C⚙de analysis *and* transf⚙rmation

Loop transformations

Simone Campanoni
simonec@eecs.northwestern.edu

Outline

- Simple loop transformations
- Loop invariants based transformations
- Induction variables based transformations
- Complex loop transformations

Simple loop transformations

Simple loop transformations are used to

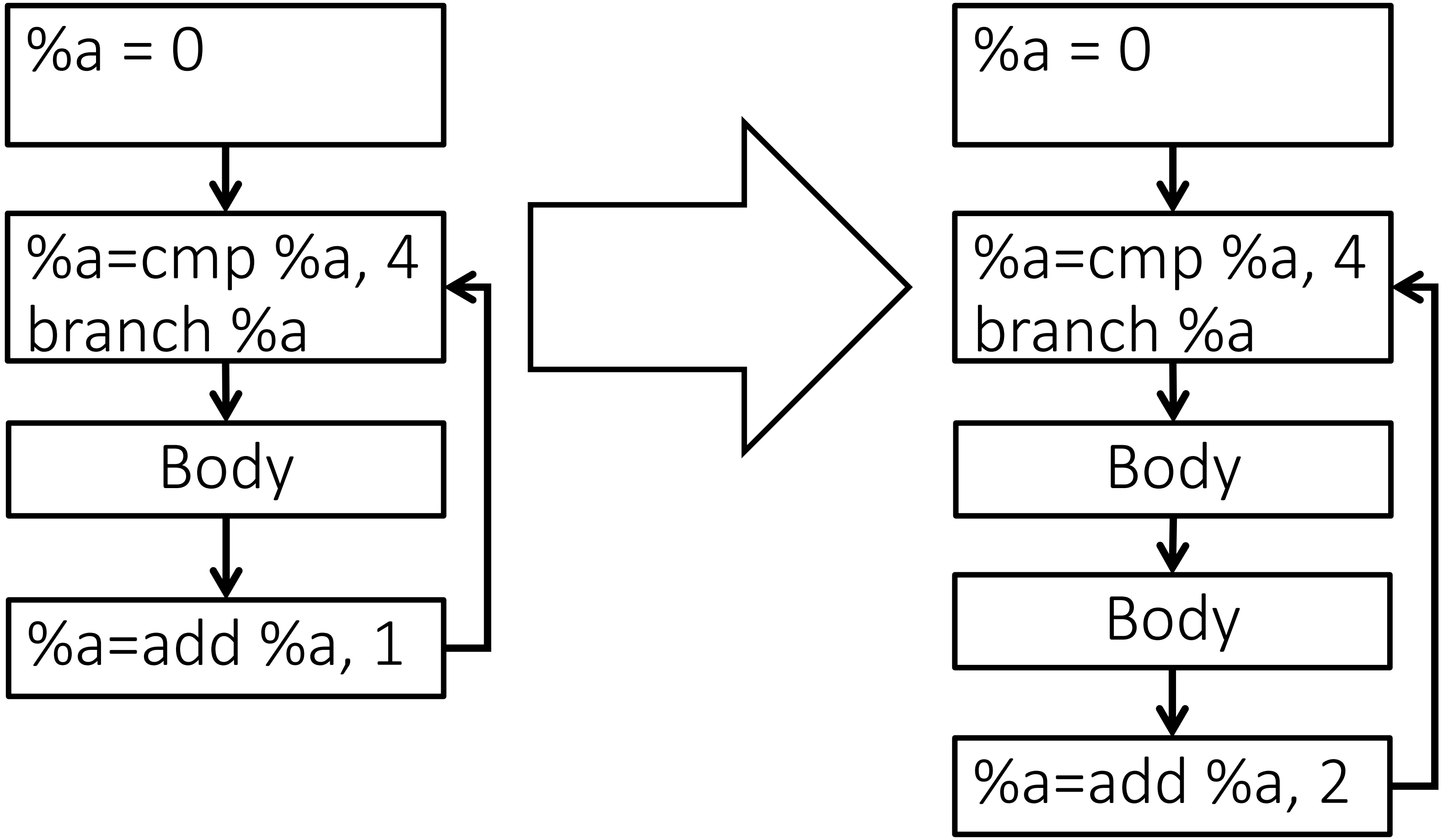
- Increase performance/energy savings

  and/or

- Unblock other transformations
  - E.g., increase the number of constant propagations
  - E.g., Extract thread-level parallelism from sequential code
  - E.g., Generate vector instructions

Loop unrolling

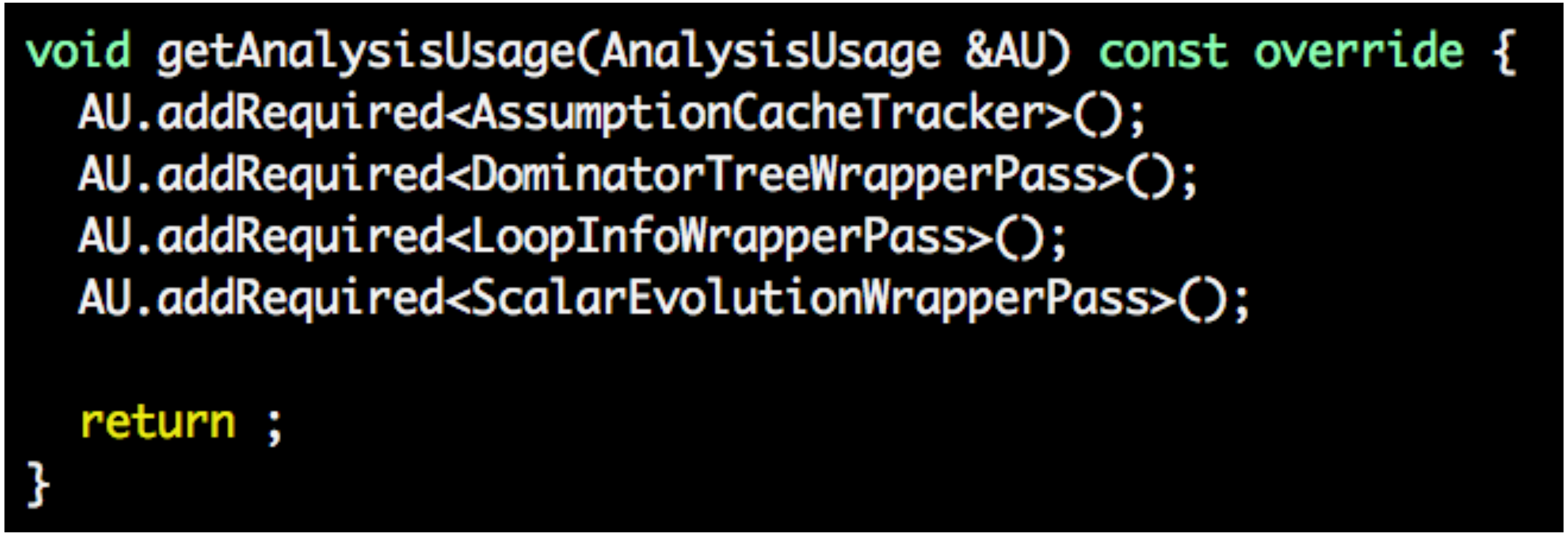
```
for (a=0; a < 4; a++){
   … // Body
}
```



Loop unrolling in LLVM: requirements

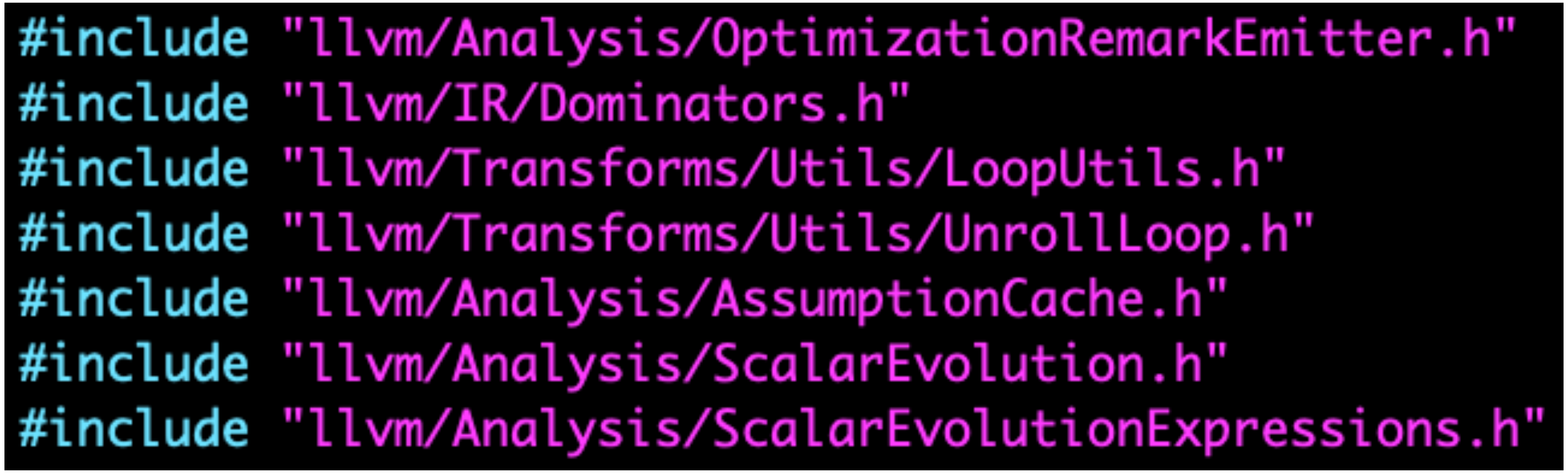
- The loop you want to unroll must be in LCSSA form

Loop unrolling in LLVM: dependences



```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<AssumptionCacheTracker>();
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.addRequired<LoopInfoWrapperPass>();
  AU.addRequired<ScalarEvolutionWrapperPass>();

  return ;
}
```

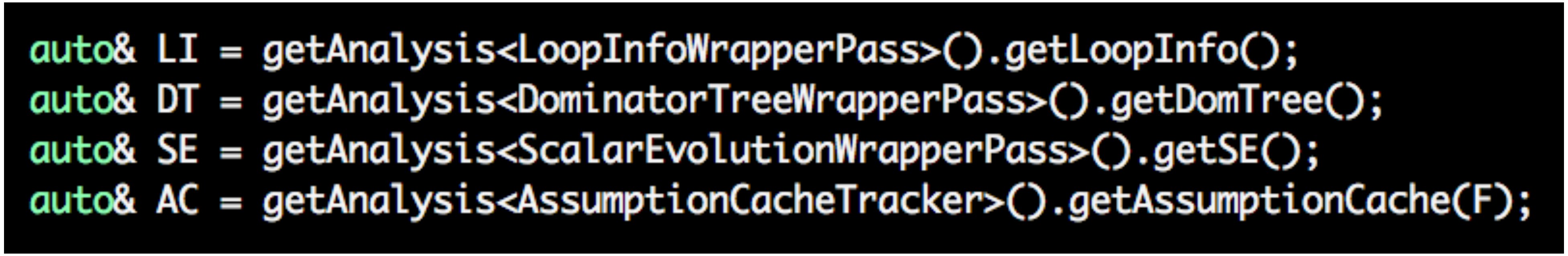
Loop unrolling in LLVM: headers



```
#include "llvm/Analysis/OptimizationRemarkEmitter.h"
#include "llvm/IR/Dominators.h"
#include "llvm/Transforms/Utils/LoopUtils.h"
#include "llvm/Transforms/Utils/UnrollLoop.h"
#include "llvm/Analysis/AssumptionCache.h"
#include "llvm/Analysis/ScalarEvolution.h"
#include "llvm/Analysis/ScalarEvolutionExpressions.h"
```

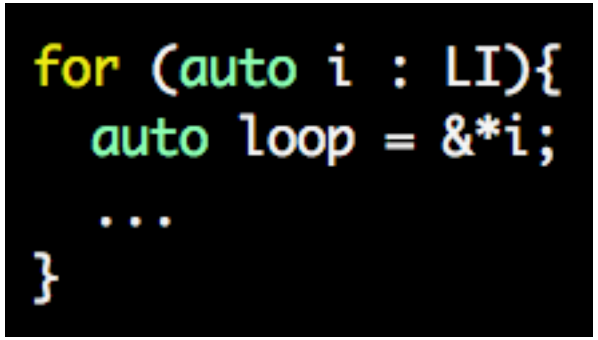
Loop unrolling in LLVM

Get the results of the required analyses

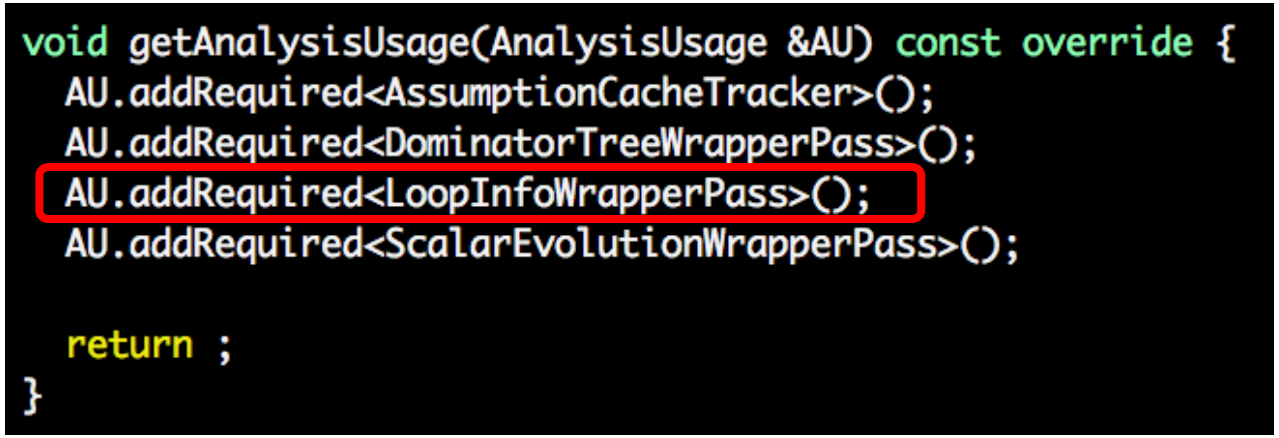


```
auto& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>().getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
```

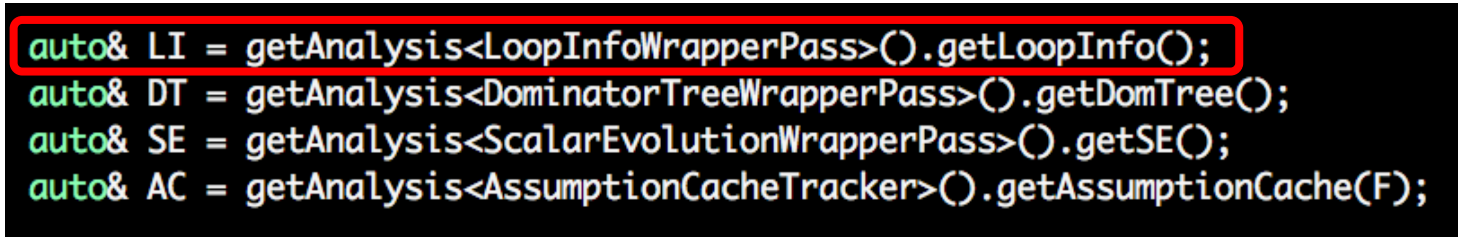
Fetch a loop



```
for (auto i : LI){
   auto loop = &*i;
   ...
}
```



