Waddle

Maintaining Canonical Form After Edge Deletion

Eric Fritz

July 17, 2018

University of Wisconsin – Milwaukee

What Does Waddle Maintain? (1)

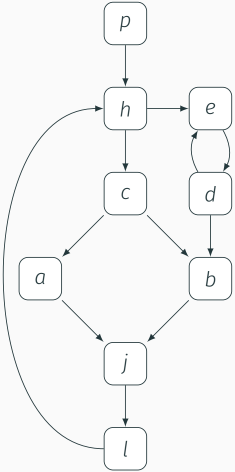
Dominator Tree

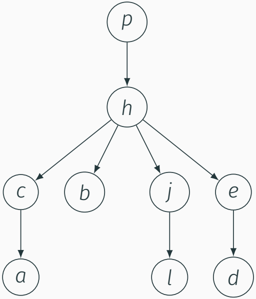
encodes which blocks occur on all paths to another block

What Does Waddle Maintain? (1)

Dominator Tree

encodes which blocks occur on all paths to another block





What Does Waddle Maintain? (2)

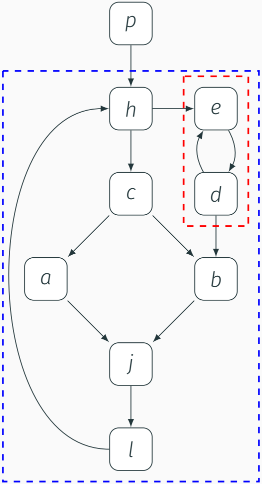
Loop Nesting Forest

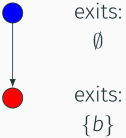
encodes loop body sets · loop exit sets · loop nesting structure

What Does Waddle Maintain? (2)

Loop Nesting Forest

encodes loop body sets · loop exit sets · loop nesting structure





What Does Waddle Maintain? (3)

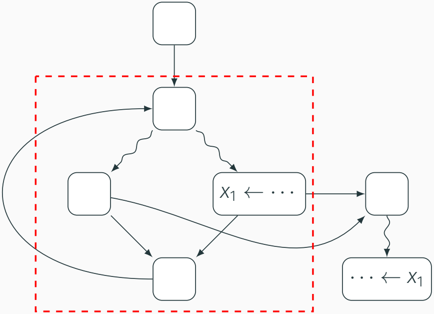
SSA + LCSSA Form

all names defined once · uses of name occur within defining loop

What Does Waddle Maintain? (3)

SSA + LCSSA Form

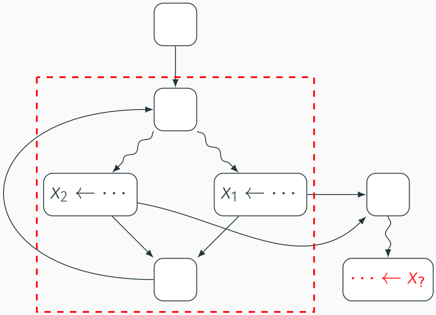
all names defined once · uses of name occur within defining loop



What Does Waddle Maintain? (3)

SSA + LCSSA Form

all names defined once · uses of name occur within defining loop



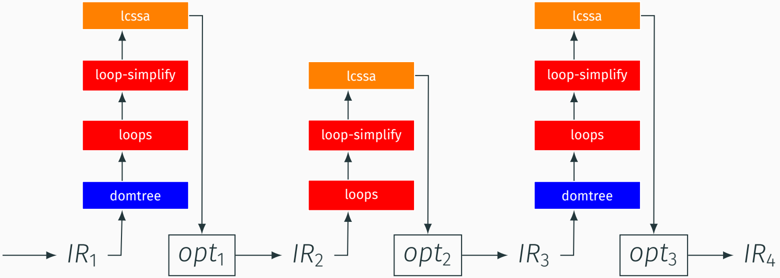
What Does Waddle Maintain? (4)

'Canonical' Properties

LLVM's Loop Simplify Form

Optimization Pipeline Strategies

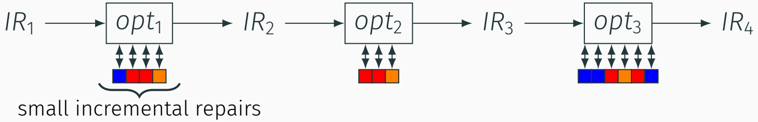
Repair On-Demand (LLVM's Approach)



LLVM 6.0.0 -O2 Passes

ipsccp
globalopt
domtree
mem2reg
deadargelim
domtree
aa
loops
lazy-block-freq
instcombine
simplifycfg
basiccg
globals-aa
prune-eh
inline
functionattrs
domtree
sroa
aa
memoryssa
early-cse-memssa
speculative-
execution
domtree
aa
lazy-value-info
jump-threading
lazy-value-info
correlated-
propagation
simplifycfg
domtree
aa
loops
lazy-block-freq
instcombine
libcalls-shrinkwrap
loops
branch-prob
block-freq
lazy-block-freq
pgo-memop-opt
domtree
aa
loops
lazy-block-freq
tailcallelim
simplifycfg
reassociate
domtree
loops
loop-simplify
lcssa
aa
scalar-evolution
loop-rotate
licm
loop-unswitch
simplifycfg
domtree
aa
loops
lazy-block-freq
instcombine
loop-simplify
lcssa
scalar-evolution
indvars
loop-idiom
loop-deletion
loop-unroll
mldst-motion
aa
memdep
lazy-block-freq
gvn
aa
memdep
memcpyopt
sccp
domtree
demanded-bits
bdce
aa
loops
lazy-block-freq
instcombine
lazy-value-info
jump-threading
lazy-value-info
correlated-
propagation
domtree
aa
memdep
dse
loops
loop-simplify
lcssa
aa
scalar-evolution
licm
postdomtree
adce
simplifycfg
domtree
aa
loops
lazy-block-freq
instcombine
barrier
elim-avail-extern
basiccg
rpo-functionattrs
globalopt
globaldce
basiccg
globals-aa
float2int
domtree
loops
loop-simplify
lcssa
aa
scalar-evolution
loop-rotate
loop-accesses
lazy-block-freq
loop-distribute
branch-prob
block-freq
scalar-evolution
aa
loop-accesses
demanded-bits
lazy-block-freq
loop-vectorize
loop-simplify
scalar-evolution
aa
loop-accesses
loop-load-elim
aa
lazy-block-freq
instcombine
simplifycfg
domtree
loops
scalar-evolution
aa
demanded-bits
lazy-block-freq
slp-vectorizer
instcombine
loop-simplify
lcssa
scalar-evolution
loop-unroll
lazy-block-freq
instcombine
loop-simplify
lcssa
scalar-evolution
licm
globaldce
constmerge
domtree
loops
branch-prob
block-freq
loop-simplify
lcssa
aa
scalar-evolution

