Loosening the Control Flow Equivalence Restriction to Loop Fusion Through Code Transformation: Enabling Loop Fusion in the "For-If-For" Case

Mabel Chan and **Kevin Wang** and **James Wu**
University of Michigan

Abstract

This paper describes an approach to extend the "Removing Impediments to Loop Fusion Through Code Transformations" (Blainey et al., 2005) by loosening an additional constraint to loop fusion: the constraint that loops must be control flow equivalent. This paper describes previous related work that we are extending, our work procedure, and our proposed algorithm for loosening the aforementioned constraint in a specific case.

1 Introduction

Loop fusion is a transformation that fuses the bodies of loops together, combining two or more loops into a single large loop. Loop fusion has the potential to decrease the number of loop branches executed and create opportunities for data reuse. Furthermore, loop fusion increases the scope for optimization for further loop optimizations, such as loop distribution, which decides what parts of a loop redistributed out of the loop to create perfectly nested loops (Blainey et al., 2005). Due to these potential benefits, efforts have been made to loosen the conditions necessary for loop fusion to take place (Blainey et al., 2005). This project seeks to extend these efforts to loosen an additional requirement for loop fusion.

2 Problem Statement

The loop fusion algorithm is a transformation that combines the bodies of multiple loops together. Figure 1 illustrates a high level example of two loops being fused. The current loop fusion pass built into LLVM has four strict conditions that must be met before loop fusion can occur:

1. Loops must be control equivalent
2. There cannot be a negative distance dependency between the loops
3. Loops must have conforming bounds
4. There cannot be intervening code between the loops

Note that in Figure 1, the loops on the left of the arrow meet all four criteria for loop fusion: the two loops are control flow equivalent, meaning that when one loop executes, the second is guaranteed to execute; there are no dependencies between the loops; the loops have conforming bounds, meaning that they execute the same number of iterations; and there is no code between the two loops.

This project aims to loosen the condition that the loops to be fused must be control flow equivalent. To be specific, our paper will allow loop fusion in the case that conditions 2, 3, and 4 are met, and the code is structured in a way such that the first for loop is followed by an if statement containing the second for loop. Hereafter, this structure will be referred to as a "for-if-for" structure. Note that under traditional loop fusion conditions, the two loops in the for-if-for structure would not be considered control flow equivalent, since the second loop will not always execute after the first, so the loops will never be fused.

3 Related Work

This project aims to extend the work described in "Removing Impediments to Loop Fusion Through Code Transformations" (Blainey et al., 2005). The innovation of this paper is its proposed loop fusion algorithm that loosens conditions 3 and 4 of traditional loop fusion. Through this algorithm, two loops with intervening code in between can be made adjacent by moving the intervening code elsewhere, on the condition that the intervening code has no dependencies with either loop.

Moreover, two adjacent loops with non-conforming bounds can be fused through this algorithm in the condition that the difference in the number of iterations between the two loops can be calculated at compile time. Upon fusion, the new for loop will iterate a number of times equal to the larger iteration count of the two loops. Within the new loop, there is an if block that runs the bodies of both loops on the condition that the current iteration is less than the smaller iteration count of the two loops. If the current iteration exceeds that, then it will only run the body of the larger loop.

The paper concludes stating that more work could be done to extend their findings by loosening condition 1, the condition that both loops must be control equivalent. Our project aims to loosen this condition