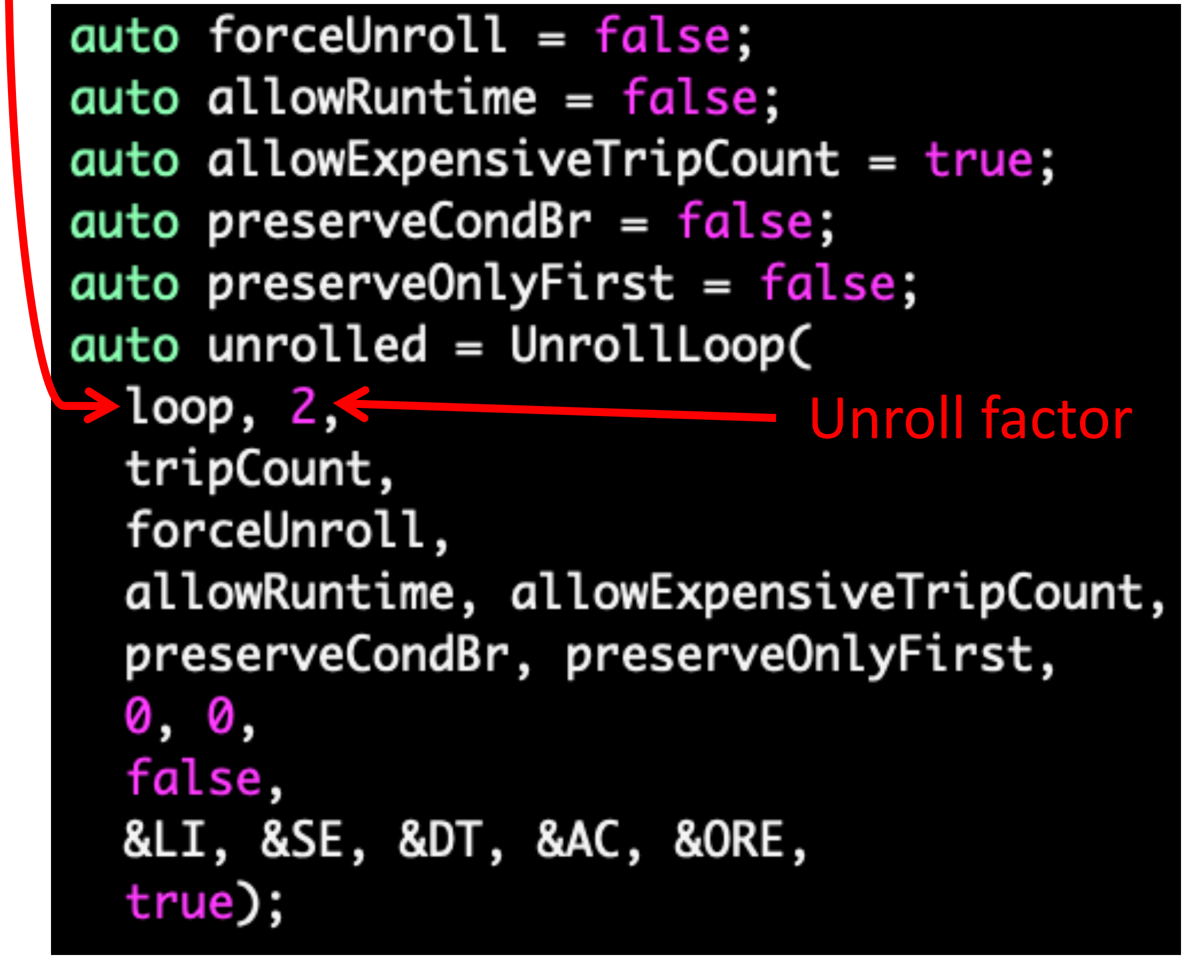
```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<AssumptionCacheTracker>();
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.addRequired<LoopInfoWrapperPass>();
  AU.addRequired<ScalarEvolutionWrapperPass>();

  return ;
}
```



```
auto& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>().getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
```

Loop unrolling in LLVM: API

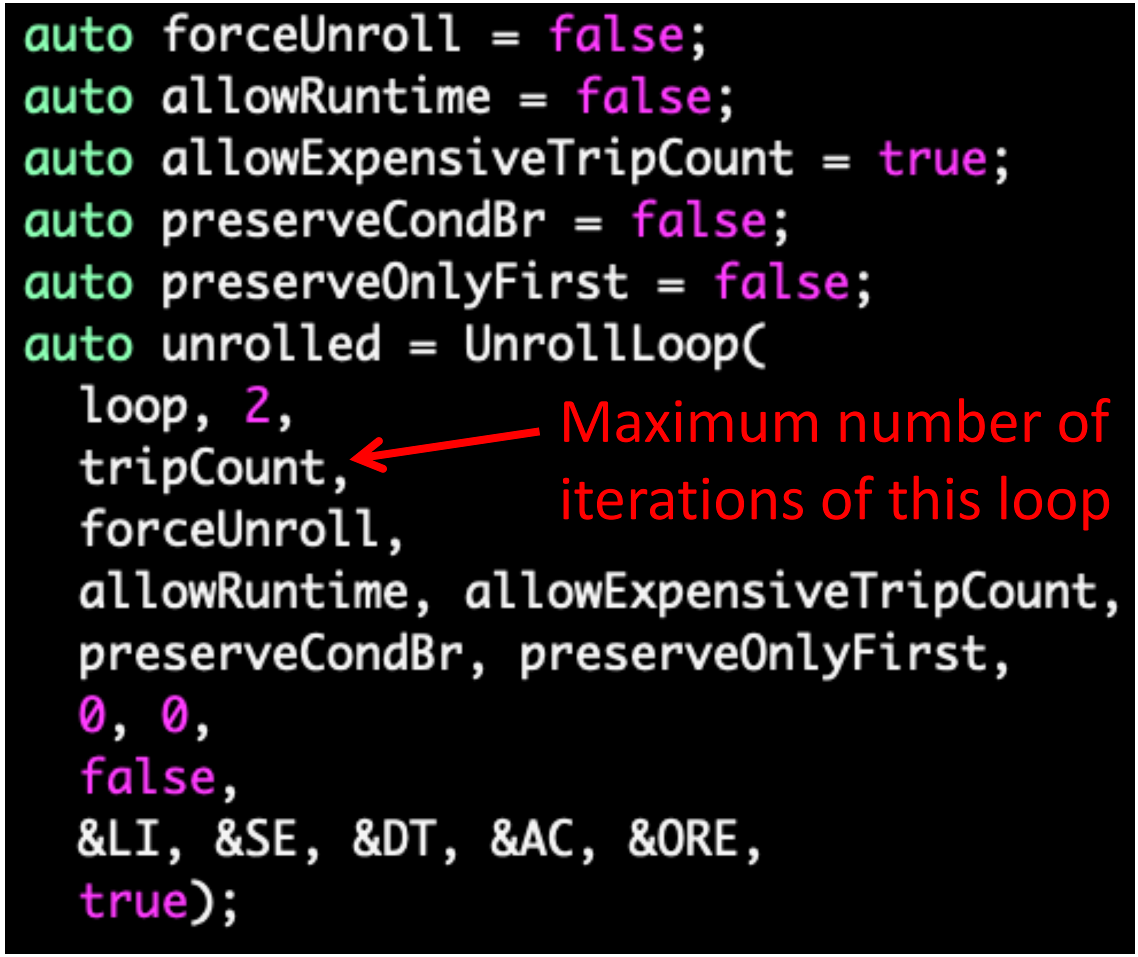
Loop to unroll



```
auto forceUnroll = false;
auto allowRuntime = false;
auto allowExpensiveTripCount = true;
auto preserveCondBr = false;
auto preserveOnlyFirst = false;
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  0, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```

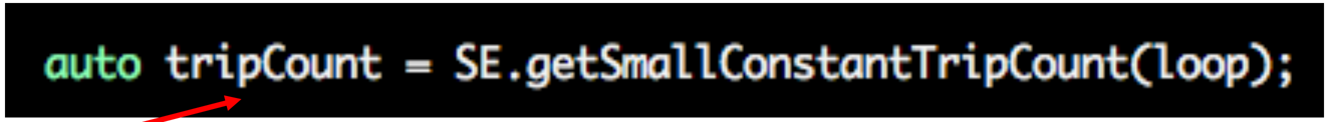
Unroll factor

Loop unrolling in LLVM: API



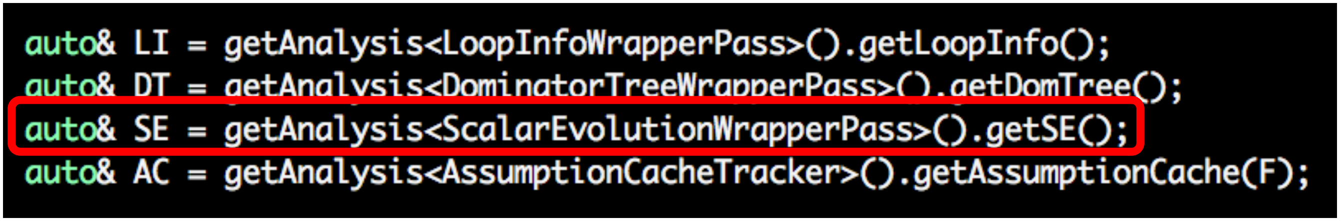
```
auto forceUnroll = false;
auto allowRuntime = false;
auto allowExpensiveTripCount = true;
auto preserveCondBr = false;
auto preserveOnlyFirst = false;
auto unrolled = UnrollLoop(
   loop, 2,
   tripCount,
   forceUnroll,
   allowRuntime, allowExpensiveTripCount,
   preserveCondBr, preserveOnlyFirst,
   0, 0,
   false,
   &LI, &SE, &DT, &AC, &ORE,
   true);
```

Maximum number of iterations of this loop

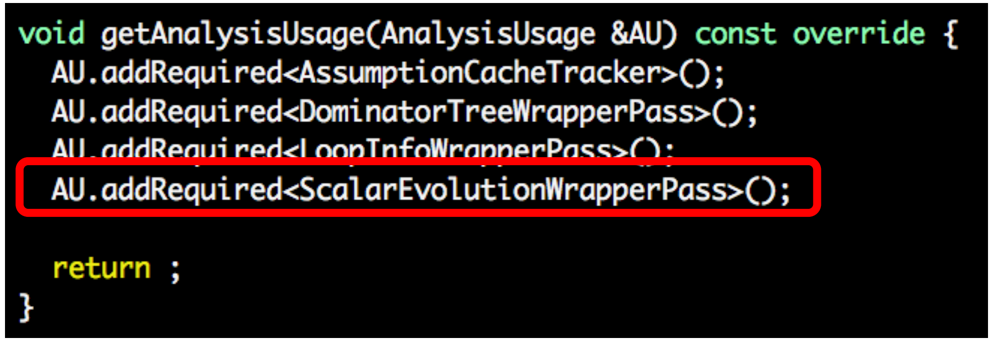


```
auto tripCount = SE.getSmallConstantTripCount(loop);
```

It is 0, or the number of iterations known by SCE

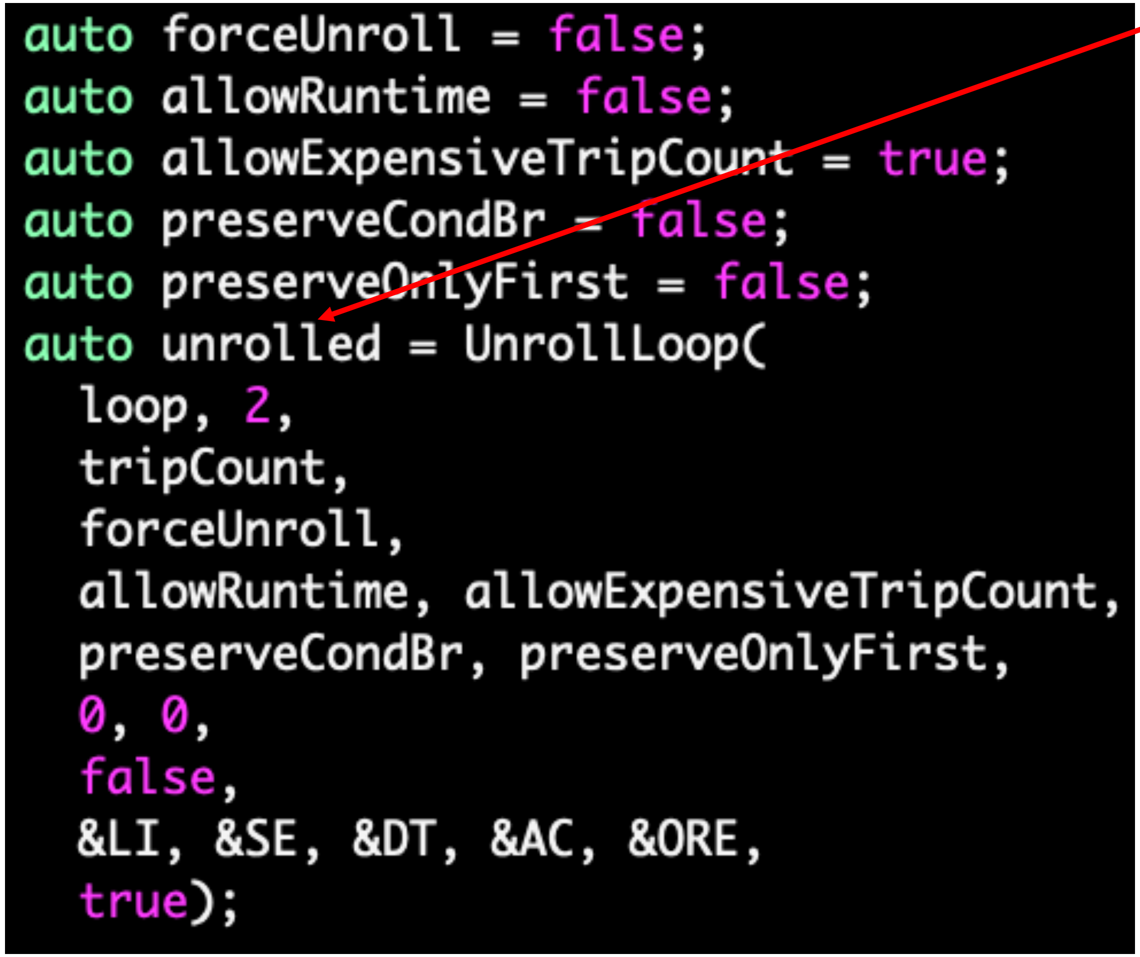


```
auto& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>().getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
```

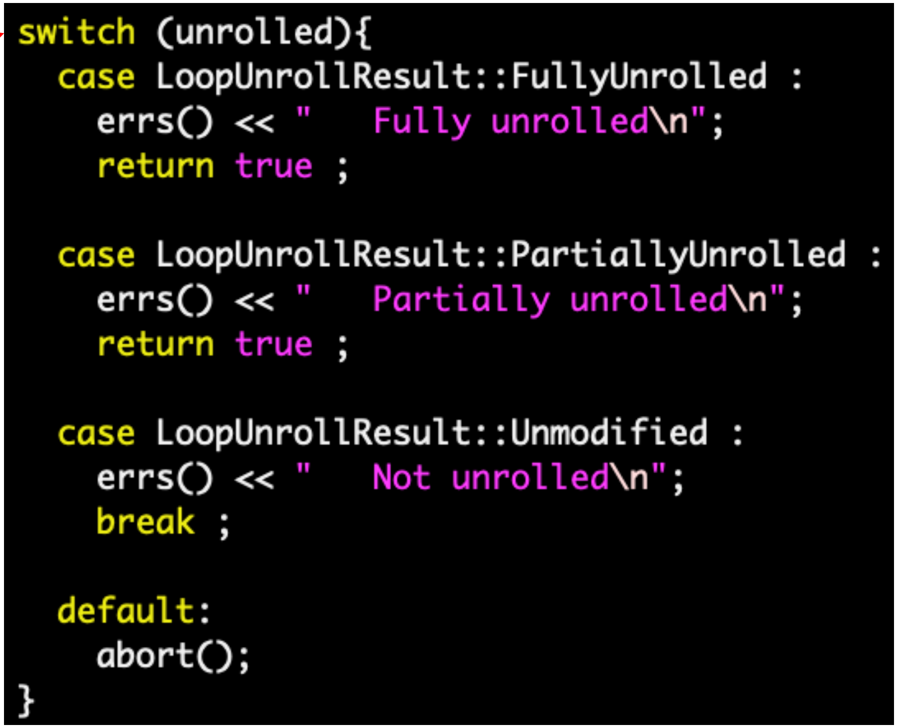


