Final Report: Implementation of Dominance-Based Duplication Simulation in LLVM

Kai Franz and Grey Golla

Motivation

When performing optimizations across a merge point, it is necessary to duplicate code to make use of information specific to each control-flow path.

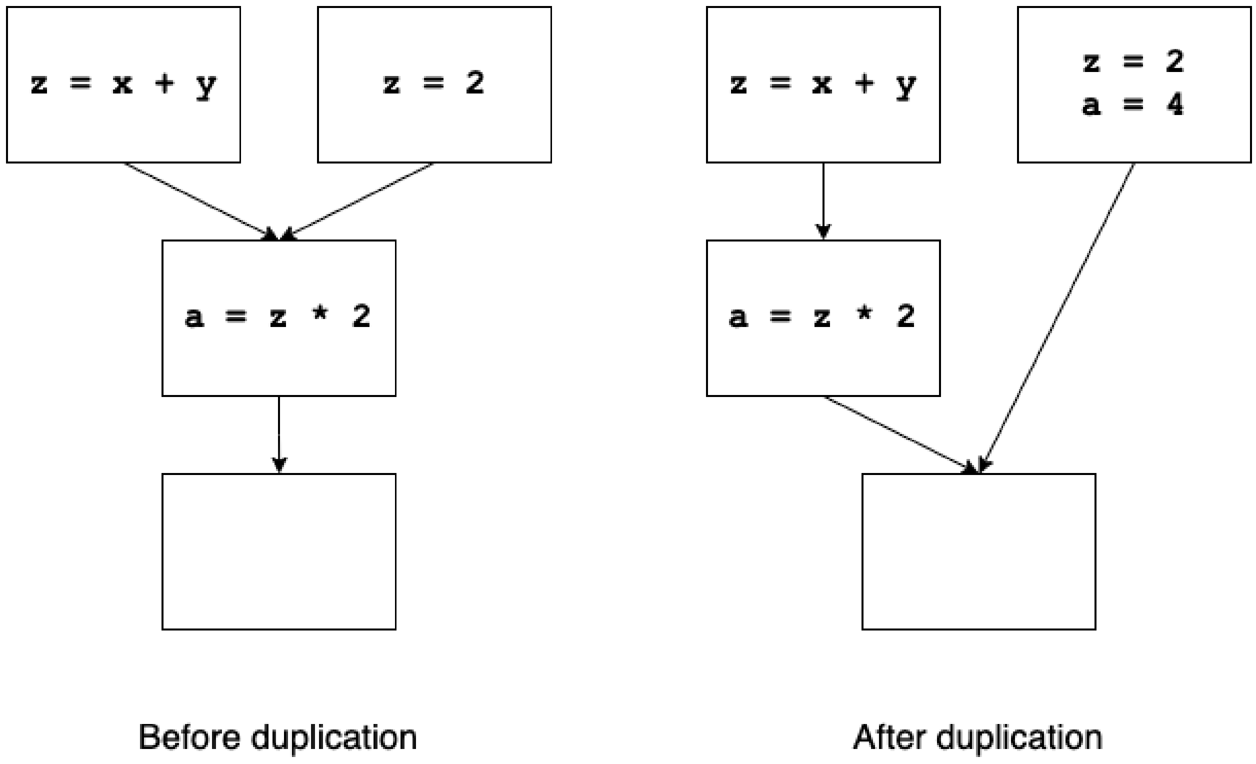


Figure 1: Example constant folding optimization enabled by duplication

Here, duplicating a = z * 2 allows us to make use of the fact that z = 2 in the right path, so we can constant fold z into the statement a = z * 2, turning it into a = 4. On the right path, the computation a = z * 2 can be skipped over, reducing the number of instructions that have to be executed. On the left path, however, since no assumptions can be made about the value of z, z * 2 must still be computed. As such, duplicating code across every merge point would be unhelpful, since it would only sometimes enable optimizations such as constant folding. Because duplication also causes an increase in code size, it can also cause subsequent optimization passes to take longer, as well as slowing down execution. We address this problem by duplicating code only when it enables optimizations to be made.

Related Work

As mentioned in the original DBDS paper, there are several related works. Meuller and Whalley introduced a code replication optimization, where unconditional branches are eliminated by duplicating code [3]. This optimization is related to DBDS because both optimizations involve duplicating code across branches to reduce dynamic instruction counts at the expense of code size.

