Dominance-Based Duplication Simulation

Kai Franz and Grey Golla

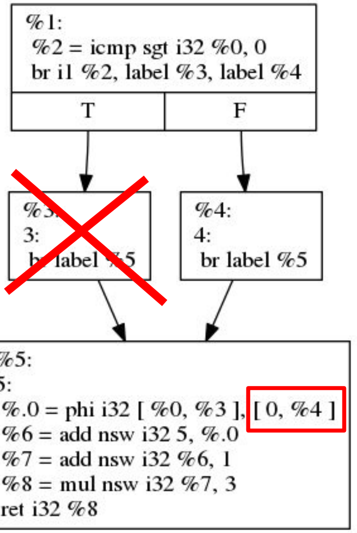
https://kai-franz.github.io/dbds/

Problem Statement

- Code duplication from shared children allows optimizations
- Duplicating all children is too slow and wasteful
- Solution: *Simulate* duplication and only apply the duplications that are “worth it”
- Two phases: simulate all possible duplications, then apply the worthwhile ones

Phase 1: Simulate Optimizations

- For each pair of (predBB, BB)
  - Generate a *Synonym Map* of the ϕ nodes in BB as if predBB were the only predecessor
  - Find optimizations in BB, add them to the synonym map
- Each simulation is completely independent*
- The synonym map holds all the information needed to apply the optimizations later



```
%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 for (%4, %5)	
%.0	i32 0
%6	5
%7	6
%8	18

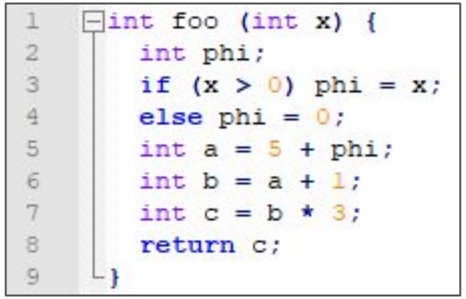
Phase 2: Applying Optimizations

- For each instruction in the duplicated block
  - Look up in the synonym map:
    - If it should be replaced with a different instruction (e.g. strength reduction), replace it
    - If it should be deleted (e.g. replaced with a constant), delete it
    - Otherwise, clone the instruction
  - For each operand:
    - Check if the operand is in the synonym map* and replace it
  - Handle the uses of the variable
    - Add the new instruction to the synonym map for future *local* instructions to see
    - If the variable is used outside the current BB, see below
  - Add the new instruction to the end of the predecessor BB

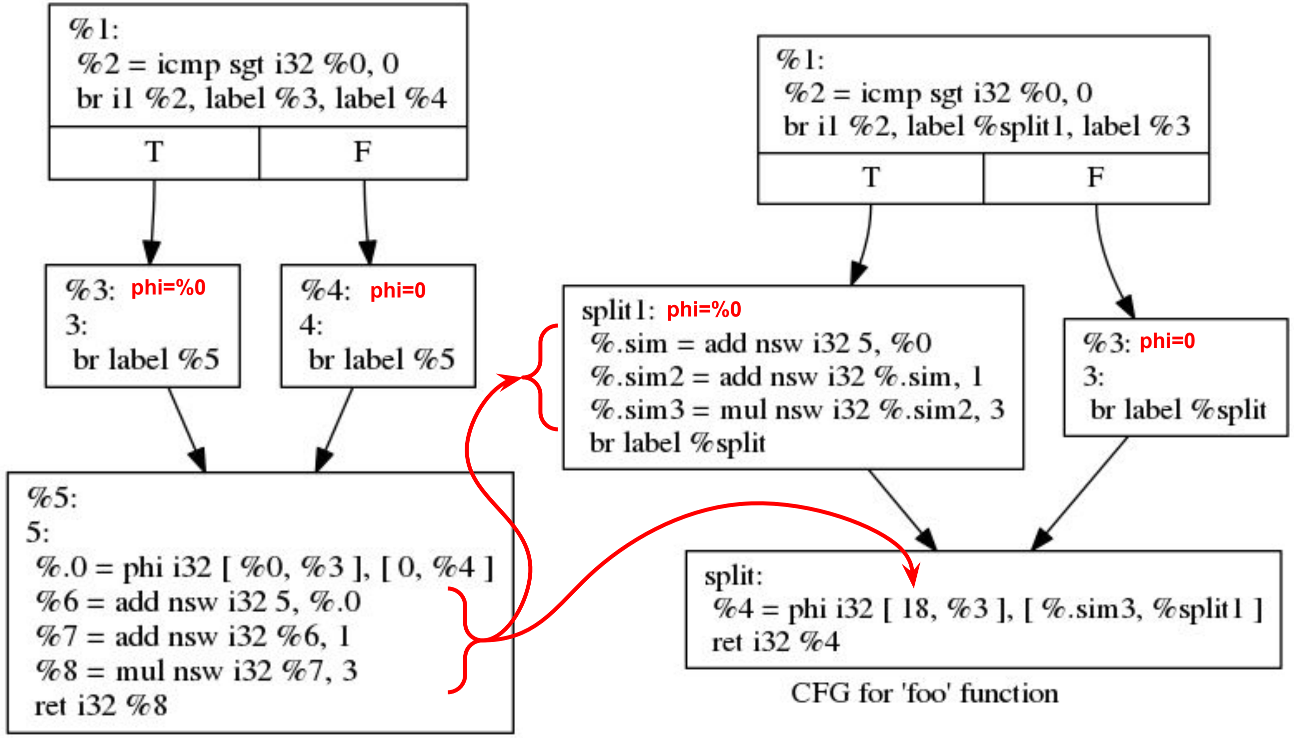
Phase 2: Duplicating Code in SSA

- Need to deal with vars that are seen “outside” the current BB
  - Referenced in other BB
  - Referenced by the phi nodes of the current BB
- Add ϕ node for each var
  - Place in new “phiBB” that is the only successor of the duplicated code
  - This successor inherits all old successors of the duplicated BB
- Replace uses of var with uses of that ϕ node
  - Except if that use is in the duplicated BB, unless it’s in a ϕ node
  - Except in the ϕ node we just generated
  - *Including in future optimizations we’ve planned*

A Trivial Example