```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired<AssumptionCacheTracker>();
  AU.addRequired<DominatorTreeWrapperPass>();
  AU.addRequired<LoopInfoWrapperPass>();
  AU.addRequired<ScalarEvolutionWrapperPass>();

  return ;
}
```

Loop unrolling in LLVM: result

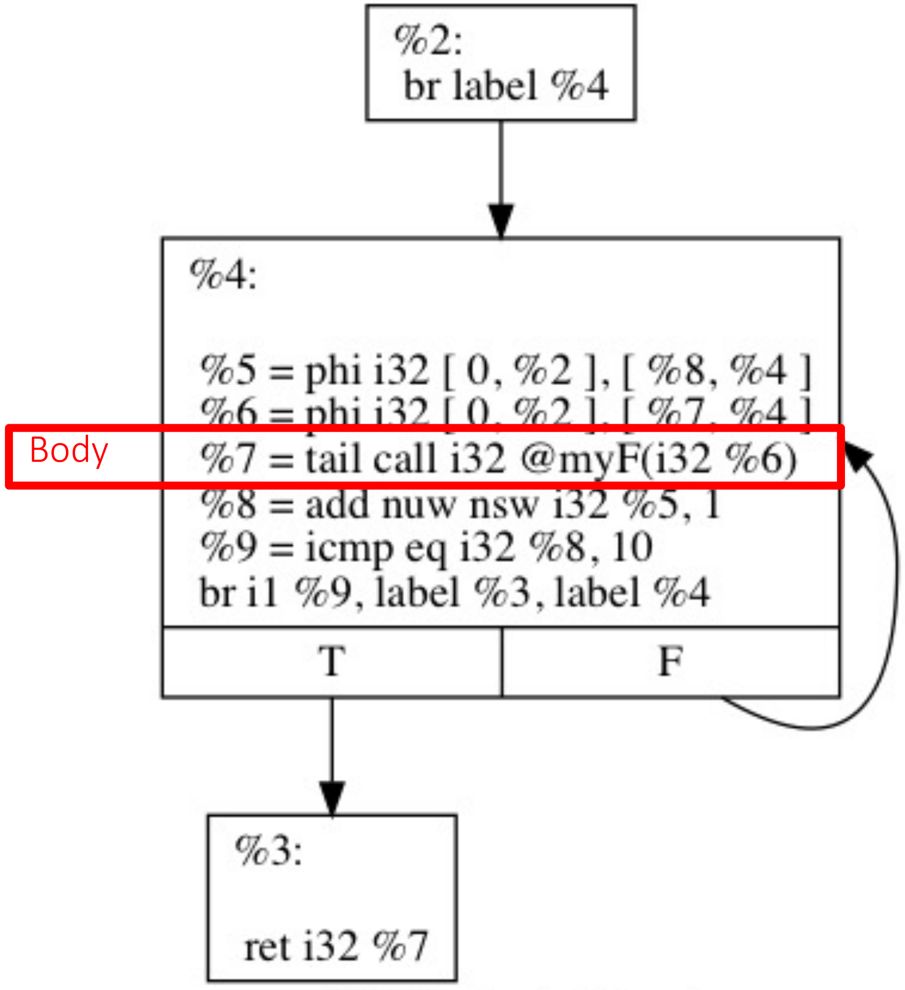


```
auto forceUnroll = false;
auto allowRuntime = false;
auto allowExpensiveTripCount = true;
auto preserveCondBr = false;
auto preserveOnlyFirst = false;
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  0, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```



```
switch (unrolled){
  case LoopUnrollResult::FullyUnrolled :
    errs() << "   Fully unrolled\n";
    return true ;

  case LoopUnrollResult::PartiallyUnrolled :
    errs() << "   Partially unrolled\n";
    return true ;

  case LoopUnrollResult::Unmodified :
    errs() << "   Not unrolled\n";
    break ;

  default:
    abort();
}
```

Loop unrolling in LLVM: example

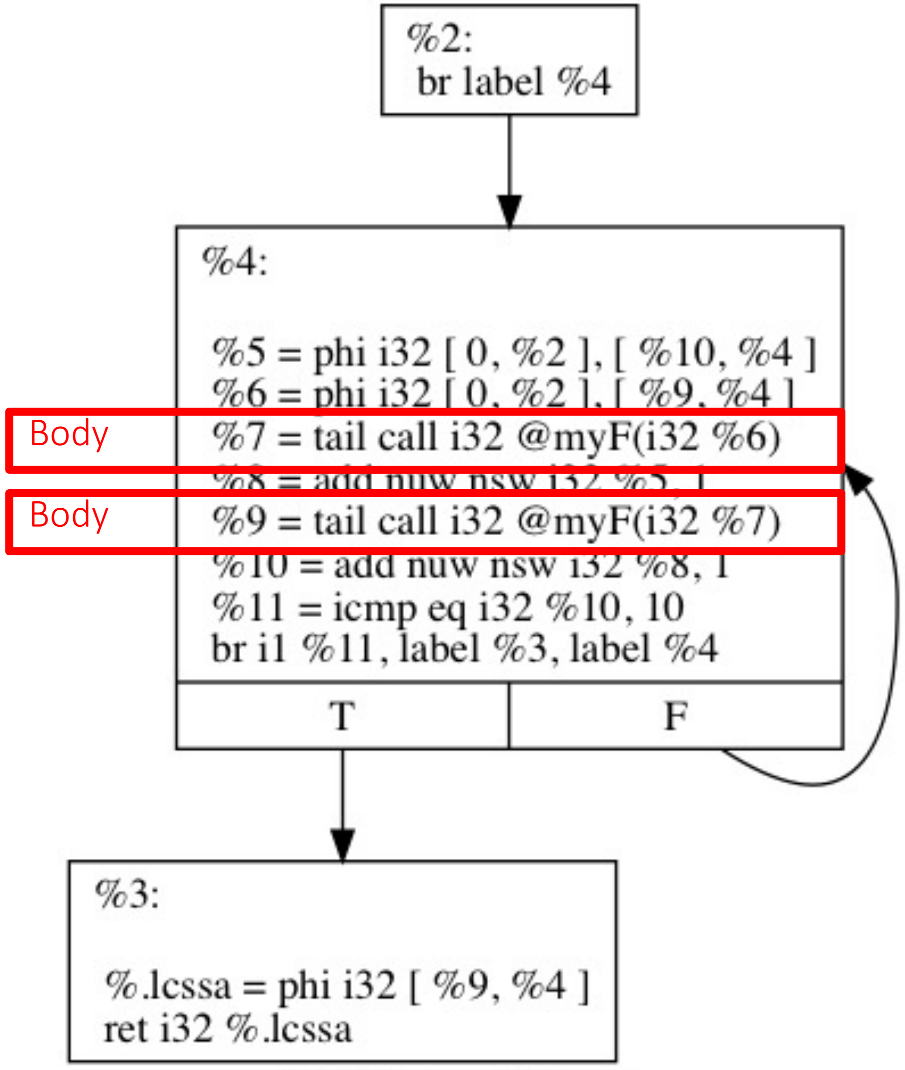


```
%2:
 br label %4
```

```
%4:

 %5 = phi i32 [ 0, %2 ], [ %8, %4 ]
 %6 = phi i32 [ 0, %2 ], [ %7, %4 ]
 %7 = tail call i32 @myF(i32 %6)
 %8 = add nuw nsw i32 %5, 1
 %9 = icmp eq i32 %8, 10
 br i1 %9, label %3, label %4
```

```
%3:

 ret i32 %7
```

CFG for 'main' function

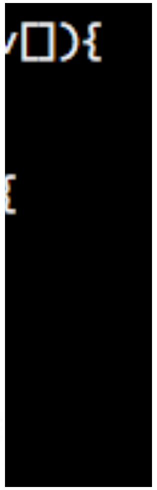


```
%2:
 br label %4
```

```
%4:

 %5 = phi i32 [ 0, %2 ], [ %10, %4 ]
 %6 = phi i32 [ 0, %2 ], [ %9, %4 ]
 %7 = tail call i32 @myF(i32 %6)
 %8 = add nuw nsw i32 %5, 1
 %9 = tail call i32 @myF(i32 %7)
 %10 = add nuw nsw i32 %8, 1
 %11 = icmp eq i32 %10, 10
 br i1 %11, label %3, label %4
```

```
%3:

 %.lcssa = phi i32 [ %9, %4 ]
 ret i32 %.lcssa
```

CFG for 'main' function

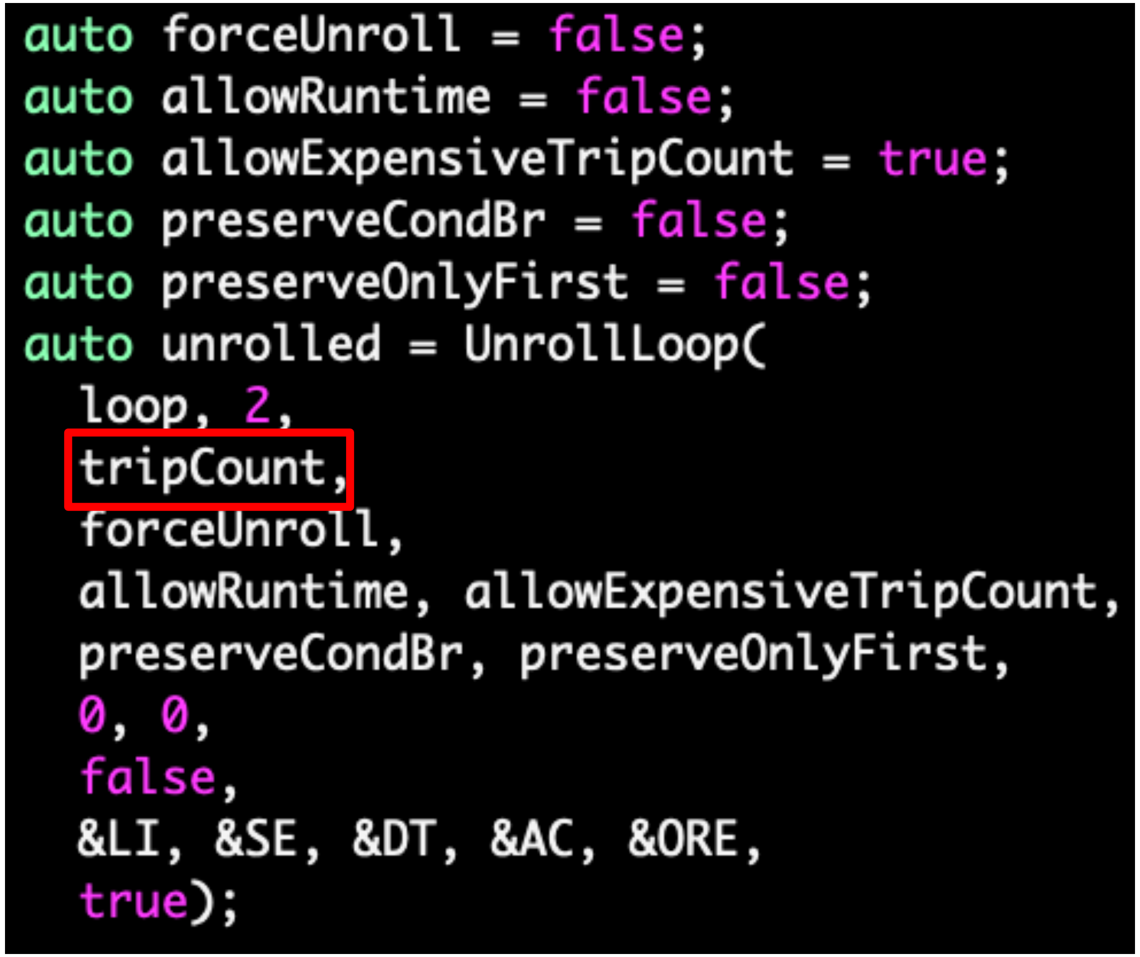


```
/□){

{
```

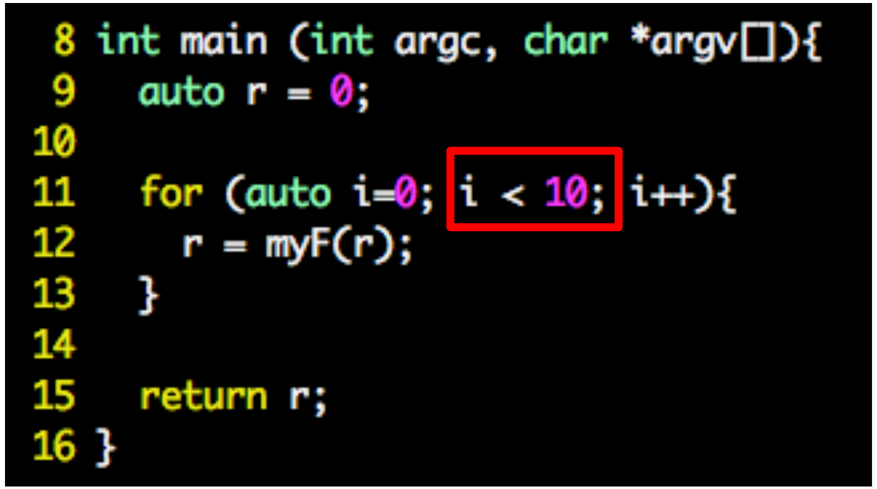
Loop unrolling in LLVM: Demo

- Detail: LLVM_loops/README
- Pass: LLVM_loops/llvm/7
- C program: LLVM_loops/code/12
- C program: LLVM_loops/code/0

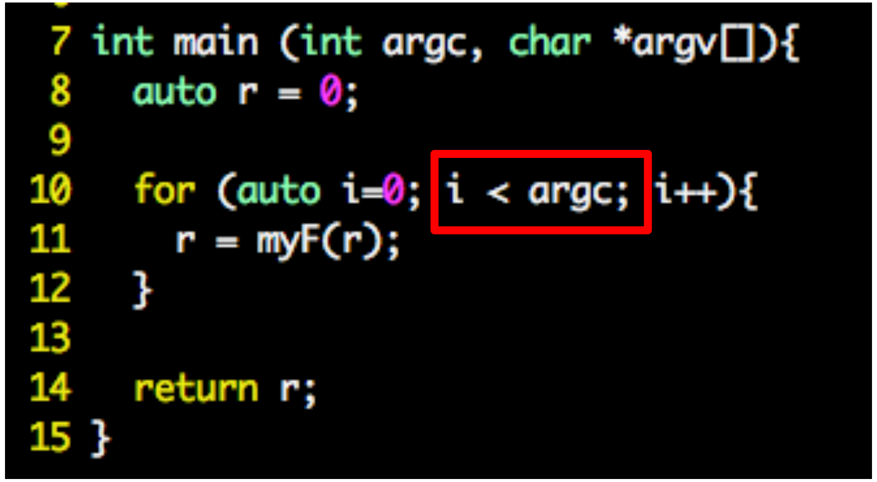
Loop unrolling: the trip count



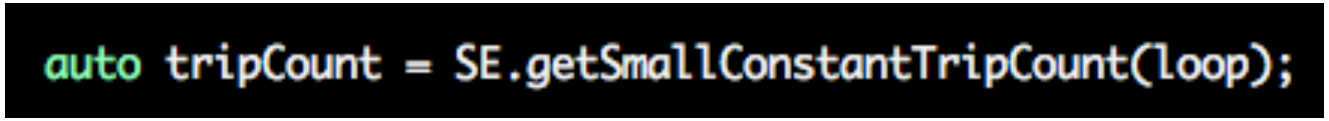
```
auto forceUnroll = false;
auto allowRuntime = false;
auto allowExpensiveTripCount = true;
auto preserveCondBr = false;
auto preserveOnlyFirst = false;
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  0, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```