Always Canonical (Waddle's Approach)



LLVM Source – LoopUnroll.cpp#L845

```
// If we have a pass and a DominatorTree we should re-simplify impacted loops
// to ensure subsequent analyses can rely on this form.  We want to simplify
// at least one layer outside of the loop that was unrolled so that any
// changes to the parent loop exposed by the unrolling are considered.
if (DT) {
  if (OuterL) {
    // OuterL includes all loops for which we can break loop-simplify, so
    // it's sufficient to simplify only it (it'll recursively simplify inner
    // loops too).

    // TODO: That potentially might be compile-time expensive.  We should try
    // to fix the loop-simplified form incrementally.
    simplifyLoop(OuterL, DT, LI, SE, AC, PreserveLCSSA);
  } else {
    // Simplify loops for which we might've broken loop-simplify form.
    for (Loop *SubLoop : LoopsToSimplify)
      simplifyLoop(SubLoop, DT, LI, SE, AC, PreserveLCSSA);
  }
}
```

Canonical Form Loop Properties

Canonical Properties

Dedicated Preheader

enables easy + efficient instruction hoisting

Canonical Properties

Dedicated Preheader

enables easy + efficient instruction hoisting

Dedicated Exit Blocks

enables easy + efficient effect sinking

Canonical Properties

Dedicated Preheader

enables easy + efficient instruction hoisting

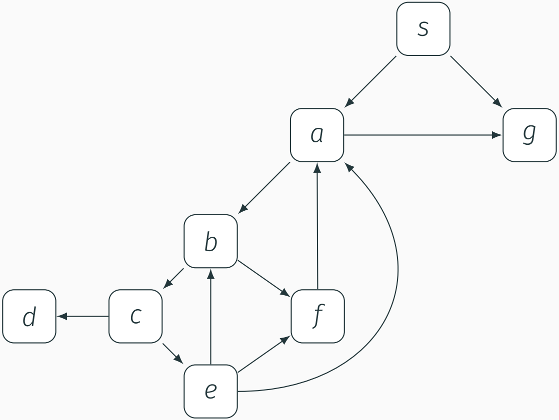
Dedicated Exit Blocks

enables easy + efficient effect sinking

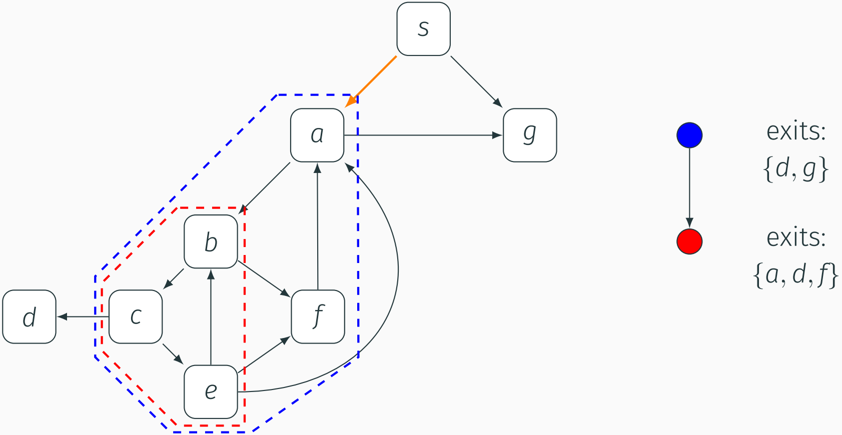
Unique Backedge + Latch

make destruction of loop unambiguous

Canonicalization

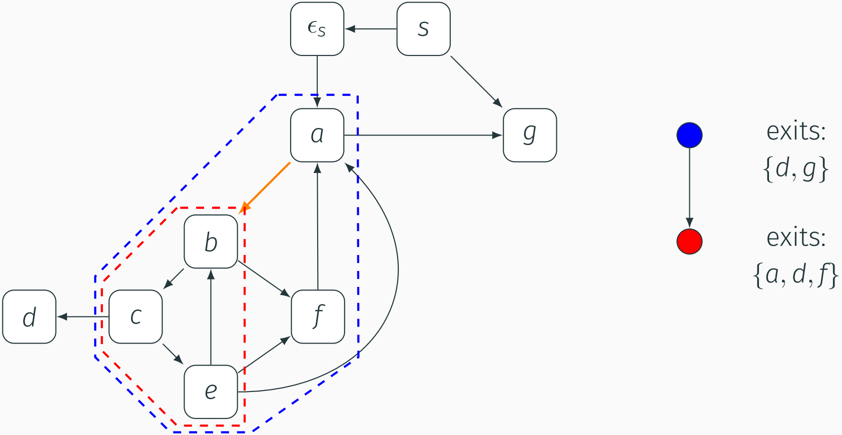


Canonicalization – Dedicate Preheaders



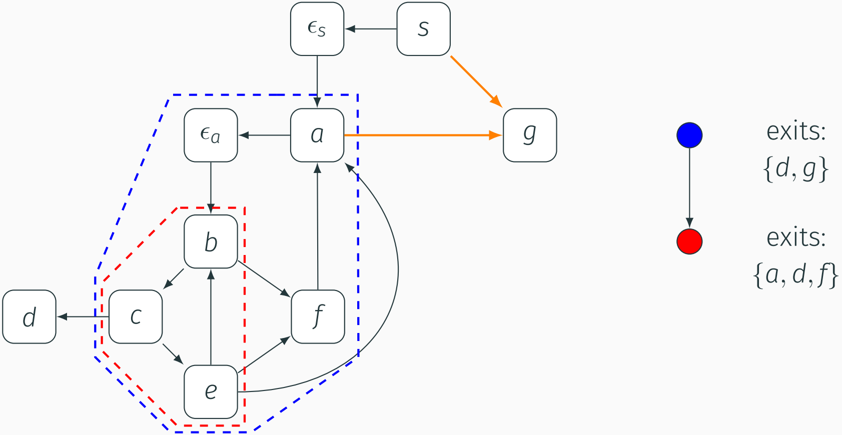
Construct loop nesting forest

Canonicalization – Dedicate Preheaders



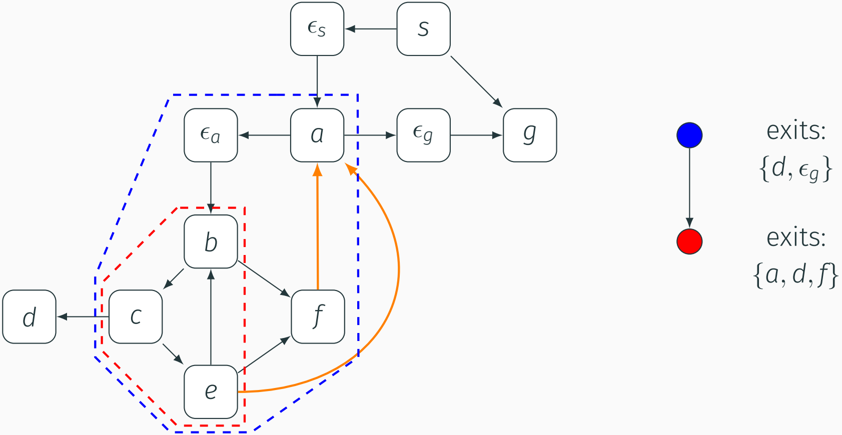
Dedicate preheader of outer (blue) loop

Canonicalization – Dedicate Exits



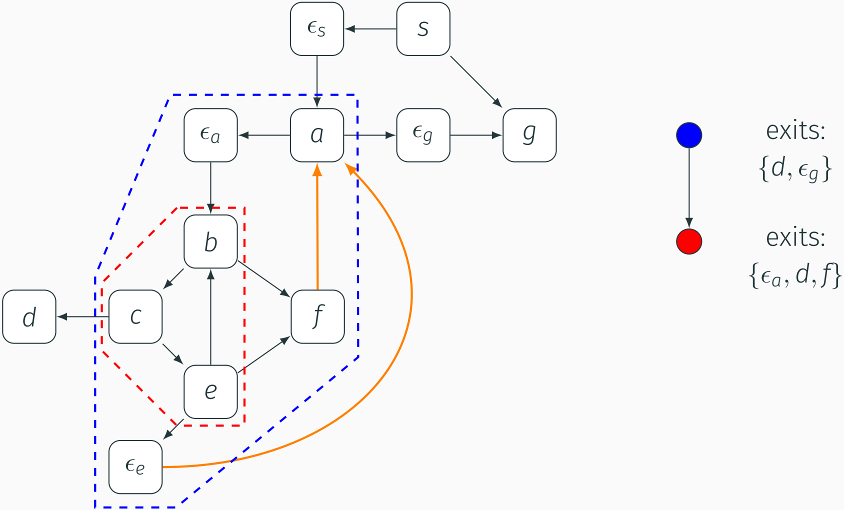
Dedicate preheader of inner (red) loop

Canonicalization – Dedicate Exits



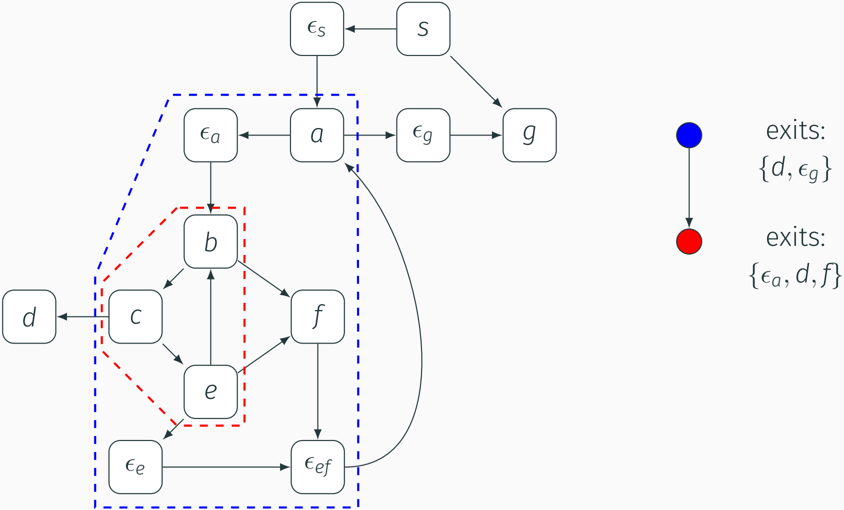
Dedicate exit (block g) of outer (blue) loop

Canonicalization – Ensure Unique Latches



Dedicate exit (block a) of inner (red) loop

Canonicalization – Ensure Unique Latches



Make latch for outer (blue) loop unique

Edge Deletion

Algorithm Outline

(1) Remove edge from graph

Algorithm Outline

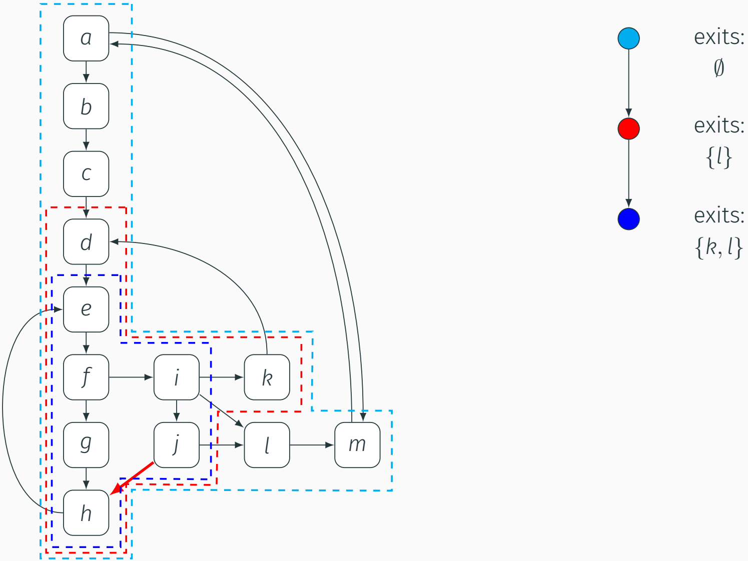
(1) Remove edge from graph
(2) Remove references to unreachable blocks and edges