Figure 1: Traditional Loop Fusion

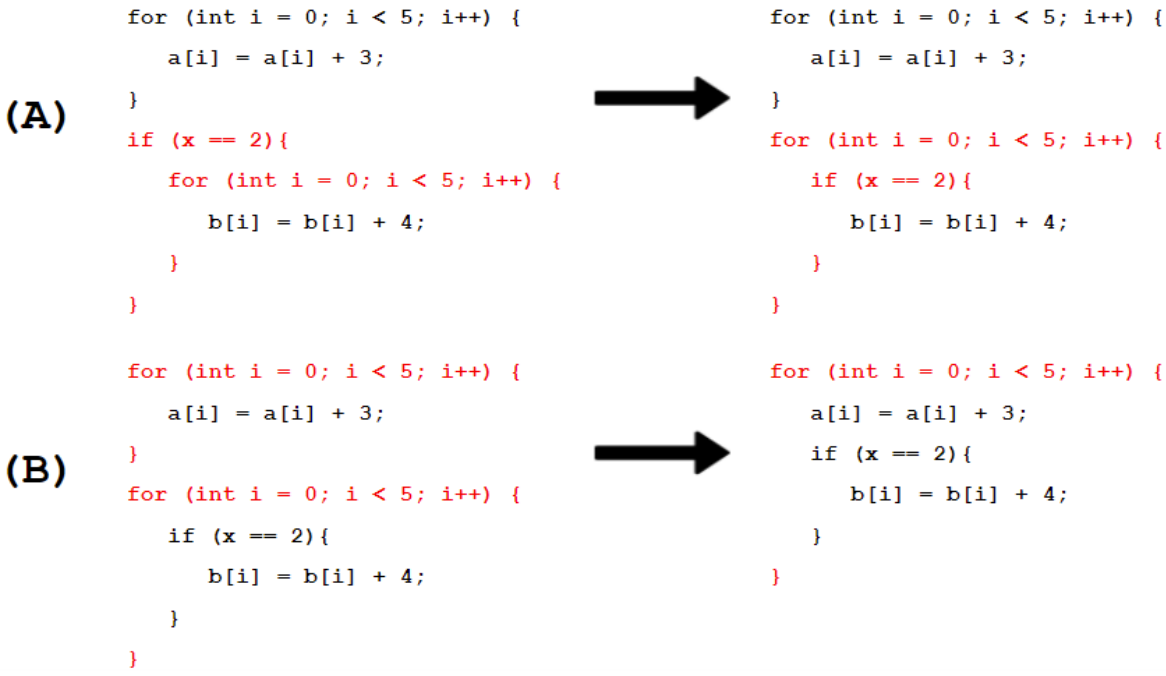

Figure 2: Proposed Loop Fusion Extension

by allowing the fusion of non-conforming loops in the for-if-for structure.

4 Procedure

To implement this loosened constraint for loop fusion, this project is comprised of an LLVM pass to identify all instances of the for-if-for structure in the code and convert them into the "for-for-if" structure, whereby the for loop nested within the if statement is flipped. After this "pre-pass" is executed, the built-in loop fusion algorithm from LLVM to conduct the fusion itself is executed, since loops in the for-for-if structure can be fused. Note that the pre-pass, which is the main innovation of this project, does not perform loop fusion, but rather, it provides further fusion opportunities for the LLVM loop fusion algorithm. The code for our pre-pass and its documentation are provided in the project GitHub repository[1].

Figure 2 illustrates an example of loop fusion under the loosened constraints. Part (A) demonstrates the if statement and the for loop changing places. As a result, the two for loops are now adjacent and control flow equivalent. Note that the behavior of the code remains the same; though the if condition is checked multiple times instead of just once. Part (B) illustrates the fusion itself to be performed by the LLVM built-in loop fusion algorithm.

For this iteration of the loop fusion pre-pass implemented, there are some edge cases that the pass will not handle, and instead will leave untouched. These cases not in scope are:

- the if condition contains a compound predicate
- the if block contains more code than just a single for loop
- there are inner loops nested in any of the for loops

For example, if the loop-fusion pass is run on code with an if condition within the second for loop, this code would not be transformed.

[1] https://github.com/kwwangkw/loopfusion-prepass

Thus, it is proposed that further extensions of this work should aim towards addressing these restrictions placed on possible inputs to the pre-pass.

5 Algorithm

Algorithm 1 on the following page gives a high-level overview of the pre-pass code. Table 1 defines some of the data structures used in the algorithm. Note that the algorithm described does not address the edge cases listed in the previous section that were omitted from the scope.