```
int main (int argc, char *argv[]){
  auto r = 0;

  for (auto i=0; i < 10; i++){
    r = myF(r);
  }

  return r;
}
```

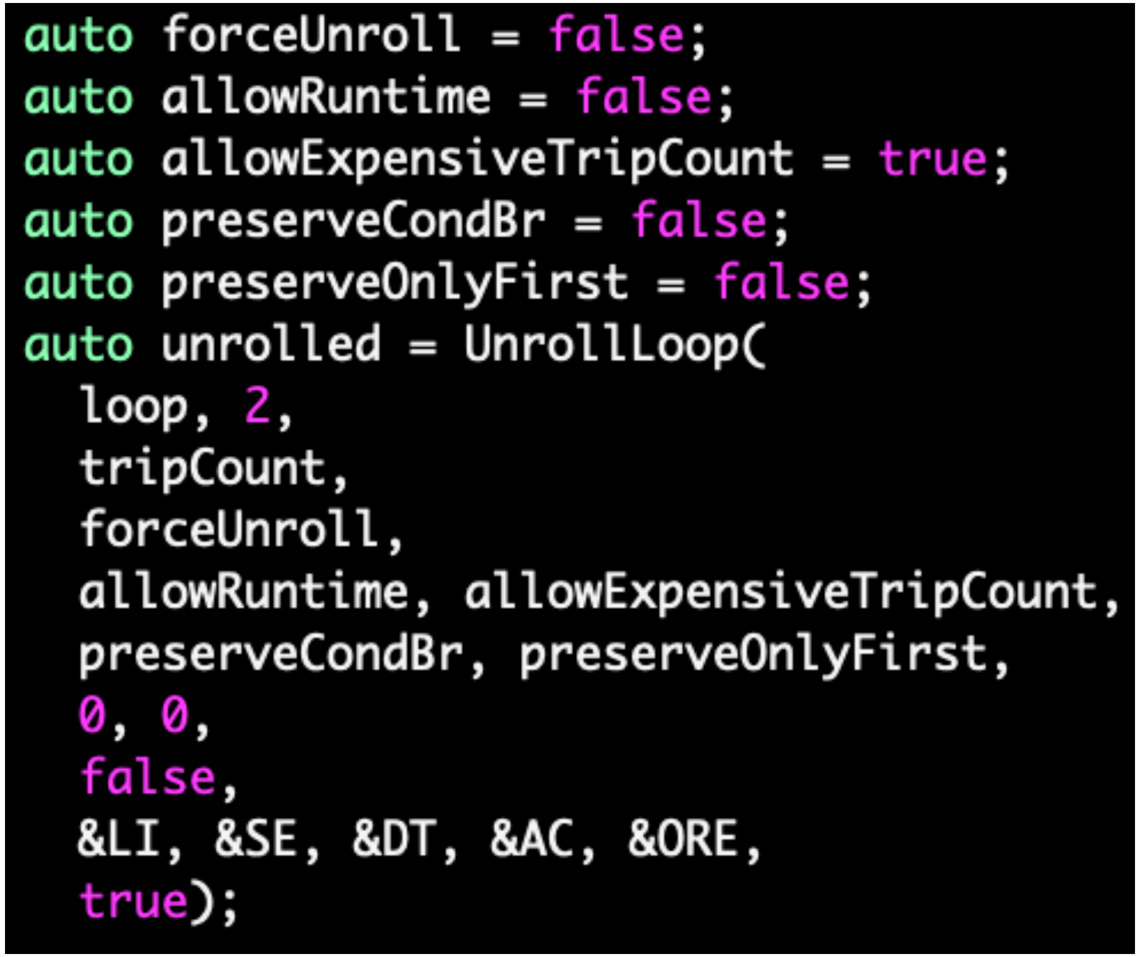


```
int main (int argc, char *argv[]){
  auto r = 0;

  for (auto i=0; i < argc; i++){
    r = myF(r);
  }

  return r;
}
```

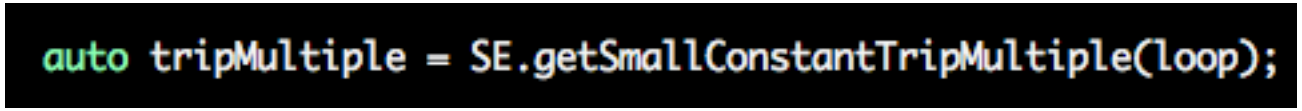


```
auto tripCount = SE.getSmallConstantTripCount(loop);
```

Loop unrolling: the trip multiple

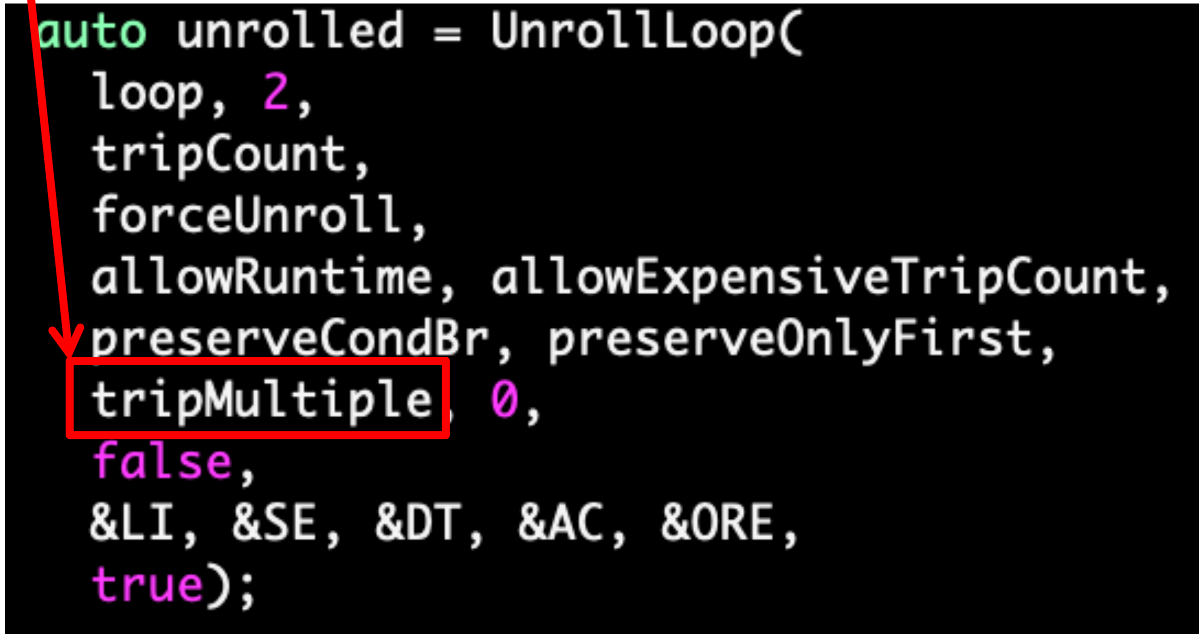


```
auto forceUnroll = false;
auto allowRuntime = false;
auto allowExpensiveTripCount = true;
auto preserveCondBr = false;
auto preserveOnlyFirst = false;
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  0, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```



```
auto tripMultiple = SE.getSmallConstantTripMultiple(loop);
```

Largest constant divisor of the trip count

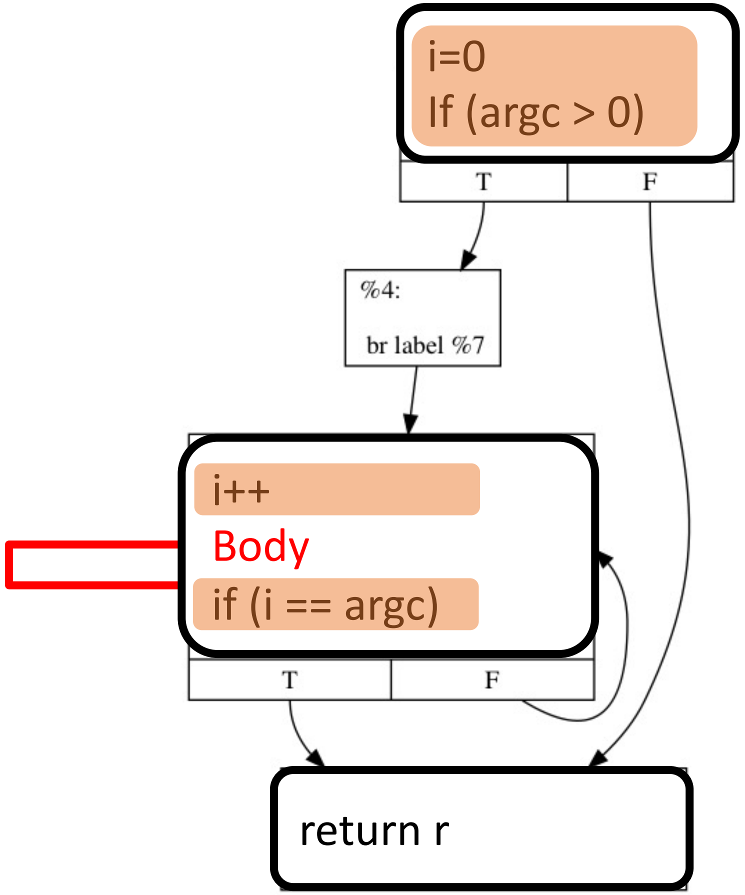


```
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  tripMultiple, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```

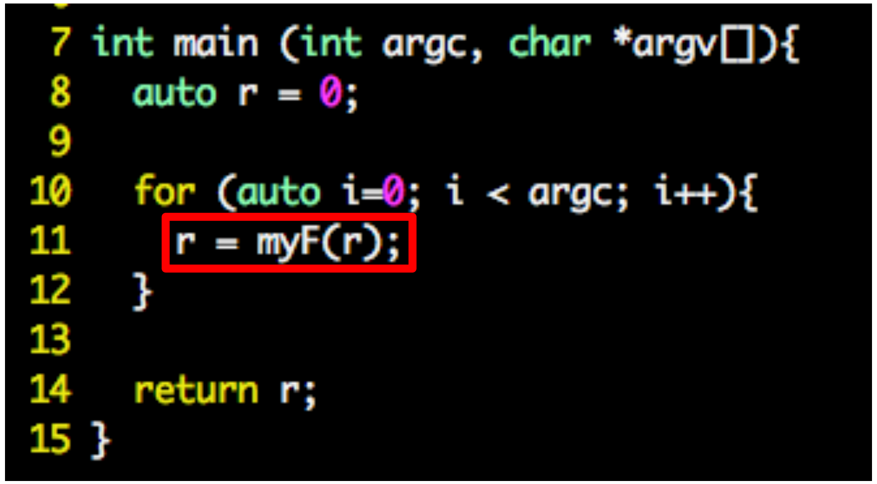
Loop unrolling in LLVM: Demo 2

- Detail: LLVM_loops/README
- Pass: LLVM_loops/llvm/8
- C program: LLVM_loops/code/0

Loop unrolling in LLVM: example 2

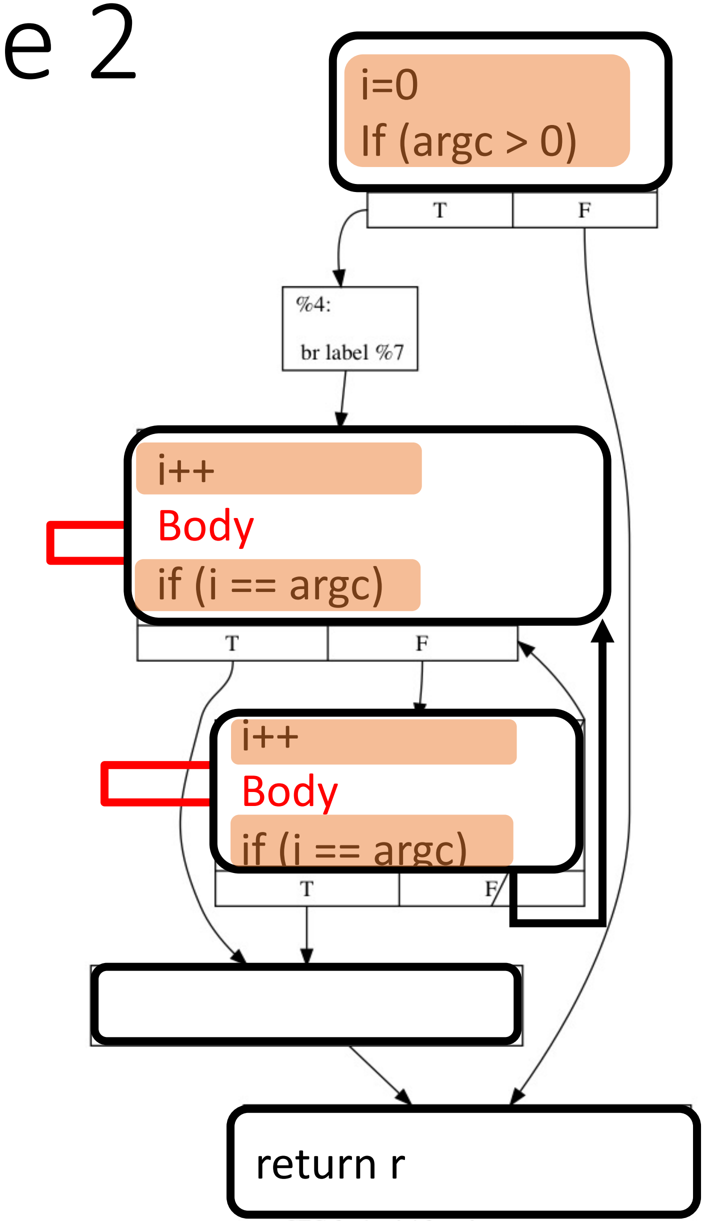


CFG for 'main' function



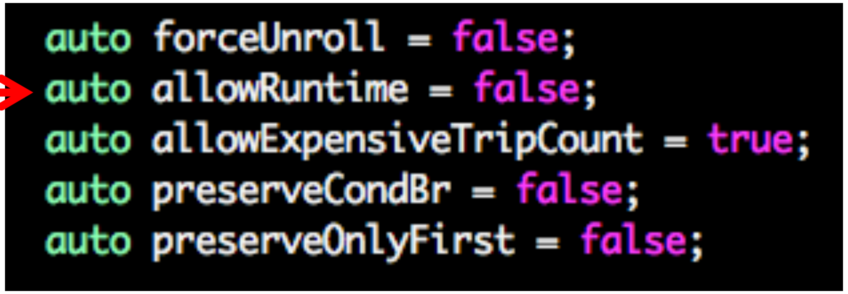
```
int main (int argc, char *argv[]){
  auto r = 0;