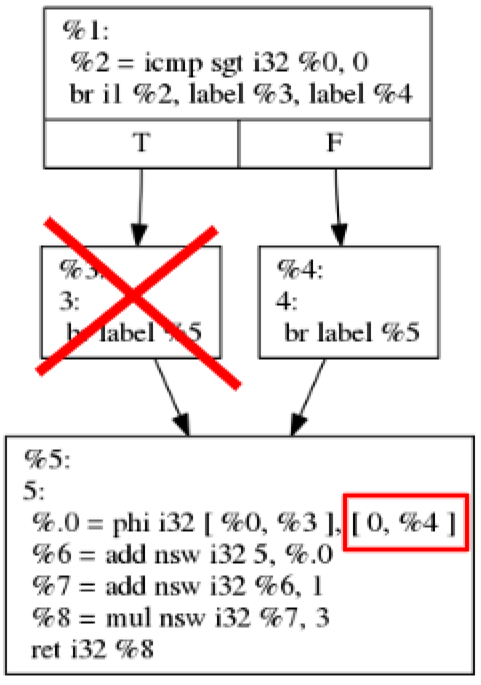
Chambers introduces another duplication optimization in the Self compiler where code is duplicated from a merge point into its predecessors and then specialized based on code in its predecessor [1]. This splitting technique was used to apply devirtualization to polymorphic code in an object-oriented language [1]. Unlike DBDS, splitting did not simulate duplicating; instead, it used either reluctant or eager splitting to determine when to perform code duplication. Under reluctant splitting, merge points are initially merged together, and the compiler later backtracks and splits them apart if it determines that splitting would be beneficial. Under eager splitting, the compiler always splits merge points, merging them together if it realizes afterwards that the two paths generated the exact same code. Eager splitting can easily cause an exponential increase in code size. The Self compiler also supports divided splitting, where eager splitting is used for hot paths while reluctant splitting is used for uncommon paths. Self's splitting optimizations differ from DBDS in that they are only used to perform monomorphism optimizations, and that DBDS's simulation is much faster than reluctant splitting, allowing the compiler to check what optimizations are enabled by duplication in all cases rather than relying on a heuristic to decide when to check.

Technical Details

Simulating the Optimizations

The optimization pass is split into two phases for each function to be optimized. In the first phase, we walk the dominance tree and find pairs of basic blocks that fit the necessary pattern: the block to be duplicated has multiple predecessors, and the predecessor we're looking at only has one successor. Once we find a pair, we generate a *synonym map* for that pair. The synonym map takes values found in the original basic block and maps them to their corresponding values assuming the code is duplicated. This map is initialized to hold a mapping for each phi node in the duplicated block: these map to the values they take given the correct predecessor block.

Once we've generated the synonym map, we have all the information we need to duplicate the code into the predecessor block, so we can run optimizations on the duplicated code. Each optimization pass that we run will add entries to the synonym map, from instructions in the original block to new instructions or constants. See Figure 2 for an example of the synonym map after a constant folding pass. It's important to note that at this point optimizations are simulated: they only affect the synonym map, and do not change the underlying CFG. Once all optimization passes are complete, the final synonym map is stored to be evaluated and applied later. This results in one synonym map for each basic block pair that could be duplicated.



```
%1:
 %2 = icmp sgt i32 %0, 0
 br i1 %2, label %3, label %4
```

```
%3:
3:
 br label %5
```

```
%4:
4:
 br label %5
```

```
%5:
5:
 %.0 = phi i32 [ %0, %3 ], [ 0, %4 ]
 %6 = add nsw i32 5, %.0
 %7 = add nsw i32 %6, 1
 %8 = mul nsw i32 %7, 3
 ret i32 %8
```

CFG for 'foo' function

Synonym Map	
%.0	i32 0
%6	5
%7	6
%8	18

Figure 2: Example synonym map during simulation of merging %5 into %4.

See figure 3 for the results of applying that synonym map to the CFG.

Applying the Simulation Results

Once all of the duplicated opportunities have been simulated, we have a set of synonym maps that we can apply to the CFG in the second phase. This would be where we apply heuristics to choose whether to apply the duplication, but in the current implementation we apply all of them. For each synonym map, we apply the code duplication:

First, we split the duplicated basic block at its terminating instruction. This provides us with a new basic block with all the successors of the duplicated block, so it will become the only successor for both copies of the duplicated code. We call this block the “phiBB” for reasons that will be clear soon.

Next, we iterate over the instructions in the basic block to be duplicated and replace them with the instruction from the synonym map, if applicable. We also need to replace the operands of this instruction with appropriate values. The need for this is simple: when we duplicate instructions, all uses of that instruction within the duplicated code need to now use the new instruction. This is conceptually simple to accomplish if we add each cloned instruction to the synonym map, we can just iterate over each operand and replace it with the mapped value. Anything not in the synonym map came from outside the current block and doesn't need to be modified. Unfortunately, this has a critical pitfall that we explain in the "surprises" section.