Data Structure	Purpose
blockToLoopHeader	Maps a basic block in a loop to the header block of the loop
loopEnds	Maps the last basic block in a loop to the header block of the loop
blocks	Vector of all BasicBlock pointers in a function, effectively in dominance order
forIfFors	Vector of BasicBlock pointers to header blocks of the first for loop in each for-if-for instance in the code
forIfForsHeaderToEnd	Maps the header block of the first for loop in a for-if-for instance to the last block of the second for loop

Table 1: Table of data structures used in the algorithm

In Algorithm 1, on lines 2-9, the "blockToLoopHeader" map and "loopEnds" map are populated by iterating through blocks in each loop in the function. Lines 10-12 populates "blocks", a vector of all blocks in the function, assumed to be in dominance order.

On lines 13-25, the algorithm iterates through the "blocks" vector to identify instances of the for-if-for pattern. Line 17 examines a window of blocks and their names to check if they match a certain pattern that identifies it as a for-if-for pattern. If there is a match, then the algorithm uses "loopEnds" to access the header block of the first for loop in the for-if-for instance. That header block is then pushed to the "forIfFors" vector. The "forIfForsHeaderToEnd" map then maps that header block to the block that ends the for-if-for instance.

Lines 26-43 of the algorithm transforms the code by "shuffling" the basic blocks to match a for-for-if configuration that can then allow the for loops to be fused by the LLVM loop fusion algorithm. Line 26 iterates through all header blocks in "forIfFors", since each header block indicates an instance of the for-if-for pattern. For each such for-if-for instance, line 27 declares

Algorithm 1 For-If-For Transformation Pre-Pass

```
1: procedure PRE-PASS(function)
2:     if there are loops in the function then
3:         for loop in function do
4:             for block in loop do
5:                 populate blockToLoopHeader map
6:                 populate loopEnds map
7:             end for
8:         end for
9:     end if
10:    for block in function do
11:        populate blocks vector
12:    end for
13:    for block in blocks do
14:        if iteration count > 2 then
15:            continue
16:        end if
17:        check if the current block is a "for.body" block, if the block 2 blocks back is an "if.then" block, and if the block 3 blocks back is a "for.end" block
18:        if previous checks pass then
19:            headerbb := loopEnds["for.end" block]
20:            add headerbb to forIfFors
21:            if successor of "for.end" is "if.end" block then
22:                forIfForsHeaderToEnd[headerbb] = "if.end" block
23:            end if
24:        end if
25:    end for
26:    for headerbb in forIfFors do
27:        declare forIfForBlocks, which will be a vector of all blocks relevant to the current for-if-for
28:        flag := false
29:        for block in blocks do
30:            if block == headerbb then
31:                flag := true
32:            end if
33:            if flag == true then
34:                push block into forIfForBlocks
35:            end if
36:            if block == forIfForsHeaderToEnd[headerbb] then
37:                break
38:            end if
39:        end for
40:        if forIfForBlocks size == 8 then
41:            shuffle the ordering of blocks in forIfForBlocks by setting new successors of each blocks until for-for-if structure is achieved
42:        end if
43:    end for
44: end procedure
```

a vector, "forIfForBlocks", that will store all the blocks relevant to that for-if-for pattern. "forIfForBlocks" is populated in lines 29-39. To summarize this process, the algorithm iterates through all blocks in the function until it reaches the header block of the current iteration, then pushes all blocks into "forIfForBlocks" until it has pushed the end block that corresponds with the header block.

Line 40 checks if "forIfForBlocks" has exactly 8 blocks: if it has more, it entails that there could be if statements or loops nested within the second for loop of the for-if-for instance, an edge case that is outside the scope of this project. Then, if this check passes, the blocks listed in "forIfForBlocks" will be "shuffled" by reassigning their successors. After "shuffling", the current for-if-for statement has been converted into its corresponding for-for-if configuration.

One note is that "blockToLoopHeader" was never used in the algorithm, but was left in the code in case it could be useful in future projects extending this one to address the edge cases mentioned previously.

