's current loop fusion algorithm is unable to fuse the loops. However, we are able to move the intervening code in Listing 2 above the first fusion candidate and then allow the loop fusion pass to continue and fuse the loops. The result of the MIC transformation followed by loop fusion for Listing 2 is illustrated by the control-flow graphs in Figure 1.

To evaluate our MIC algorithm, we ran the contrived

example in Listing 2 which requires moving the intervening code up, as well as the example in Listing 3, which requires moving it down. We changed the loop iteration count from five to one billion to make the impact more clear. We compare the total execution time and the number of dynamic instructions in each test case. We use Linux time command to collect execution time information and a Pin tool [3] to count the number of dynamic instructions. The results are listed in the table in Figure 2. We find that, on these examples which run two loops of a billion iterations each, the binary which has the loops fused achieves around a 40% decrease in run time and around a 12% decrease in dynamic instructions executed.

It seems that, from the control flow graphs presented in Figure 1, the improvement in performance is due to the reduction in the number of branches that need to be executed. Specifically, the bodies of the two loops are combined such that they are both guarded by the same loop branches.

V. Conclusion

The current LLVM loop fusion pass requires the loops to be adjacent, making intervening code an obstacle to loop fusion. We were able to move the intervening code to enable loop fusion by evaluating if we can move the code above the first fusion candidate or below the second fusion candidate. We demonstrated that enabling loop fusion by moving intervening code can have benefits in terms of run time and dynamic instructions executed. The main limitation to removing intervening code is at compile-time, as this MIC algorithm is dependent on the number of instructions present in the loop body and intervening basic blocks. In the future, additional code transformations can similarly extend the existing loop fusion pass to further enable loop fusion.

References

[1] C. Barton, "Code transformations to augment the scope of loop fusion in a production compiler," Master's Thesis, University of Alberta, January 2003.

[2] B. Blainey, C. Barton, and J. N. Amaral, "Removing impediments to loop fusion through code transformations," in *Languages and Compilers for Parallel Computing*, B. Pugh and C.-W. Tseng, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 309–328.

[3] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[4] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, p. 424–453, jul 1996. [Online]. Available: https://doi.org/10.1145/233561.233564

[5] A. Qasem and K. Kennedy, "A cache-conscious profitability model for empirical tuning of loop fusion," in *Languages and Compilers for Parallel Computing*, E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 106–120.