Algorithm Outline

(1) Remove edge from graph
(2) Remove references to unreachable blocks and edges
(3) Eject extraneous blocks from loop where edge was removed

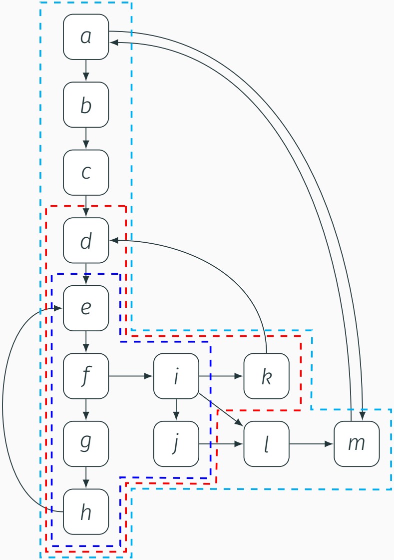
Example #1

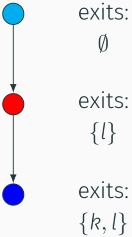
Deleting Edge (j, h) – Incremental Repair



Initial graph

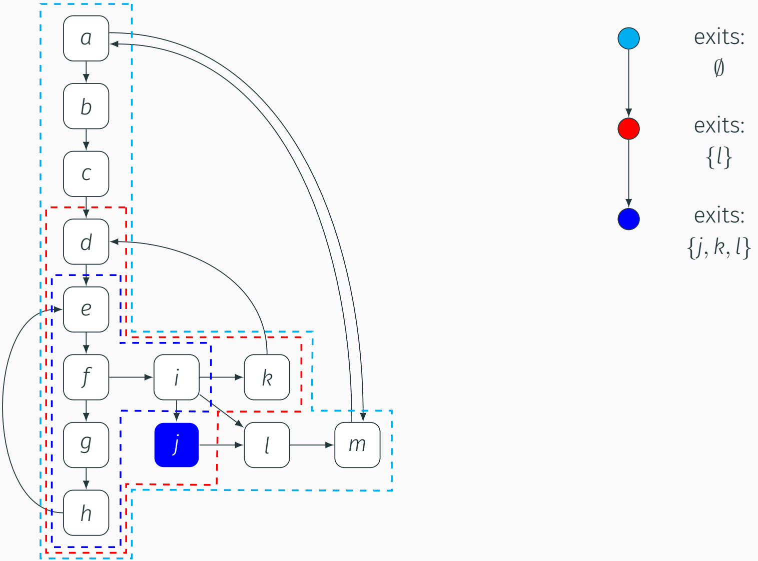
Deleting Edge (j, h) – Incremental Repair





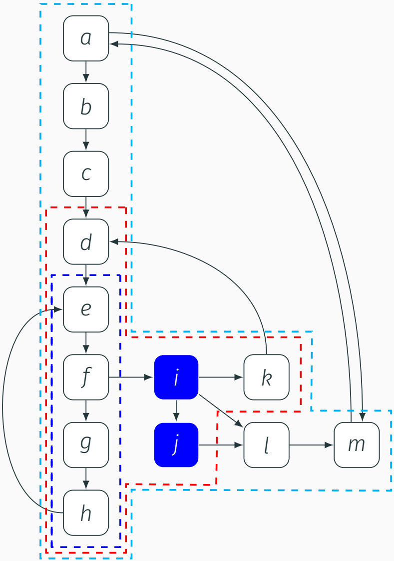
Edge deleted

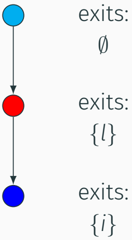
Deleting Edge (j, h) – Incremental Repair



Eject block j from inner (blue) loop

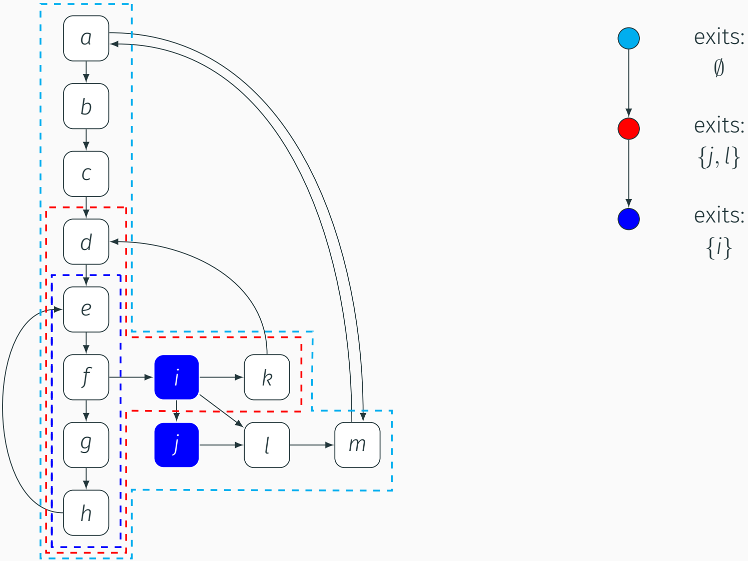
Deleting Edge (j, h) – Incremental Repair