  for (auto i=0; i < argc; i++){
    r = myF(r);
  }

  return r;
}
```

There is still the same amount of loop overhead!

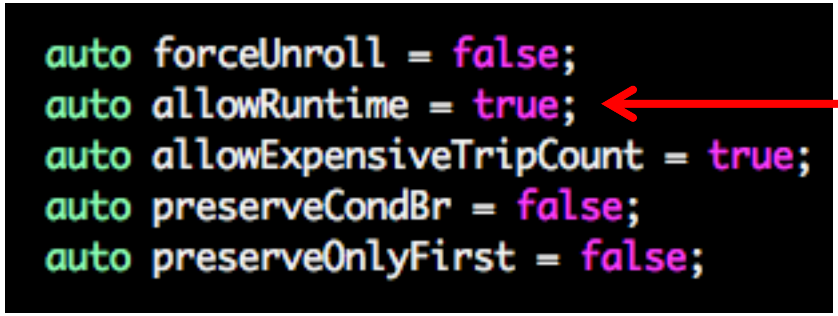


CFG for 'main' function

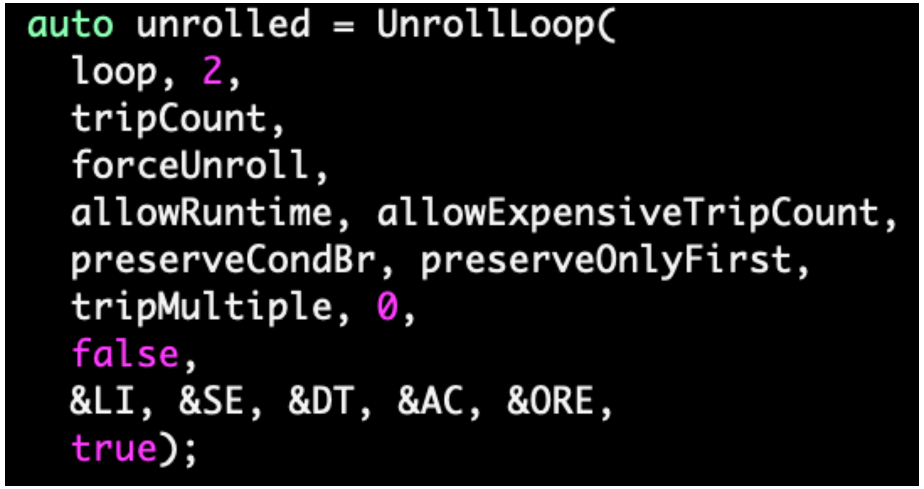
Loop unrolling in LLVM: the runtime checks



```
auto forceUnroll = false;
auto allowRuntime = false;
auto allowExpensiveTripCount = true;
auto preserveCondBr = false;
auto preserveOnlyFirst = false;
```

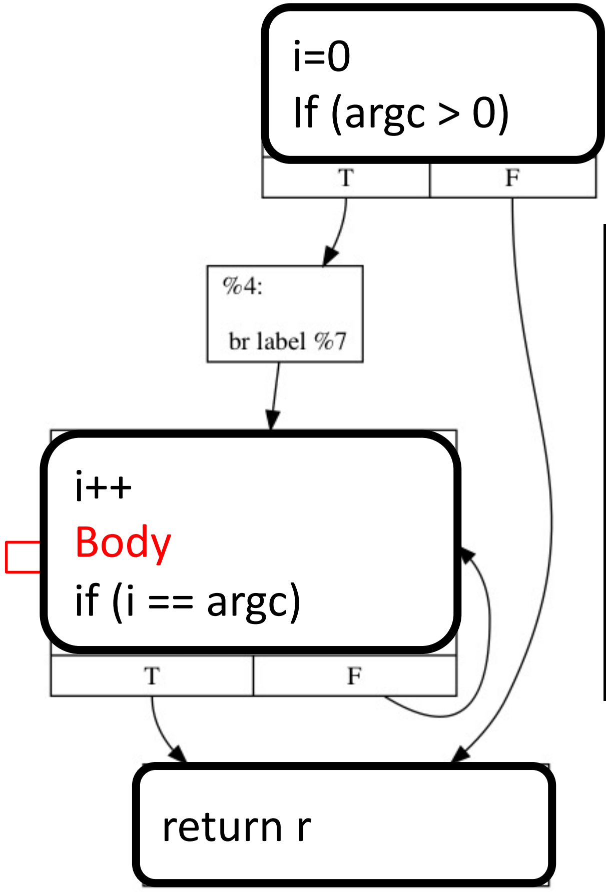


```
auto forceUnroll = false;
auto allowRuntime = true;
auto allowExpensiveTripCount = true;
auto preserveCondBr = false;
auto preserveOnlyFirst = false;
```

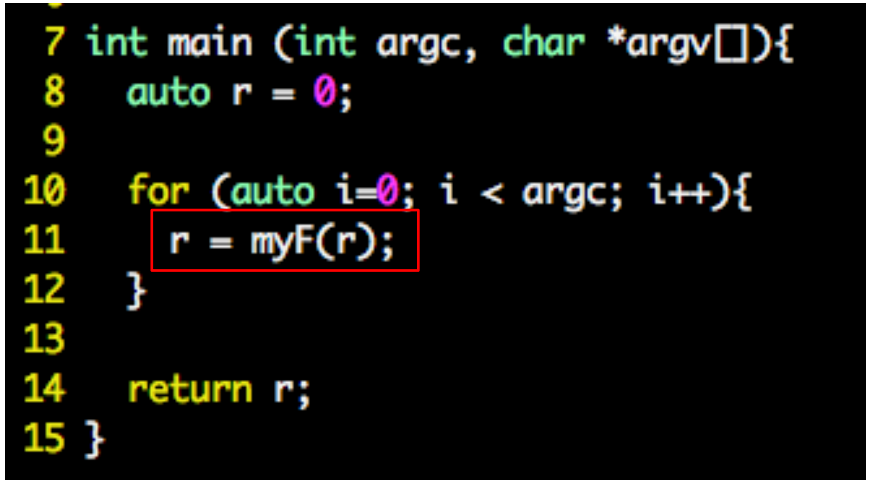


```
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  tripMultiple, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```

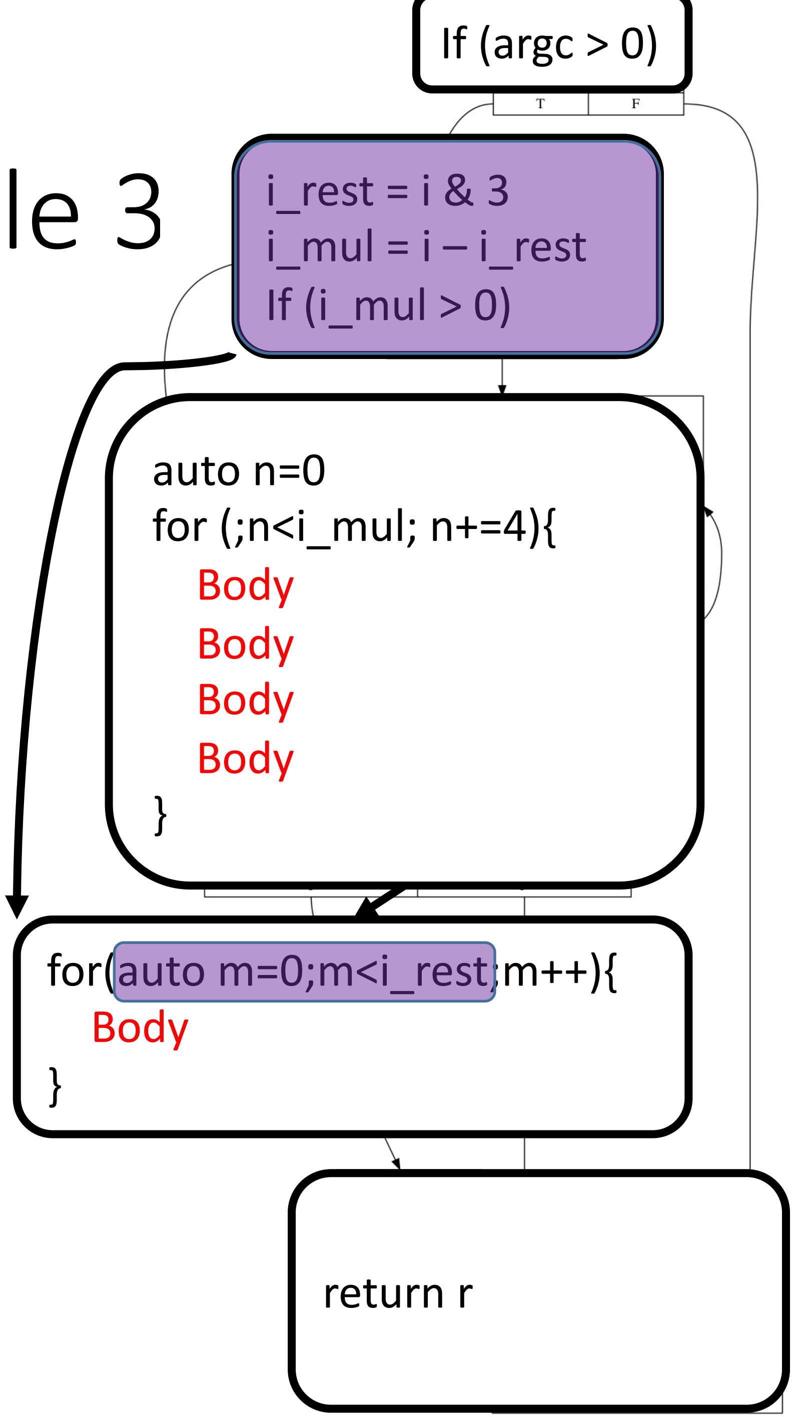
Loop unrolling in LLVM: example 3



CFG for 'main' function



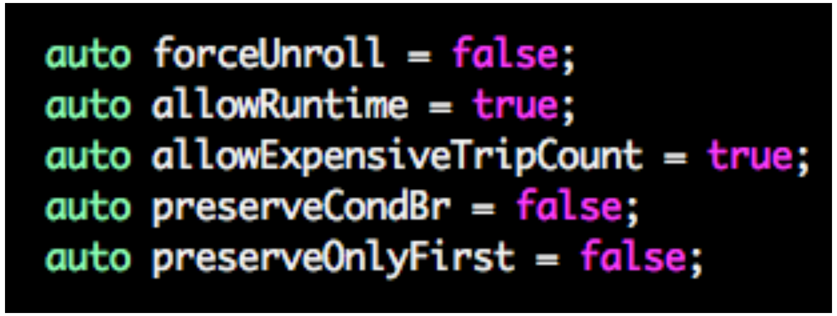
```
int main (int argc, char *argv[]){
  auto r = 0;

  for (auto i=0; i < argc; i++){
    r = myF(r);
  }

  return r;
}
```



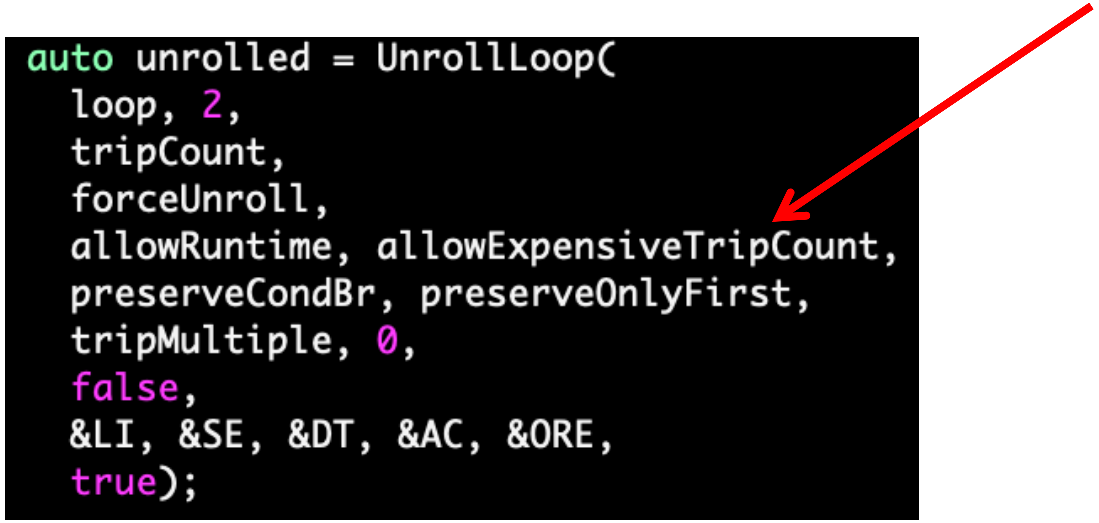
CFG for 'main' function

Runtime checks

Loop unrolling in LLVM: the runtime checks

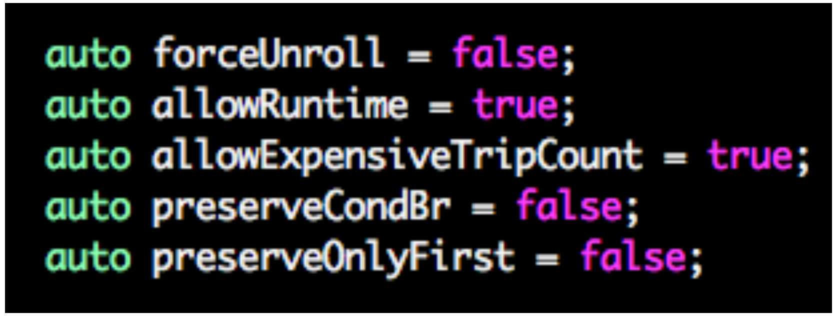


```
auto forceUnroll = false;
auto allowRuntime = true;
auto allowExpensiveTripCount = true;
auto preserveCondBr = false;
auto preserveOnlyFirst = false;
```

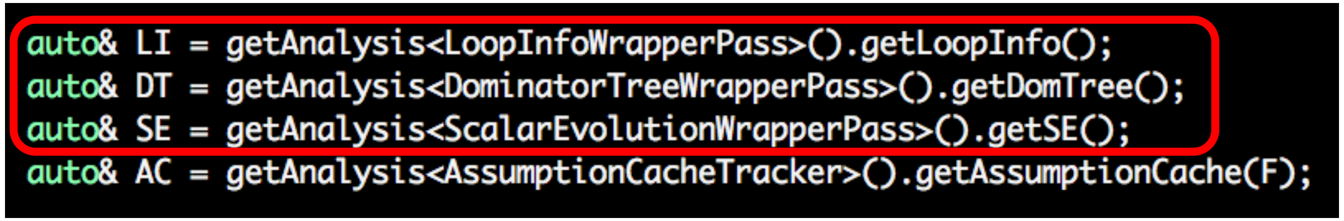


```
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  tripMultiple, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```

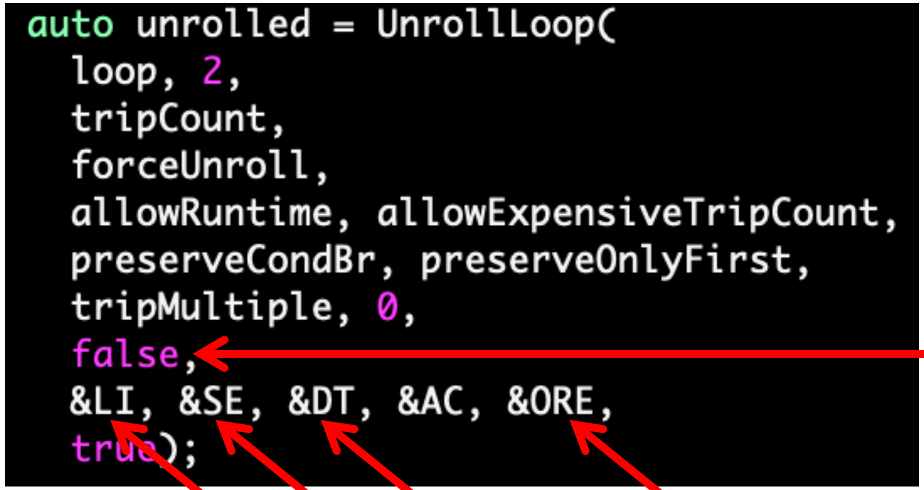
Loop unrolling in LLVM: API



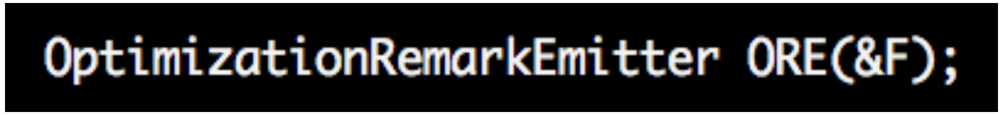
```
auto forceUnroll = false;
auto allowRuntime = true;
auto allowExpensiveTripCount = true;
auto preserveCondBr = false;
auto preserveOnlyFirst = false;
```