```c
int foo (int x) {
    int phi;
    if (x > 0) phi = x;
    else phi = 0;
    int a = 5 + phi;
    int b = a + 1;
    int c = b * 3;
    return c;
}
```



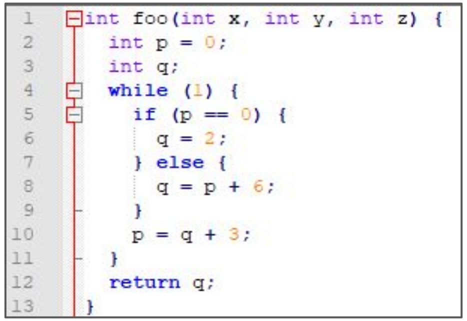
```
%1:
 %2 = icmp sgt i32 %0, 0
 br i1 %2, label %3, label %4

%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

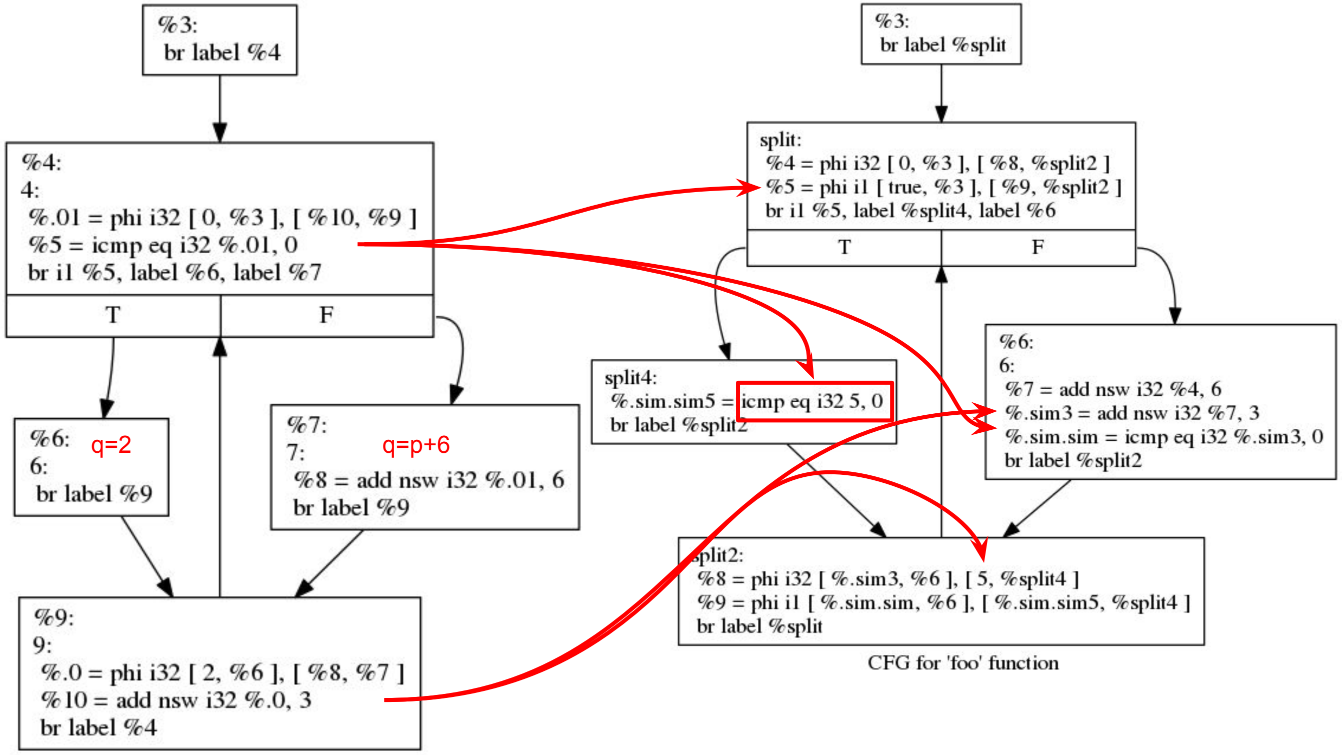
```
%1:
 %2 = icmp sgt i32 %0, 0
 br i1 %2, label %split1, label %3

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
```