The operand modification explained above handles local uses of this instruction, but global uses need to be handled as well. Because there are now two places where the value is computed, we need to insert a phi node to choose which one to use. We generate this phi node in the phiBB, which has the two computations as its only two predecessors. Any uses "seen outside" the current basic block need to be changed to point to the newly generated phi node, including phi nodes inside the block that is currently being duplicated.

Finally, we can place this duplicated and modified instruction in the (former) predecessor block.

Optimization Specifics

This subsection goes over the implementation of each simulated optimization. Generally, these passes are extremely simple because we are only considering local optimizations.

Constant folding is implemented by using LLVM's "ConstantFoldInstruction" function. We do this by first checking if the operands are constants, then, if they are, cloning the instruction and applying the synonym map to the operands. This allows us to *correctly* support all of the constant folding operations that LLVM supports instead of hand-making a buggy subset of them.

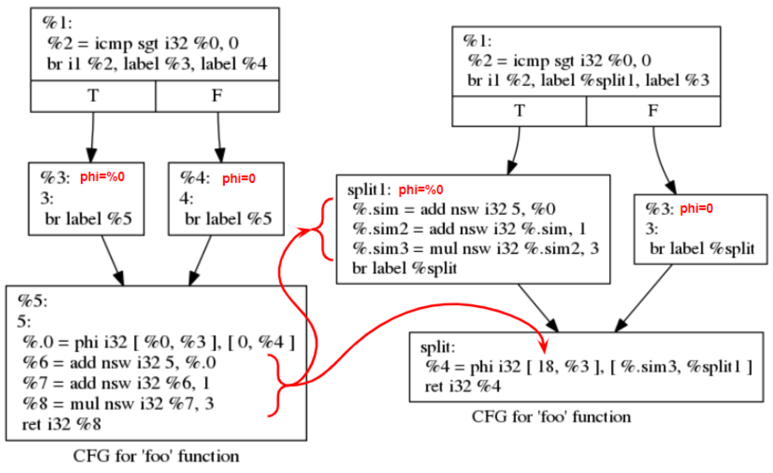
Strength reduction is implemented by hand, the same way we implemented it in lab 1.

Read Elimination, as described in the paper, seems to be a subset of common subexpression elimination. As such, implementing it inside of duplicated code essentially implements partial redundancy elimination. We implemented CSE using a local version of the “available expression” pass that we wrote for lab 2, using the provided “Expression” class.

Experimental Results

We did not implement heuristics to choose only some of the code duplication opportunities, so it would be impossible to test our system on meaningful benchmarks due to the explosion in code size. As such, we focused on making and testing microbenchmarks that demonstrate the correctness of our pass.

Figure 3 shows the results of duplicating code into one branch with no improvement, and into the other branch which allows significant constant folding opportunities. Along the right path through the code, we remove 3 computations, while the left path is completely unaffected.



```
%1:
 %2 = icmp sgt i32 %0, 0
 br i1 %2, label %3, label %4

%3:
3:
 br label %5

%4:
4:
 br label %5

%5:
5:
 %.0 = phi i32 [ %0, %3 ], [ 0, %4 ]
 %6 = add nsw i32 5, %.0
 %7 = add nsw i32 %6, 1
 %8 = mul nsw i32 %7, 3
 ret i32 %8
```

```
%1:
 %2 = icmp sgt i32 %0, 0
 br i1 %2, label %split1, label %3

split1:
 %.sim = add nsw i32 5, %0
 %.sim2 = add nsw i32 %.sim, 1
 %.sim3 = mul nsw i32 %.sim2, 3
 br label %split

%3:
3:
 br label %split

split:
 %4 = phi i32 [ 18, %3 ], [ %.sim3, %split1 ]
 ret i32 %4
```

Figure 3: Microbenchmark of constant folding

Figure 4 shows a microbenchmark for strength reduction that is enabled by code duplication. While we do not reduce the number of instructions in this case, the right hand side now only has to compute an inexpensive left shift operation instead of the multiply formerly required by both sides.