```
auto& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>().getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
```

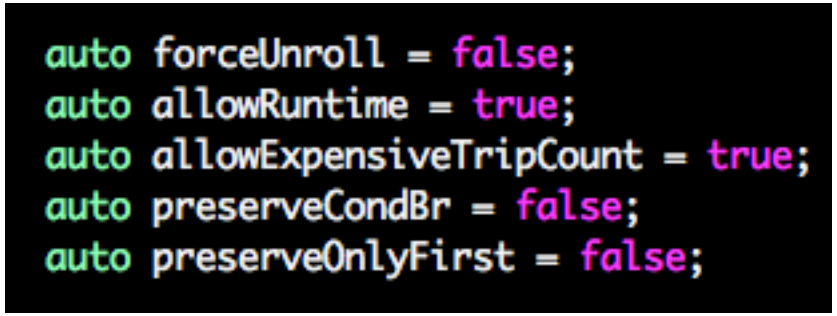


```
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  tripMultiple, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```

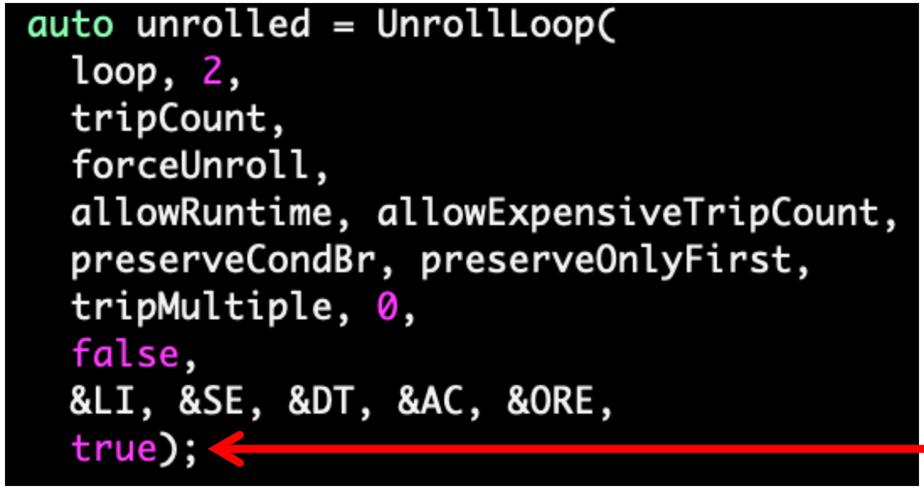


```
OptimizationRemarkEmitter ORE(&F);
```

Loop unrolling in LLVM: API



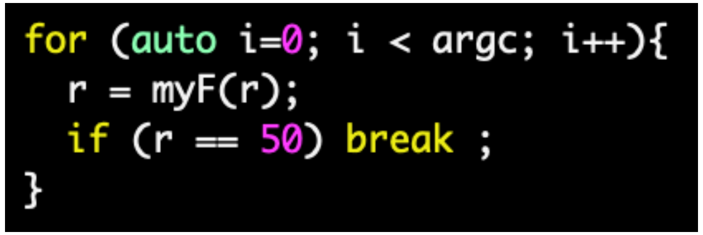
```
auto forceUnroll = false;
auto allowRuntime = true;
auto allowExpensiveTripCount = true;
auto preserveCondBr = false;
auto preserveOnlyFirst = false;
```



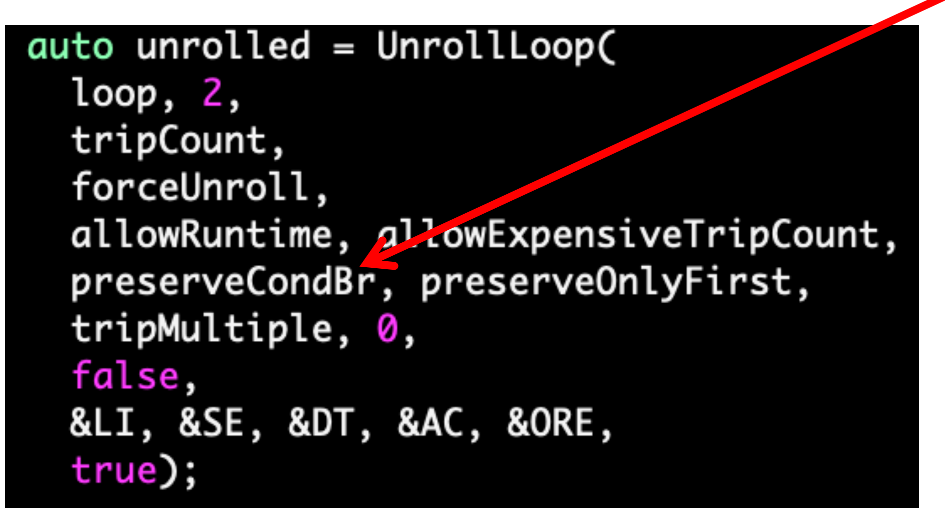
```
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  tripMultiple, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```

Normalize the generated loop to LCSSA

Code example



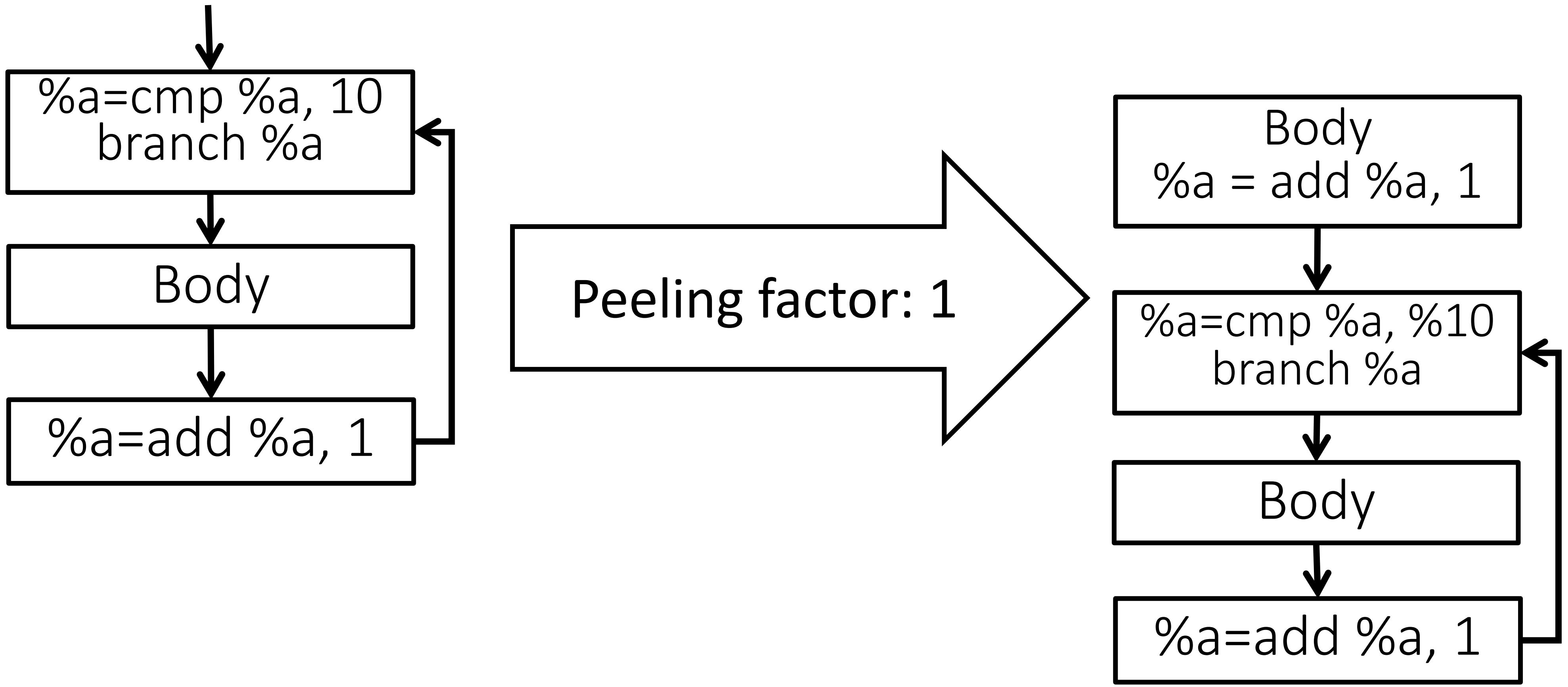
```
for (auto i=0; i < argc; i++){
  r = myF(r);
  if (r == 50) break ;
}
```



```
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  tripMultiple, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```

It needs to be set to true

Loop peeling



%a=cmp %a, 10
branch %a

Body

%a=add %a, 1

Peeling factor: 1

Body
%a = add %a, 1

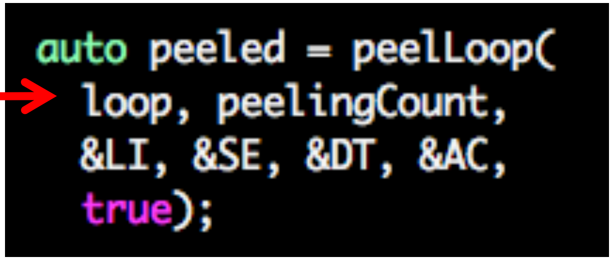
%a=cmp %a, %10
branch %a

Body

%a=add %a, 1

Loop peeling in LLVM

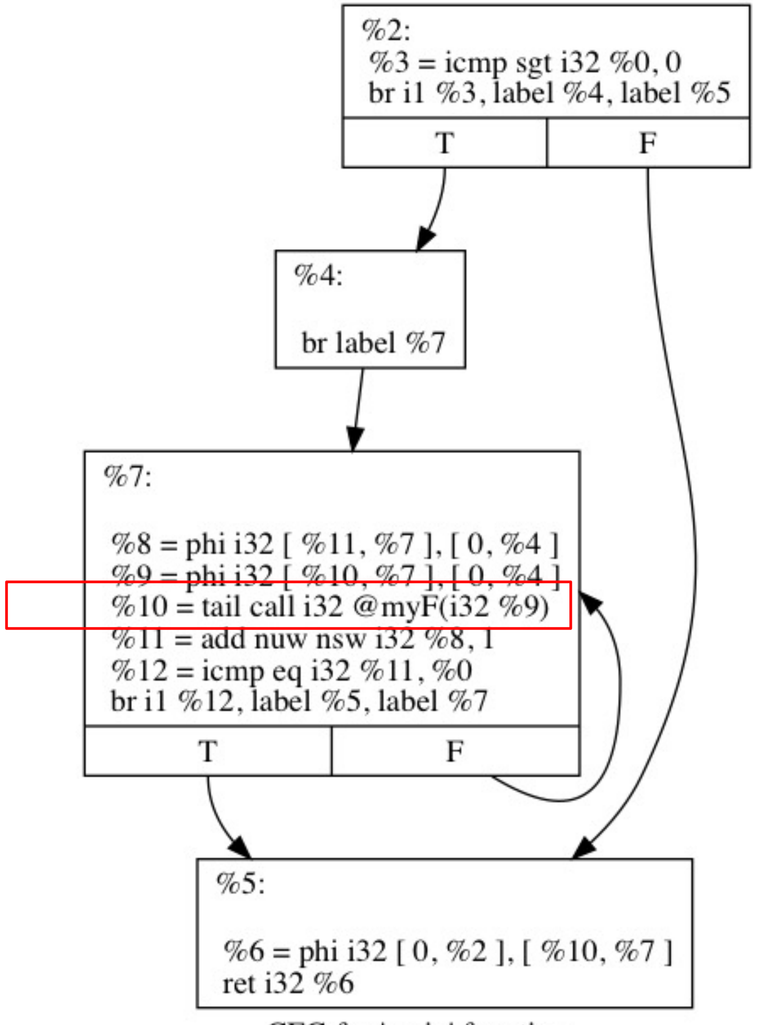
- API



```
auto peeled = peelLoop(
  loop, peelingCount,
  &LI, &SE, &DT, &AC,
  true);
```

  - No trip count
  - No flags
  - (almost) always possible
  - To check if you can peel, invoke the following API: bool canPeel(Loop *loop)

Loop peeling in LLVM: example



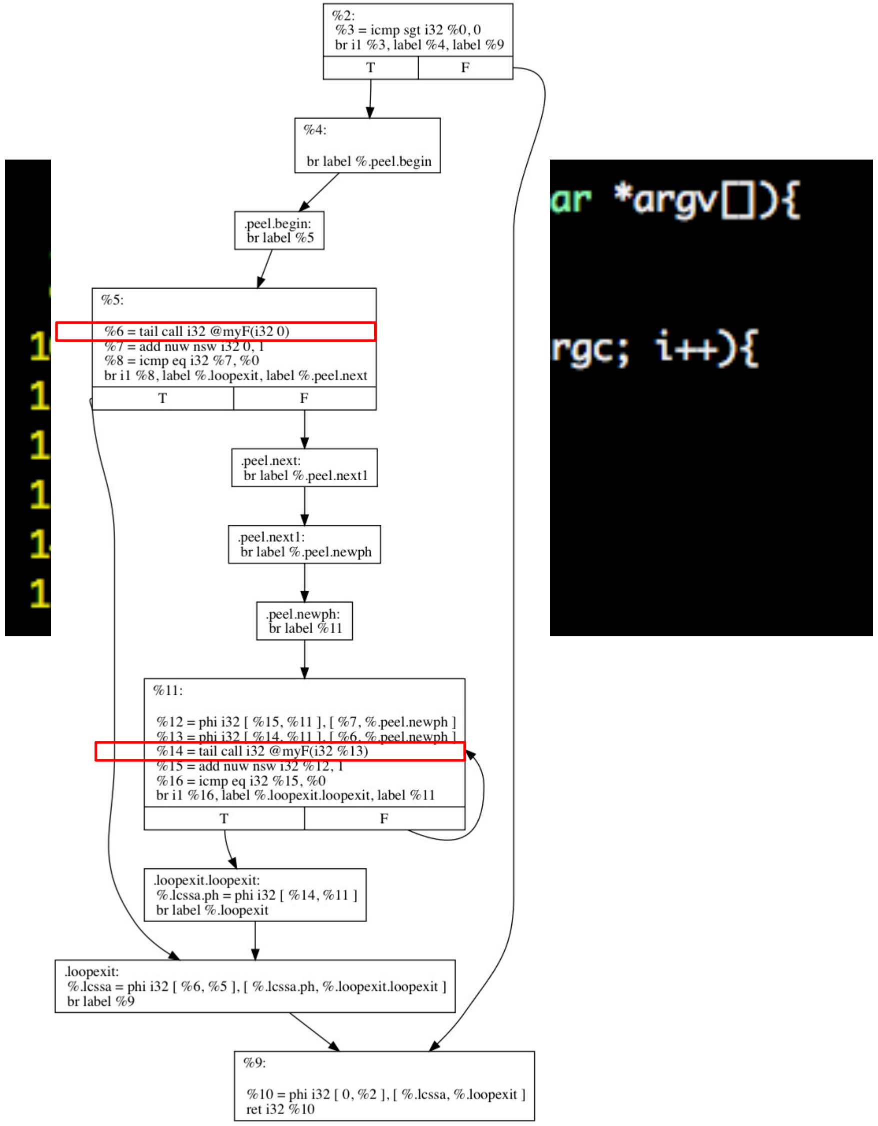
```
%2:
 %3 = icmp sgt i32 %0, 0
 br i1 %3, label %4, label %5
```

```
%4:

 br label %7
```

```
%7:

 %8 = phi i32 [ %11, %7 ], [ 0, %4 ]
 %9 = phi i32 [ %10, %7 ], [ 0, %4 ]
 %10 = tail call i32 @myF(i32 %9)
 %11 = add nuw nsw i32 %8, 1
 %12 = icmp eq i32 %11, %0
 br i1 %12, label %5, label %7
```

```
%5:

 %6 = phi i32 [ 0, %2 ], [ %10, %7 ]
 ret i32 %6
```