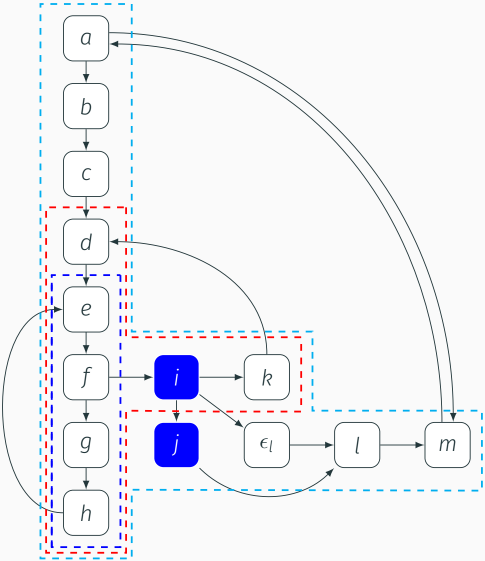
Eject block i from inner (blue) loop

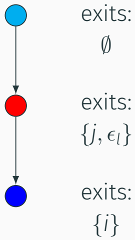
Deleting Edge (j, h) – Incremental Repair



Eject block j from middle (red) loop

Deleting Edge (j, h) – Incremental Repair

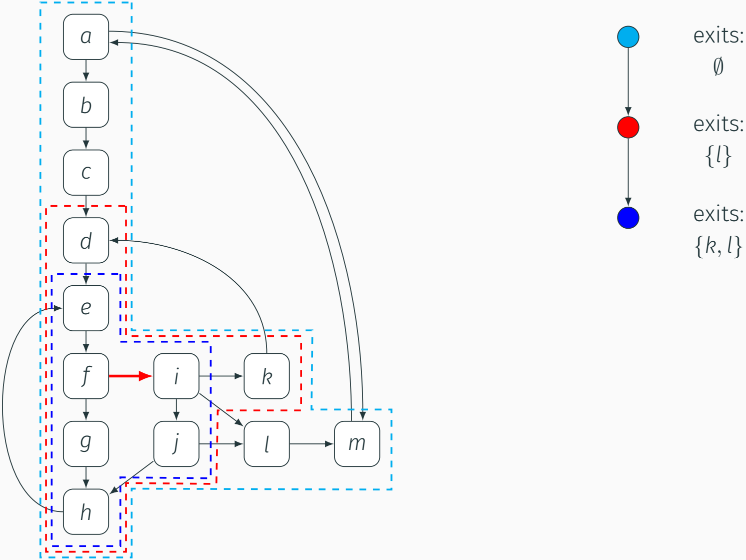




Place block ϵ_l on edge (i, l) to dedicate exit

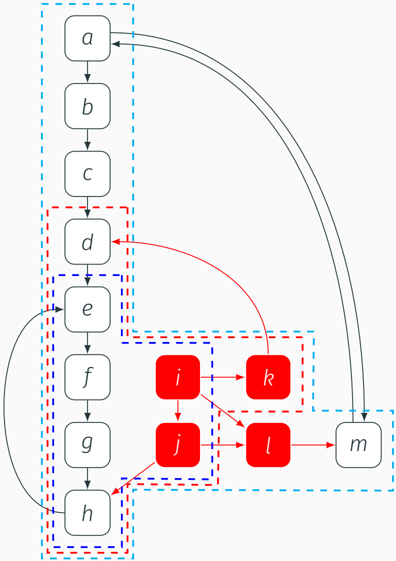
Example #2

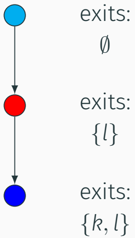
Deleting Edge (f, i) – Incremental Repair



Initial graph

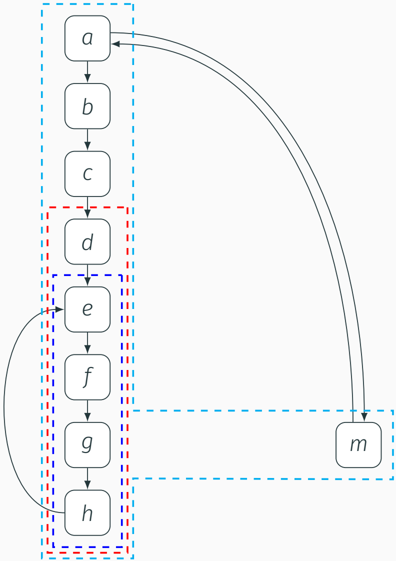
Deleting Edge (f, i) – Incremental Repair

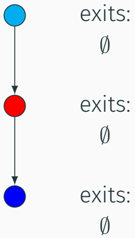




Edge deleted

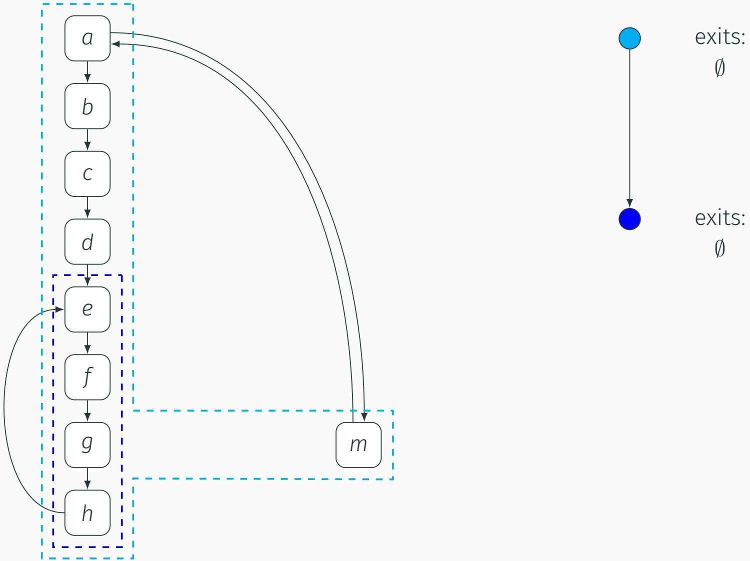
Deleting Edge (f, i) – Incremental Repair