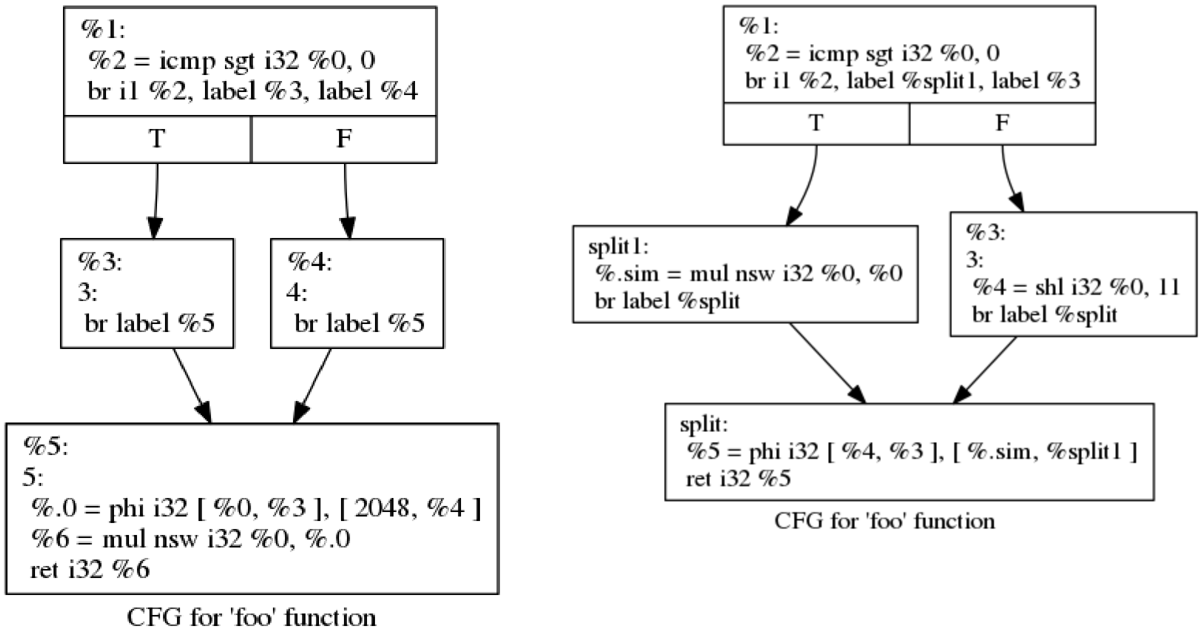
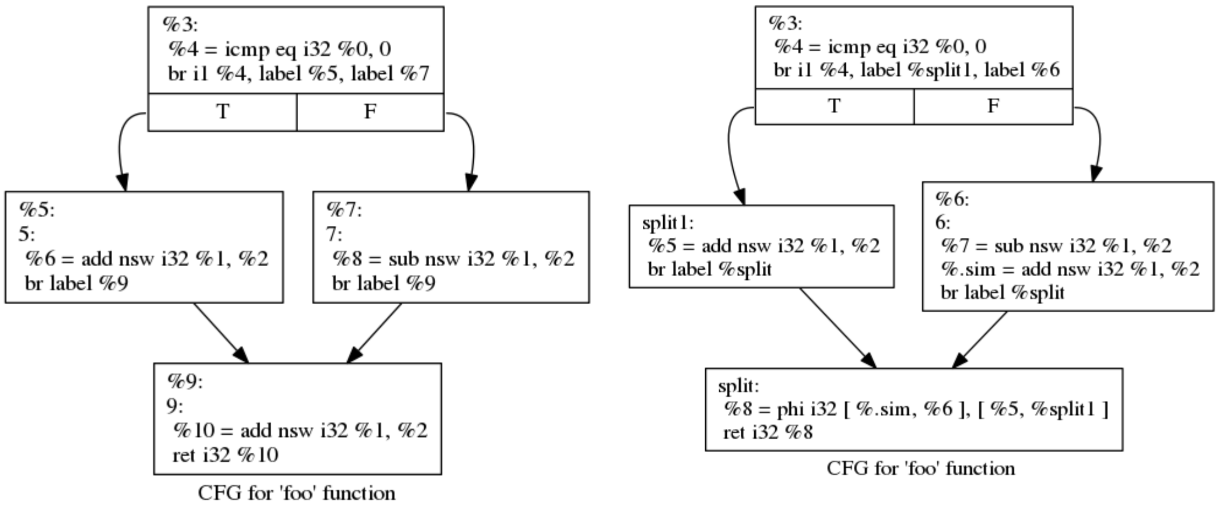

```
%1:
 %2 = icmp sgt i32 %0, 0
 br i1 %2, label %3, label %4
```
```
%3:
3:
 br label %5
```
```
%4:
4:
 br label %5
```
```
%5:
5:
 %.0 = phi i32 [ %0, %3 ], [ 2048, %4 ]
 %6 = mul nsw i32 %0, %.0
 ret i32 %6
```
```
%1:
 %2 = icmp sgt i32 %0, 0
 br i1 %2, label %split1, label %3
```
```
split1:
 %.sim = mul nsw i32 %0, %0
 br label %split
```
```
%3:
3:
 %4 = shl i32 %0, 11
 br label %split
```
```
split:
 %5 = phi i32 [ %4, %3 ], [ %.sim, %split1 ]
 ret i32 %5
```