CFG for 'main' function



```
%2:
 %3 = icmp sgt i32 %0, 0
 br i1 %3, label %4, label %9
```

```
%4:

 br label %.peel.begin
```

```
.peel.begin:
 br label %5
```

```
%5:

 %6 = tail call i32 @myF(i32 0)
 %7 = add nuw nsw i32 0, 1
 %8 = icmp eq i32 %7, %0
 br i1 %8, label %.loopexit, label %.peel.next
```

```
.peel.next:
 br label %.peel.next1
```

```
.peel.next1:
 br label %.peel.newph
```

```
.peel.newph:
 br label %11
```

```
%11:

 %12 = phi i32 [ %15, %11 ], [ %7, %.peel.newph ]
 %13 = phi i32 [ %14, %11 ], [ %6, %.peel.newph ]
 %14 = tail call i32 @myF(i32 %13)
 %15 = add nuw nsw i32 %12, 1
 %16 = icmp eq i32 %15, %0
 br i1 %16, label %.loopexit.loopexit, label %11
```

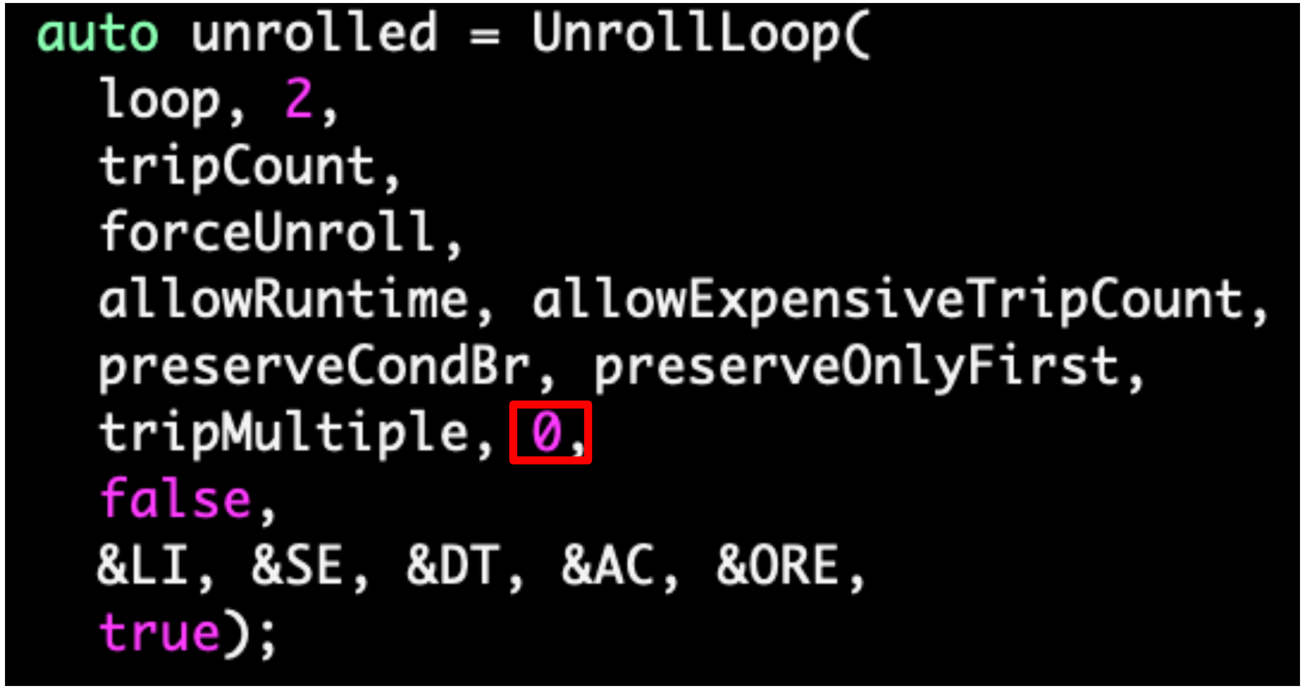
```
.loopexit.loopexit:
 %.lcssa.ph = phi i32 [ %14, %11 ]
 br label %.loopexit
```

```
.loopexit:
 %.lcssa = phi i32 [ %6, %5 ], [ %.lcssa.ph, %.loopexit.loopexit ]
 br label %9
```

```
%9:

 %10 = phi i32 [ 0, %2 ], [ %.lcssa, %.loopexit ]
 ret i32 %10
```

CFG for 'main' function

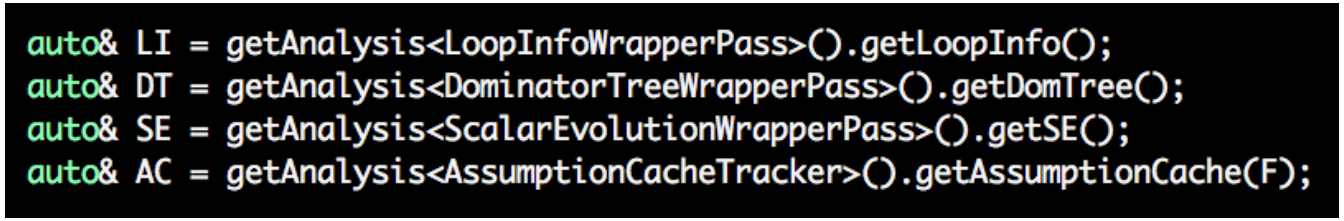
Loop unrolling and peeling together



```
auto unrolled = UnrollLoop(
  loop, 2,
  tripCount,
  forceUnroll,
  allowRuntime, allowExpensiveTripCount,
  preserveCondBr, preserveOnlyFirst,
  tripMultiple, 0,
  false,
  &LI, &SE, &DT, &AC, &ORE,
  true);
```

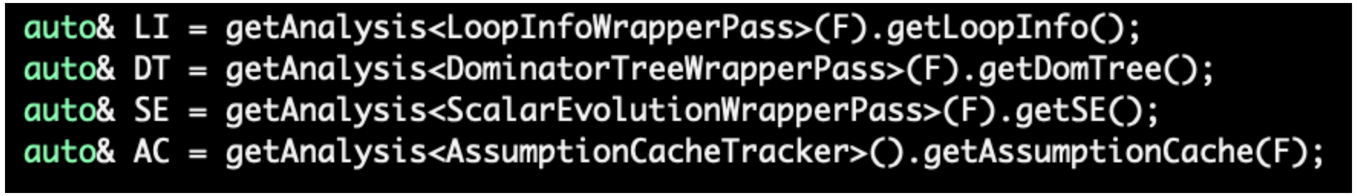
Fetching analyses outputs from a module pass

- From a function pass



```
auto& LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>().getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>().getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
```

- From a module pass



```
auto& LI = getAnalysis<LoopInfoWrapperPass>(F).getLoopInfo();
auto& DT = getAnalysis<DominatorTreeWrapperPass>(F).getDomTree();
auto& SE = getAnalysis<ScalarEvolutionWrapperPass>(F).getSE();
auto& AC = getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
```

Outline

- Simple loop transformations

- Loop invariants based transformations

- Induction variables based transformations

- Complex loop transformations

Optimizations in small, hot loops

- Most programs: 90% of time is spent in few, small, hot loops

```
while (){
  statement 1
  statement 2
  statement 3
}
```

- Deleting a single statement from a small, hot loop
  might have a big impact
  (100 seconds -> 70 seconds)

Loop example

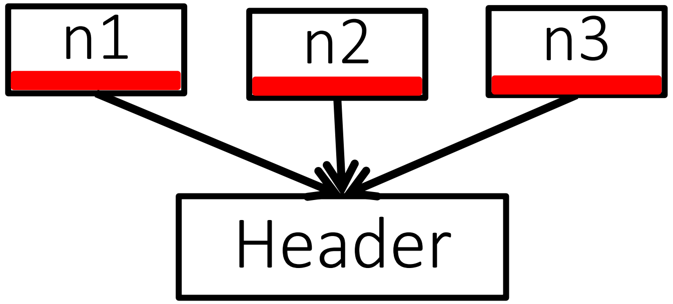
```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
   do {
3:   a = 1;
4:   y = x + N;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10:  x++;
11:} while (x < N);
```

- **Observation**: each statement in that loop will contribute to the program execution time
- **Idea**: what about moving statements from inside a loop to outside it?
- Which statements can be moved outside our loop?
- How to identify them automatically? (code analysis)
- How to move them? (code transformation)

Hoisting code

- In order to “hoist” a loop-invariant computation out of a loop, we need a place to put it
- We could copy it to all immediate predecessors of the loop header...

```
for (auto pBB : predecessors(H)){
  p = pBB->getTerminator();
  inv->moveBefore(p);
}
```



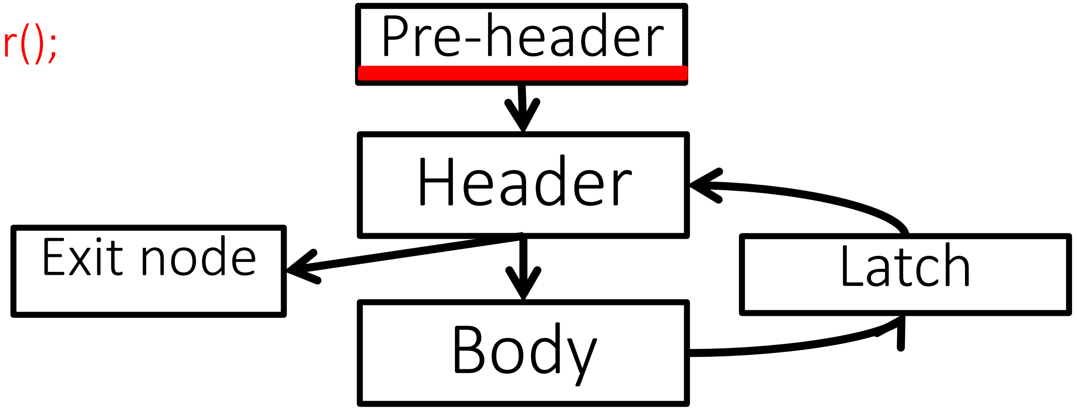
Is it correct?

- ...But we can avoid code duplication (and bugs) by taking advantage of loop normalization that guarantees the existence of the pre-header

Hoisting code

- In order to "hoist" a loop-invariant computation out of a loop, we need a place to put it
- We could copy it to all immediate predecessors of the loop header...

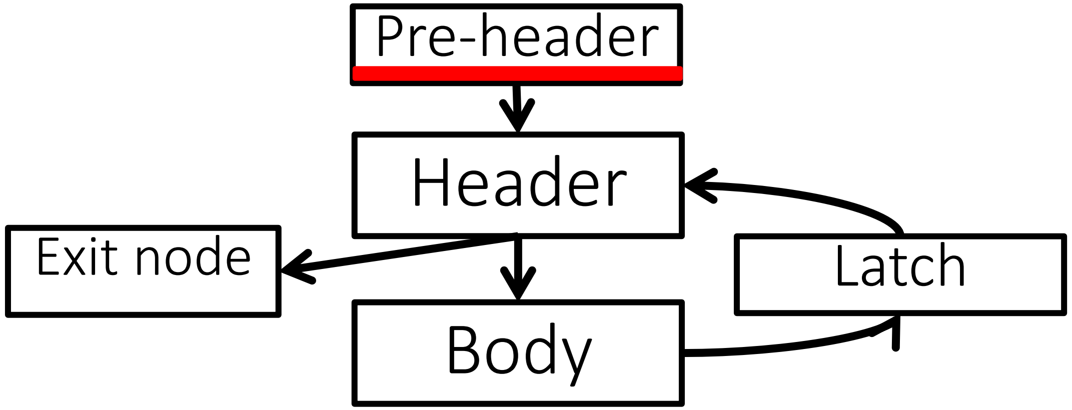
```
pBB = loop->getLoopPreheader();
p = pBB->getTerminator();
inv->moveBefore(p);
```



- ...but we can avoid code duplication (and bugs) by taking advantage of loop normalization that guarantees the existence of the pre-header

Can we hoist all invariant instructions of a loop L in the pre-header of L?

```
for (inv : invariants(loop)){
  pBB = loop->getLoopPreheader();
  p = pBB->getTerminator();
  inv->moveBefore(p);
}
```



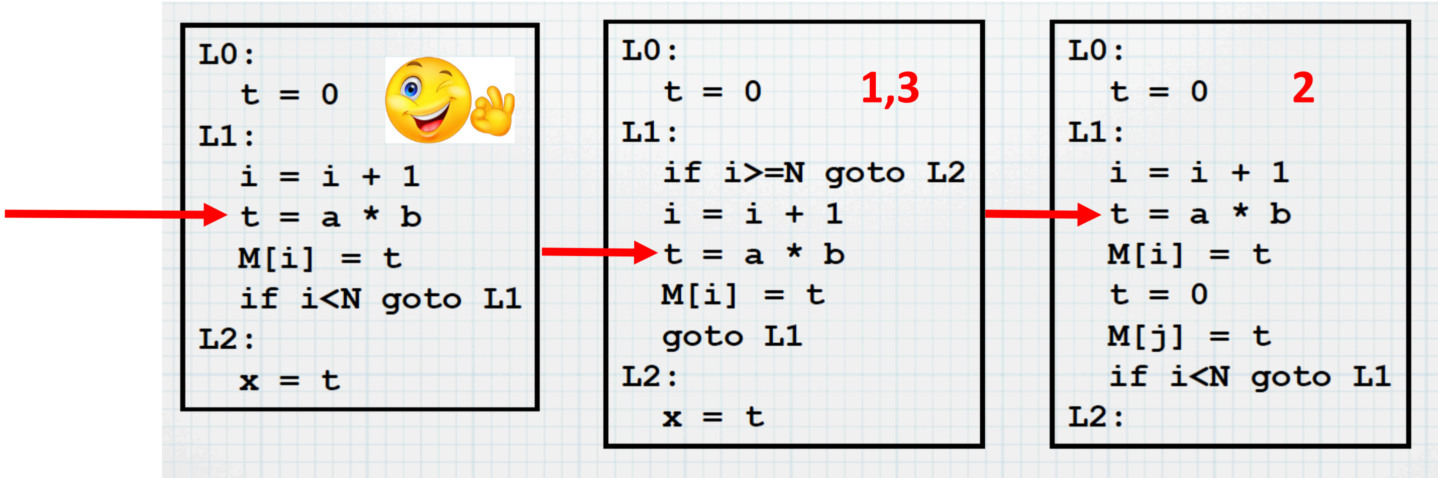
Hoisting conditions

Loop invariant code motion

- For a loop-invariant definition

(d) t = x op y

- We can hoist d into the loop’s pre-header if ??
  1. d dominates all loop exits at which t is live-out, and
  2. there is only one definition of t in the loop, and
  3. t is not live-out of the pre-header



```
L0:
   t = 0
L1:
   i = i + 1
   t = a * b
   M[i] = t
   if i<N goto L1
L2:
   x = t
```

```
L0:
   t = 0
L1:
   if i>=N goto L2
   i = i + 1
   t = a * b
   M[i] = t
   goto L1
L2:
   x = t
```

```
L0:
   t = 0
L1:
   i = i + 1
   t = a * b
   M[i] = t
   t = 0
   M[j] = t
   if i<N goto L1
L2:
```

Outline

- Simple loop transformations
- Loop invariants based transformations
- Induction variables based transformations
- Complex loop transformations

Loop example

```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
    do {
3:   a = 1;
4:   y = x + N;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:      m = 5;
9:      break;
     }
10:  x++;
11:} while (x < N);
```

Assuming a,b,c,m are used after our code

Do we have to execute 4 for every iteration?