6 Results

To test the proposed algorithm, the LLVM pass developed was run on five different benchmarks to test against several evaluation metrics. The benchmarks consist of for-if-for structures. Listed below are descriptions of each benchmark:

1. A singular for-if-for example
2. Two for-if-for statements, with no intervening code statements
3. A for-if-for with array indexing (both loads and stores)
4. Two for-if-for statements with an intervening for statement that can be fused
5. Many, many back-to-back for-if-for statements and intervening code to stress test

The evaluation metrics used to measure the success of the algorithm implementation are:

1. The number of loops fused
2. Code size: number of basic blocks
3. Code size: number of instructions
4. Runtime

6.1 Number of Loops Fused

Table 2 compares the number of loops present in each benchmark when no transformation is used, when only the LLVM fusion algorithm is run, and when the proposed pre-pass is run, followed by the LLVM fusion algorithm. As shown in the table, when the original loop fusion algorithm is run, there is relatively no significant change in the number of loops fused. The proposed algorithm increases the number of loops fused. Compared to the LLVM fusion, the Pre-pass + LLVM fusion algorithm minimally doubles the number of loops fused. Therefore, it can be concluded that the proposed algorithm loosens control equivalency and enables more loops to be fused.

Benchmark	No Change	LLVM Fusion	Pre-Pass + LLVM Fusion
1	2	2	1
2	4	4	2
3	2	2	1
4	5	4	2
5	30	24	12

Table 2: Total number of loops fused before and after transformation

Benchmark	LLVM Fusion	Pre-Pass + LLVM Fusion
1	13	9
2	25	17
3	13	9
4	26	18
5	151	103

Table 3: Number of basic blocks after transformation

6.2 Code Size: Number of Basic Blocks

Table 3 compares the number of basic blocks in each benchmark when only the LLVM fusion algorithm is run against when the proposed pre-pass and the LLVM fusion algorithm are both executed. As shown in the table, there is a significant decrease in the number of basic blocks when ran with the proposed algorithm within each of the benchmarks. The proposed algorithm decreases the number of basic blocks by approximately 31%.

Benchmark	LLVM Fusion	Pre-Pass + LLVM Fusion
1	25	22
2	43	35
3	32	28
4	67	59
5	377	329

Table 4: Number of instructions after transformation

6.3 Code Size: Number of Instructions

Table 4 compares the number of instructions in each benchmark when only the LLVM fusion algorithm is run agaist when the proposed pre-pass is run, followed by the LLVM fusion algorithm. As shown in the table, there is a significant decrease in the number of instructions when ran with the proposed algorithm within each of the benchmarks. The proposed algorithm decreases the number of instructions by approximately 13%. Please refer to the Appendix for more visualizations.

6.4 Runtime

After running each benchmark against the LLVM pass and the proposed algorithm, there was no discernible change in runtime. Overall, there was no significant increase or decrease in runtime for any of the benchmarks.

7 Conclusion

In conclusion, this pass loosens the conditions necessary for the LLVM loop fusion pass to fuse loops. This pre-pass specifically loosens the control flow equivalent fusion condition and allows non-control flow equivalent loops to be fused. This is done by identifying and restructuring code through checking for "for-if-for" statements. As seen from the benchmarks upon which this pass has been tested, there is a significant increase in number of loops fused as well as a significant decrease in code size and complexity.

References

Bob Blainey, Christopher Barton, and José Nelson Amaral. 2005. Removing impediments to loop fusion through code transformations. In *Languages and Compilers for Parallel Computing*, pages 309–328, Berlin, Heidelberg. Springer Berlin Heidelberg.