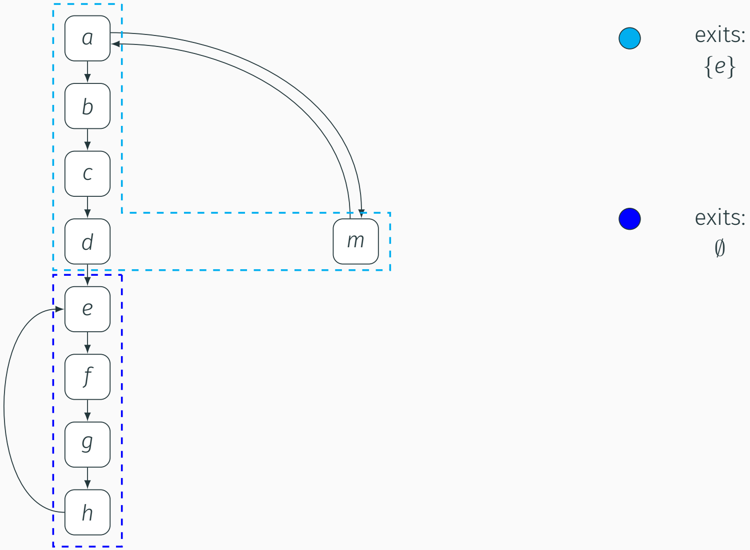
Remove unreachable blocks from graph, loop nesting forest

Deleting Edge (f, i) – Incremental Repair



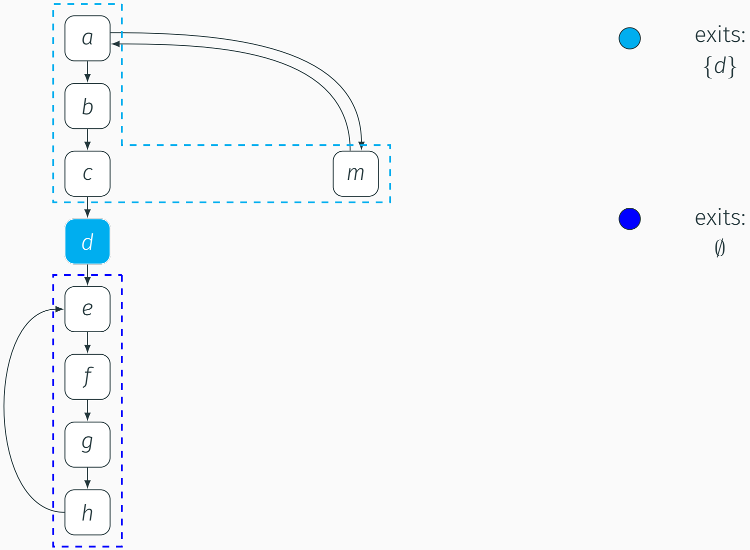
Remove destroyed middle (red) loop

Deleting Edge (f, i) – Incremental Repair



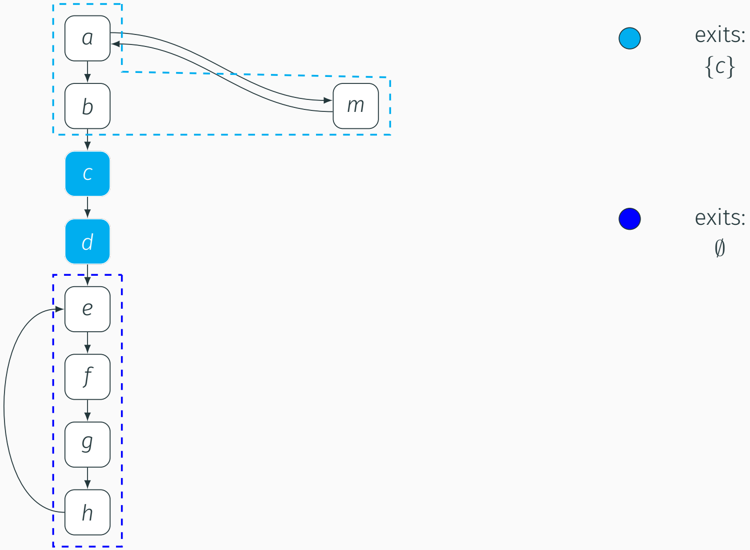
Eject block e (and its loop) from the outer (cyan) loop

Deleting Edge (f, i) – Incremental Repair



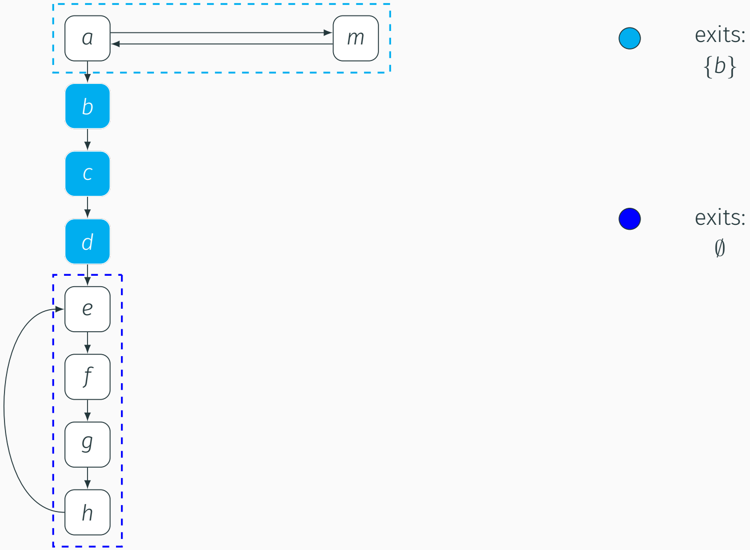
Eject block *d* from outer (cyan) loop

Deleting Edge (f, i) – Incremental Repair



Eject block c from outer (cyan) loop

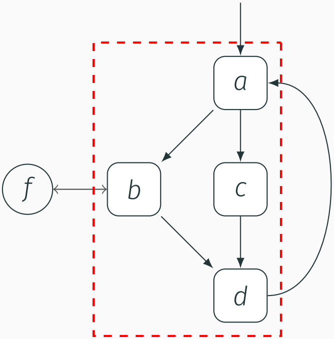
Deleting Edge (f, i) – Incremental Repair