Figure 4: Duplication enabling strength reduction

Figure 5 shows a microbenchmark showing common subexpression elimination. The left path computes “%1 + %2” twice, while the right path only computes it once, in a textbook example for partial redundancy elimination. While another solution, such as lazy code motion, would catch cases like this, this pass allows the subexpressions to be detected completely locally instead of requiring a complicated, multi-pass system like lazy code motion.



```
%3:
 %4 = icmp eq i32 %0, 0
 br i1 %4, label %5, label %7

%5:
5:
 %6 = add nsw i32 %1, %2
 br label %9

%7:
7:
 %8 = sub nsw i32 %1, %2
 br label %9

%9:
9:
 %10 = add nsw i32 %1, %2
 ret i32 %10
```

```
%3:
 %4 = icmp eq i32 %0, 0
 br i1 %4, label %split1, label %6

split1:
 %5 = add nsw i32 %1, %2
 br label %split

%6:
6:
 %7 = sub nsw i32 %1, %2
 %.sim = add nsw i32 %1, %2
 br label %split

split:
 %8 = phi i32 [ %.sim, %6 ], [ %5, %split1 ]
 ret i32 %8
```

Figure 5: Microbenchmark for “read elimination” or common subexpression elimination

Surprises and Lessons Learned

Because the DBDS optimization involves significant changes to a program's control-flow graph, there were many SSA-related details that we had to address when moving instructions between basic blocks. For example, we had to construct an extra basic block at the merge point that holds phi nodes for the instructions that were duplicated across the predecessor paths.

One of the biggest challenges was tracking down all of the global uses of the instructions that we were duplicating. As stated in the “technical details” section, we need to replace the uses of the duplicated instruction with the newly generated phi node. LLVM tracks the uses of the instruction, so we can use “replaceUsesWithIf()” to find the global uses of the instruction.

Unfortunately, this misses a critical location that hides uses of these instructions: other synonym maps. If a phi node references a value that was generated in a basic block that is then duplicated, the entry in the second synonym map for that phi node will still hold a reference to the original computation. This is because LLVM does not know that the synonym map is a “use” of the instruction. We solved this by generating a global synonym map mapping from the original instruction to the generated phi node, then, when replacing values that come from outside the current basic block, we check the global synonym map and then check the local synonym map.

Overall, we learned that modifying a program's control-flow graph in SSA form can be prohibitively complex. If we did this again, we would run the reg2mem pass before our pass to de-SSA the program, which according to the LLVM documentation is "intended [to] make CFG hacking much easier." [4] This would allow us to simply copy instructions that read/write to each memory location, totally ignoring the source(s) or use(s) of the value. The mem2reg pass could then be run to convert the program back into valid SSA form.

Conclusion

Our implementation of Dominance Based Duplication Simulation is complete according to the goals set out at the beginning of this project. We are able to simulate and then apply several optimizations on duplicated basic blocks. In terms of this being a useful addition to LLVM, the answer is certainly “not yet,” due to the lack of heuristics to choose which simulations to apply. The bones, however, are all there and we believe that implementing the code duplication and simulation application step was the biggest barrier to this being a useful optimization. With future work adding heuristic choices of duplications to actually complete, as well as work adding more optimizations to the pass, this could be a meaningful optimization pass for real code.

Distribution of Total Credit

We believe that we each deserve 50% of the credit. All of the work was completed with “pair programming” sitting together, and we worked together to solve all of the many, many problems that came up during implementation.

Bibliography

[1] C. Chambers, “The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages,” thesis, 1992.

[2] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, and H. Mössenböck, “Dominance-based duplication simulation (DBDS): Code duplication to enable compiler optimizations,” *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018.

[3] F. Mueller and D. B. Whalley, “Avoiding unconditional jumps by code replication,” *ACM SIGPLAN Notices*, vol. 27, no. 7, pp. 322–330, 1992.

[4] “LLVM's analysis and transform passes¶,” *LLVM's Analysis and Transform Passes - LLVM 15.0.0git documentation*. [Online]. Available: https://llvm.org/docs/Passes.html#reg2mem-demote-all-values-to-stack-slots. [Accessed: 27-Apr-2022].