Appendix

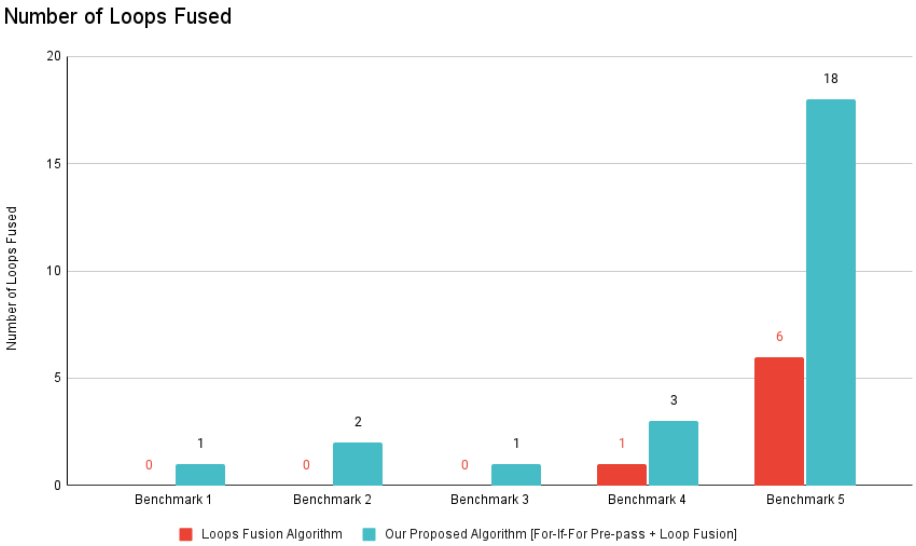

Figure 3: Number of Loops Fused

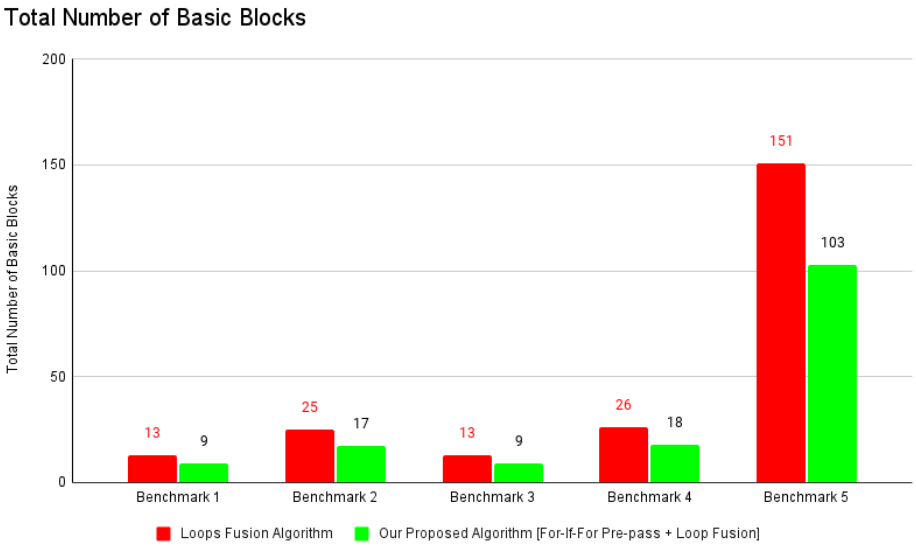

Figure 4: Total Number of Basic Blocks

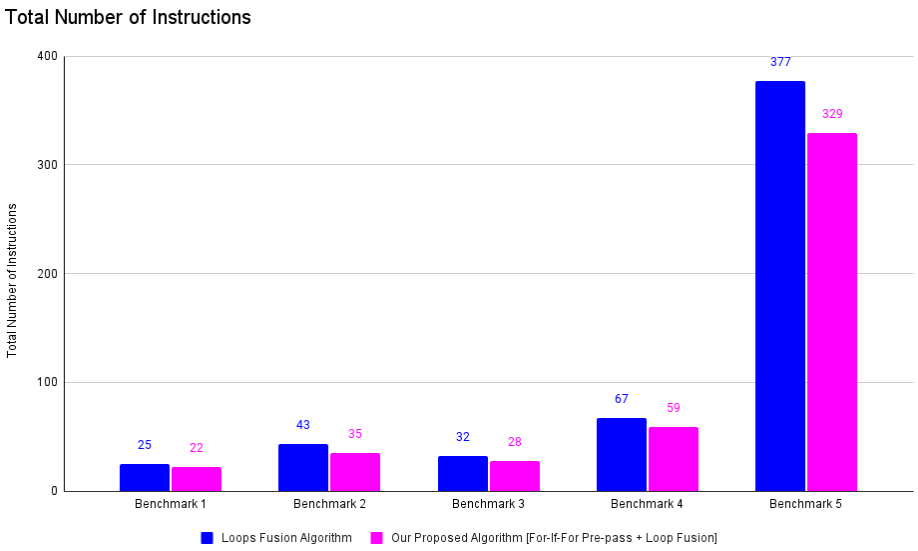

Figure 5: Total Number of Instructions