Eject block b from outer (cyan) loop

Additional Applications

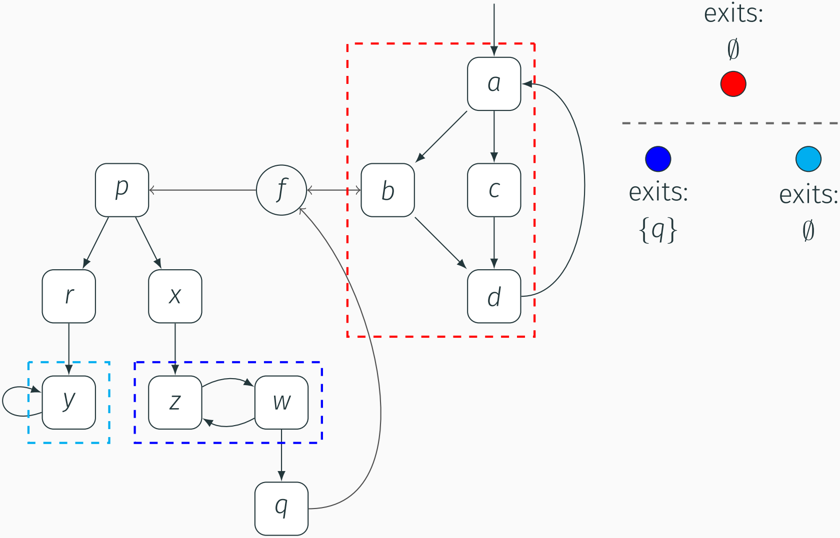
Function Inlining



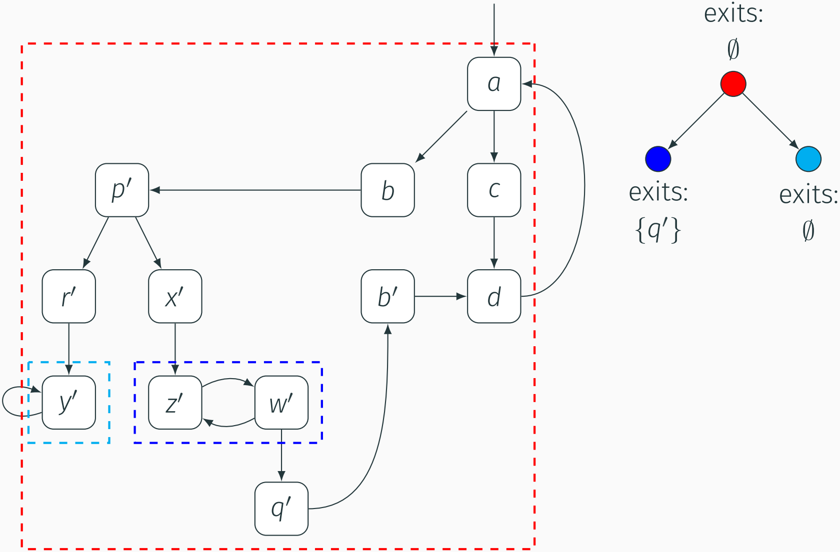
exits:

$\emptyset$

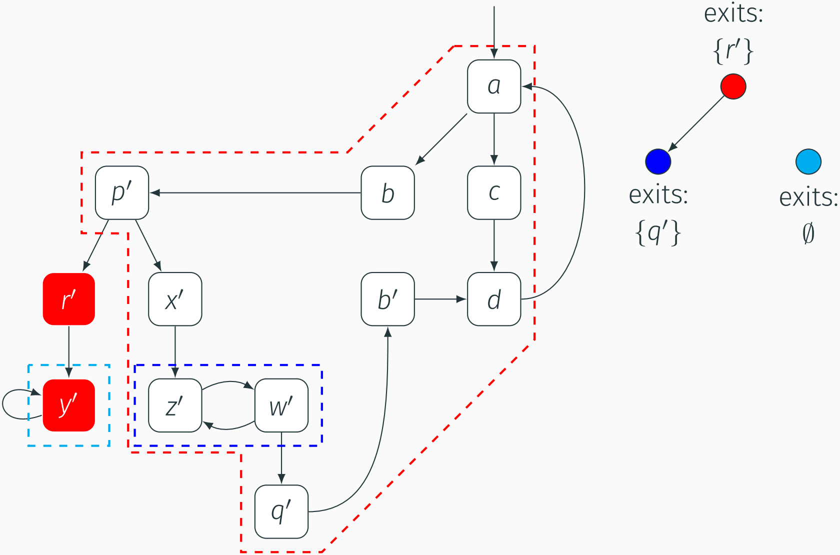
Function Inlining



Function Inlining



Function Inlining



Evaluation

Methodology

(1) Construct Waddle IR from C++ source (through LLVM)
- 6 compilation units
- ~85 interesting functions per compilation unit
- ~21 blocks, ~30 edges, $\leq$ 10 loops ($\leq$ depth of 4) per function

Methodology

(1) Construct Waddle IR from C++ source (through LLVM)
   - 6 compilation units
   - ~85 interesting functions per compilation unit
   - ~21 blocks, ~30 edges, $\leq$ 10 loops ($\leq$ depth of 4) per function

(2) Construct a stable order of edges

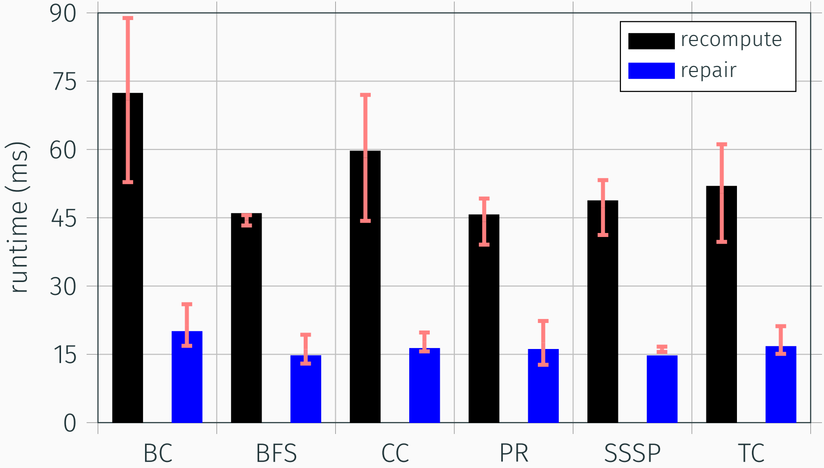
Methodology

(1) Construct Waddle IR from C++ source (through LLVM)
  - 6 compilation units
  - ~85 interesting functions per compilation unit
  - ~21 blocks, ~30 edges, $\leq$ 10 loops ($\leq$ depth of 4) per function