CFG for 'foo' function

Loop and Interacting Duplications



```c
int foo(int x, int y, int z) {
  int p = 0;
  int q;
  while (1) {
    if (p == 0) {
      q = 2;
    } else {
      q = p + 6;
    }
    p = q + 3;
  }
  return q;
}
```



```
%3:
 br label %4

%4:
4:
 %.01 = phi i32 [ 0, %3 ], [ %10, %9 ]
 %5 = icmp eq i32 %.01, 0
 br i1 %5, label %6, label %7

%6:
6:
 br label %9

%7:
7:
 %8 = add nsw i32 %.01, 6
 br label %9

%9:
9:
 %.0 = phi i32 [ 2, %6 ], [ %8, %7 ]
 %10 = add nsw i32 %.0, 3
 br label %4
```

CFG for 'foo' function

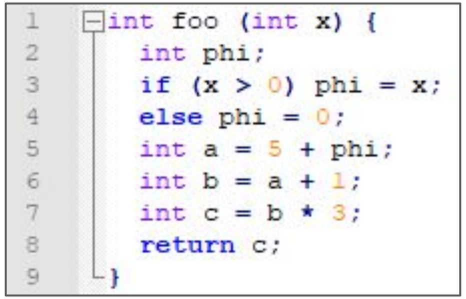
```
%3:
 br label %split

split:
 %4 = phi i32 [ 0, %3 ], [ %8, %split2 ]
 %5 = phi i1 [ true, %3 ], [ %9, %split2 ]
 br i1 %5, label %split4, label %6

split4:
 %.sim.sim5 = icmp eq i32 5, 0
 br label %split2

%6:
6:
 %7 = add nsw i32 %4, 6
 %.sim3 = add nsw i32 %7, 3
 %.sim.sim = icmp eq i32 %.sim3, 0
 br label %split2

split2:
 %8 = phi i32 [ %.sim3, %6 ], [ 5, %split4 ]
 %9 = phi i1 [ %.sim.sim, %6 ], [ %.sim.sim5, %split4 ]
 br label %split
```

CFG for 'foo' function

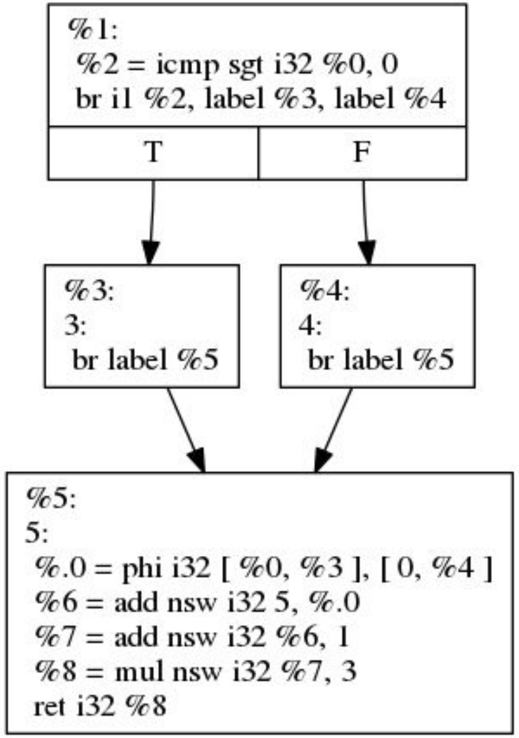
Future Work

- Use cost heuristic to determine when duplication is beneficial
    - Duplicating increases code size
        - Increased workload for later optimization passes
        - Can use JIT profiling information to guide optimization of hot paths
- Better clean-up of simulation artifacts
    - Detect new optimizations allowed by the application of multiple simulated optimizations
    - Detect simulation interactions at simulation time?
- More simulated optimizations
    - Paper lists Conditional Elimination
    - Any local optimization could be beneficial

Generation of extra basic blocks



```
int foo (int x) {
   int phi;
   if (x > 0) phi = x;
   else phi = 0;
   int a = 5 + phi;
   int b = a + 1;
   int c = b * 3;
   return c;
}
```



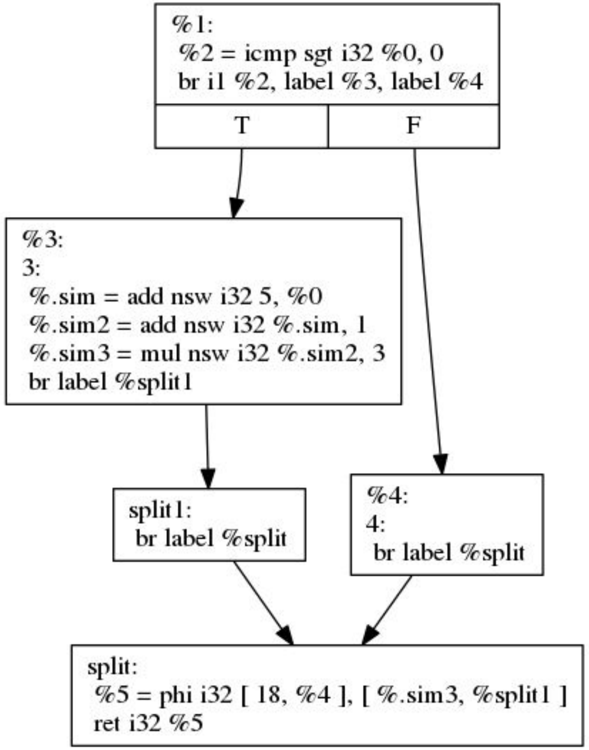
```
%1:
 %2 = icmp sgt i32 %0, 0
 br i1 %2, label %3, label %4
```

```
%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



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

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

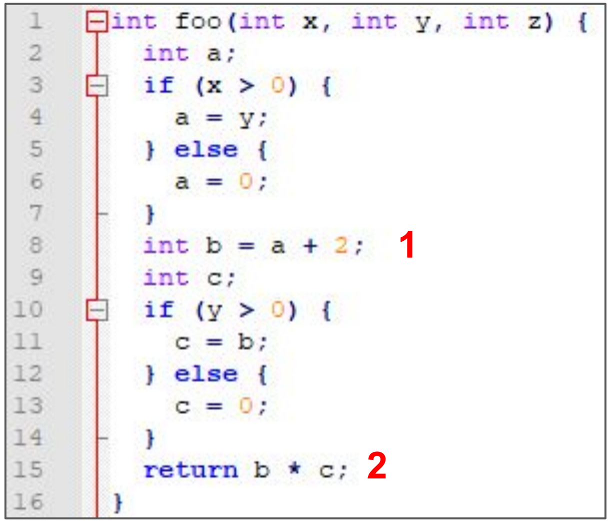
```
split1:
 br label %split
```