(2) Construct a stable order of edges

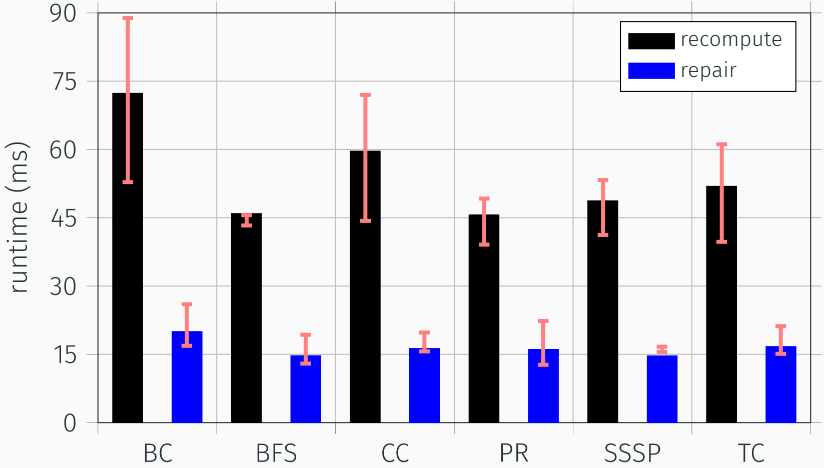
(3) For each edge that has siblings remaining:
  - Delete edge and reconstruct canonical form (baseline)
  - Delete edge using procedure described here

Results



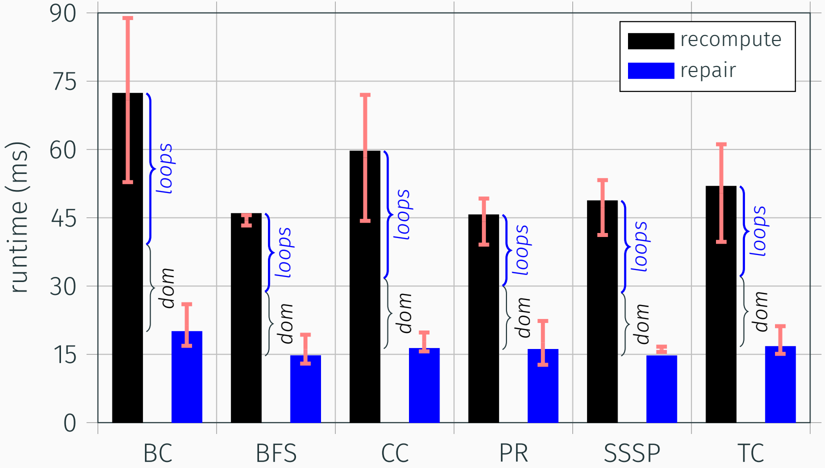
90
75
60
45
30
15
0
runtime (ms)
recompute
repair
BC
BFS
CC
PR
SSSP
TC

Results



64.80 to 72.7% decrease in runtime

Results

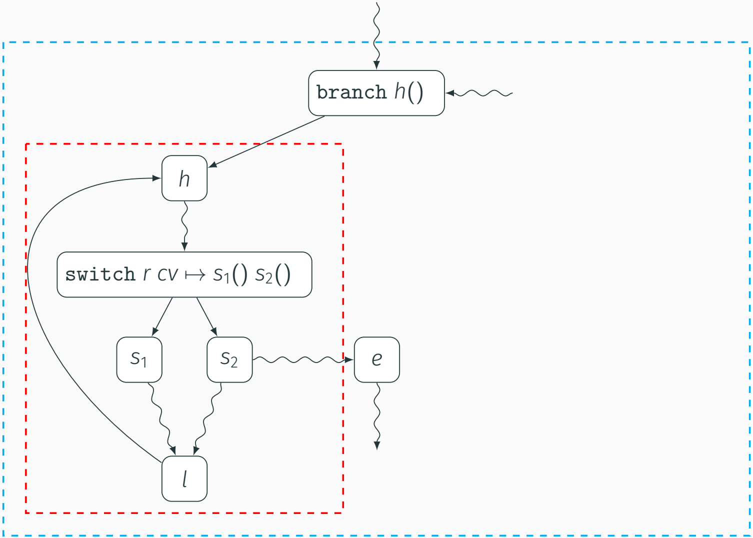


64.80 to 72.7% decrease in runtime

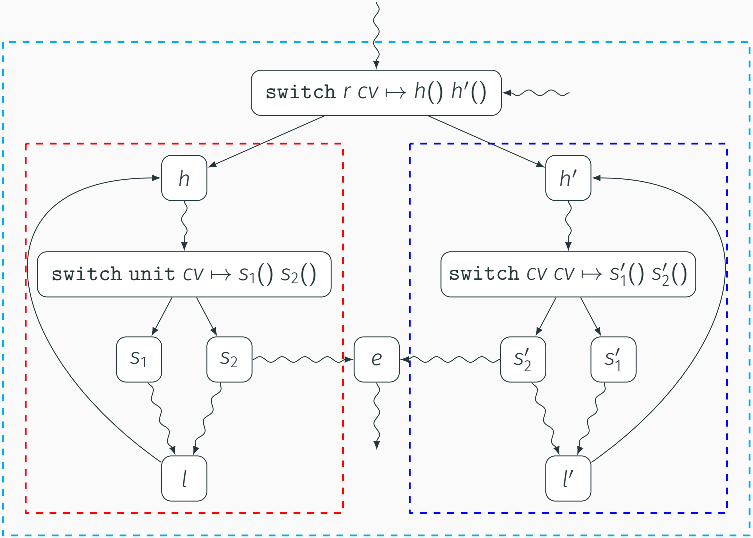
Questions?

Bonus Slides

Subgraph Duplication (Loop Unswitching)



Subgraph Duplication (Loop Unswitching)



Thank You!