```
%4:
4:
 br label %split
```

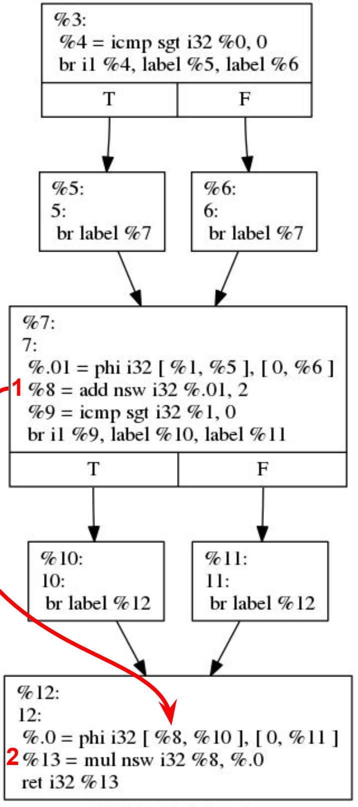
```
split:
 %5 = phi i32 [ 18, %4 ], [ %.sim3, %split1 ]
 ret i32 %5
```

CFG for 'foo' function

Multiple Optimizations in a row

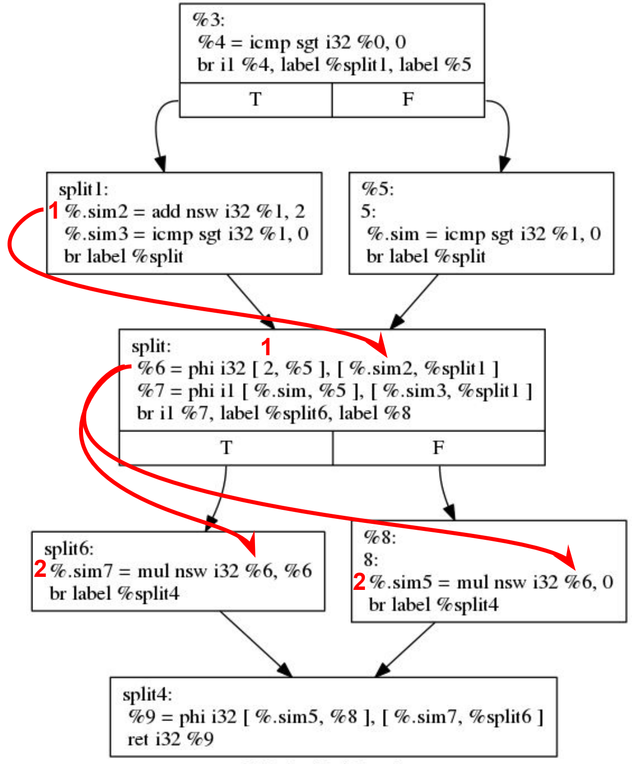


```
int foo(int x, int y, int z) {
    int a;
    if (x > 0) {
        a = y;
    } else {
        a = 0;
    }
    int b = a + 2;   1
    int c;
    if (y > 0) {
        c = b;
    } else {
        c = 0;
    }
    return b * c;  2
}
```



```
%3:
 %4 = icmp sgt i32 %0, 0
 br i1 %4, label %5, label %6

%5:
5:
 br label %7

%6:
6:
 br label %7

%7:
7:
 %.01 = phi i32 [ %1, %5 ], [ 0, %6 ]
1 %8 = add nsw i32 %.01, 2
 %9 = icmp sgt i32 %1, 0
 br i1 %9, label %10, label %11

%10:
10:
 br label %12

%11:
11:
 br label %12

%12:
12:
 %.0 = phi i32 [ %8, %10 ], [ 0, %11 ]
2 %13 = mul nsw i32 %8, %.0
 ret i32 %13
```

CFG for 'foo' function



```
%3:
 %4 = icmp sgt i32 %0, 0
 br i1 %4, label %split1, label %5

split1:
1 %.sim2 = add nsw i32 %1, 2
 %.sim3 = icmp sgt i32 %1, 0
 br label %split

%5:
5:
 %.sim = icmp sgt i32 %1, 0
 br label %split

split:          1
 %6 = phi i32 [ 2, %5 ], [ %.sim2, %split1 ]
 %7 = phi i1 [ %.sim, %5 ], [ %.sim3, %split1 ]
 br i1 %7, label %split6, label %8

split6:
2 %.sim7 = mul nsw i32 %6, %6
 br label %split4

%8:
8:
2 %.sim5 = mul nsw i32 %6, 0
 br label %split4

split4:
 %9 = phi i32 [ %.sim5, %8 ], [ %.sim7, %split6 ]
 ret i32 %9
```

CFG for 'foo' function