Do we have to execute 10 for every iteration?

Loop example

```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
   do {
3:   a = 1;
4:   y = x + N;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10:  x++;
11:} while (x < N);
```

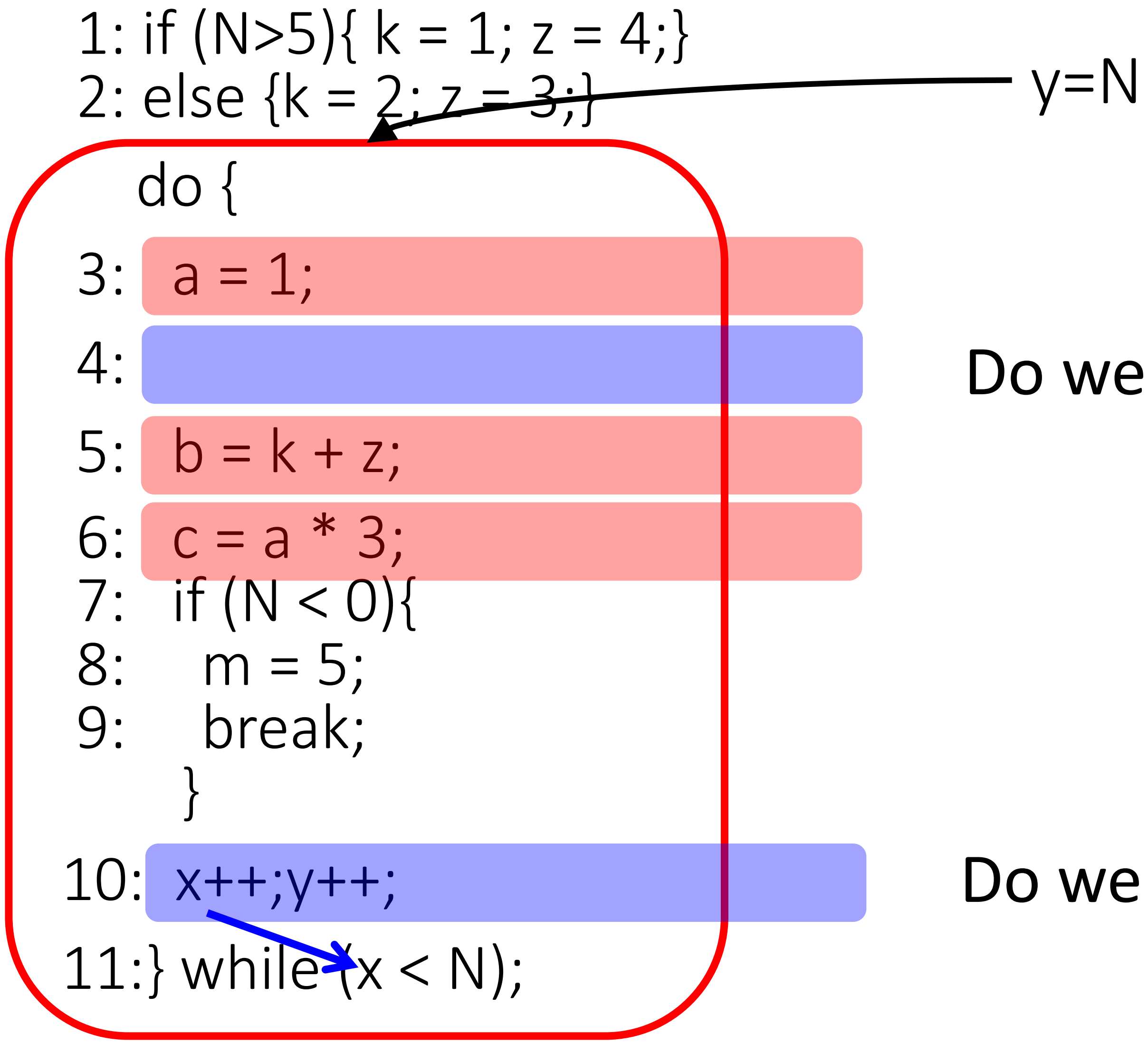
y=N

Do we have to execute 4 for every iteration?

Compute manually values of x and y
for every iteration
What do you see?

Do we have to execute 10 for every iteration?

Loop example

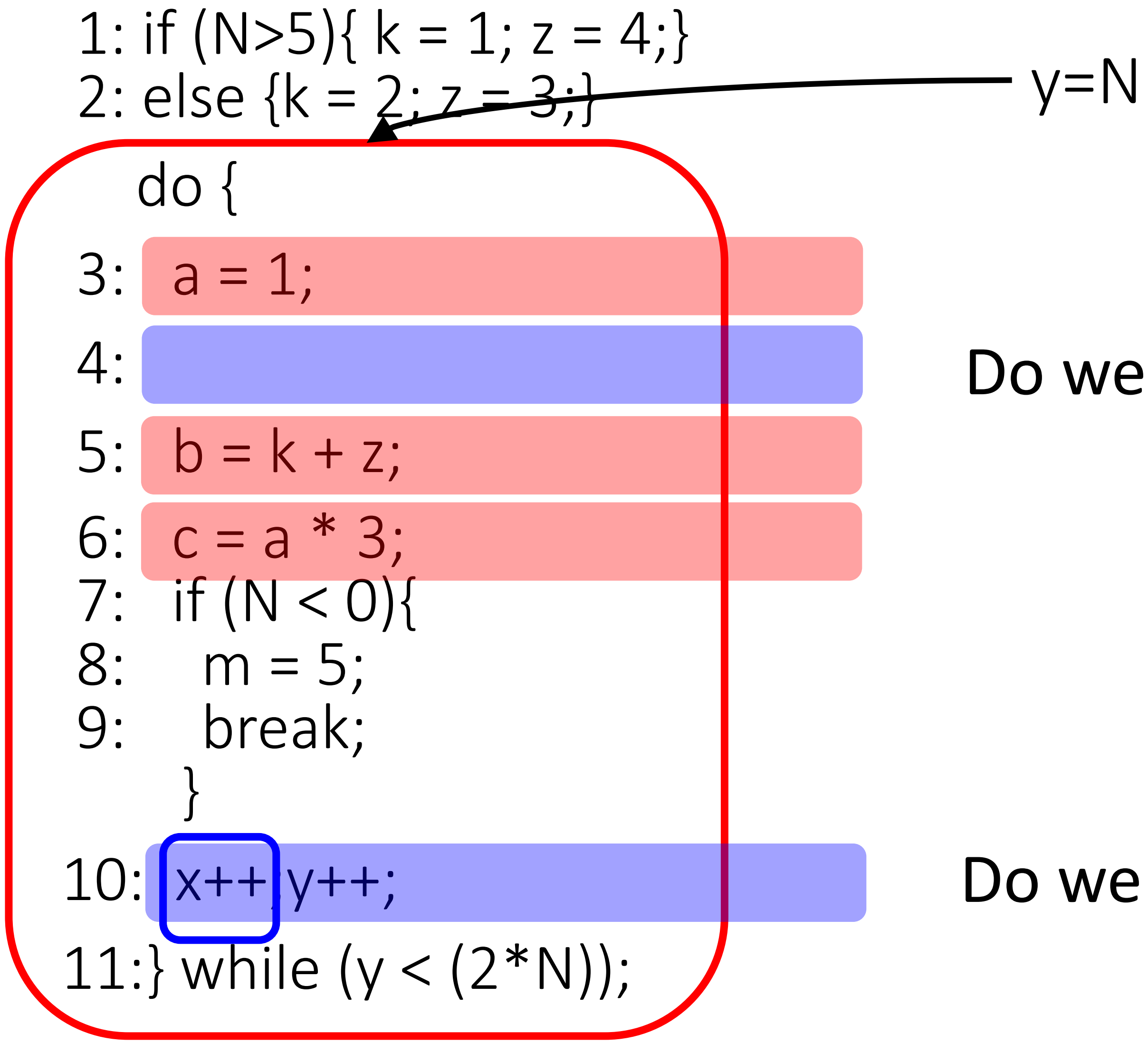


```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
   do {
3:  a = 1;
4:
5:  b = k + z;
6:  c = a * 3;
7:  if (N < 0){
8:     m = 5;
9:     break;
    }
10: x++;y++;
11:} while (x < N);
```

Do we have to execute 4 for every iteration?

Do we have to execute 10 for every iteration?

Loop example



```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
   do {
3:   a = 1;
4:
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10:  x++;y++;
11:} while (y < (2*N));
```

y=N

Do we have to execute 4 for every iteration?

Do we have to execute 10 for every iteration?

Loop example

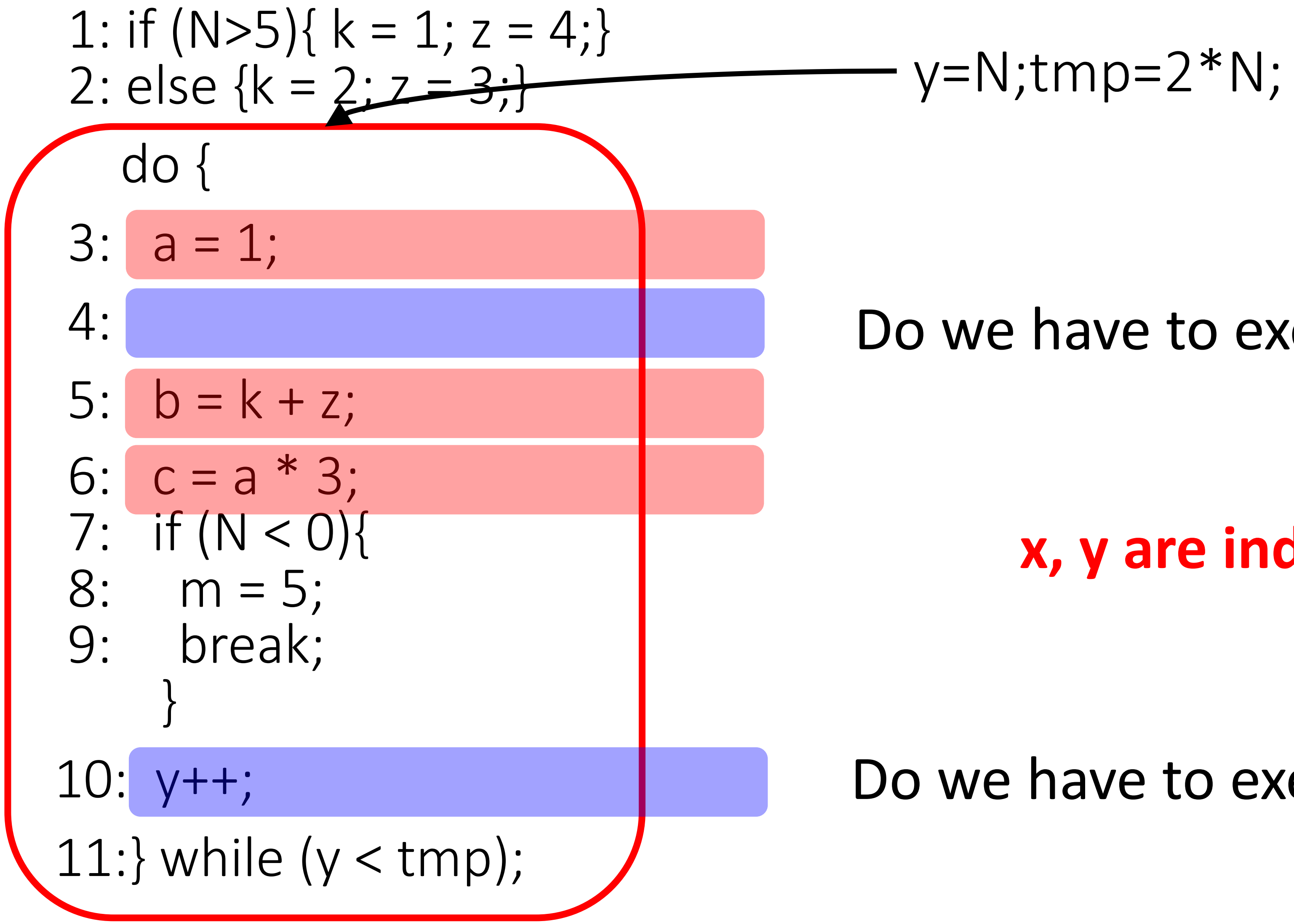
```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
   do {
3:   a = 1;
4:
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10:  y++;
11:} while (y < (2*N));
```

y=N

Do we have to execute 4 for every iteration?

Do we have to execute 10 for every iteration?

Loop example



```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
   do {
3:   a = 1;
4:
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10:  y++;
11:} while (y < tmp);
```

y=N;tmp=2*N;

Do we have to execute 4 for every iteration?

x, y are induction variables

Do we have to execute 10 for every iteration?

Is the code transformation worth it?

```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
A :y=N;tmp=2*N;
   do {
3:   a = 1;

5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10:  y++;
11:} while (y < tmp);
```

Induction variable elimination

```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
   do {
3:   a = 1;
4:   y = x + N;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10:  x++;
11:} while (x < N);
```

… and after Loop Invariant Code Motion …

```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}
A :y=N;tmp=2*N;
3 :a=1;
5 :b=k+z;
6: c=a*3;

    do{
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10: y++;
11:} while (y < tmp);
```

```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}

    do {
3:   a = 1;
4:   y = x + N;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10: x++;
11:} while (x < N);
```

… and with a better Loop Invariant Code Motion …

```
 1: if (N>5){ k = 1; z = 4;}
 2: else {k = 2; z = 3;}
A :y=N;tmp=2*N;
3 :a=1;
5 :b=k+z;
6: c=a*3;
7: if (N < 0){
8:    m=5;
    }

    do{
10:      y++;
11:} while (y < tmp);
```

```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}

    do {
3:   a = 1;
4:   y = x + N;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10:  x++;
11:} while (x < N);
```

… and after dead code elimination …

```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}

3 :a=1;
5 :b=k+z;
6: c=a*3;
7: if (N < 0){
8:    m=5;
    }
```

Assuming a,b,c,m are used after our code

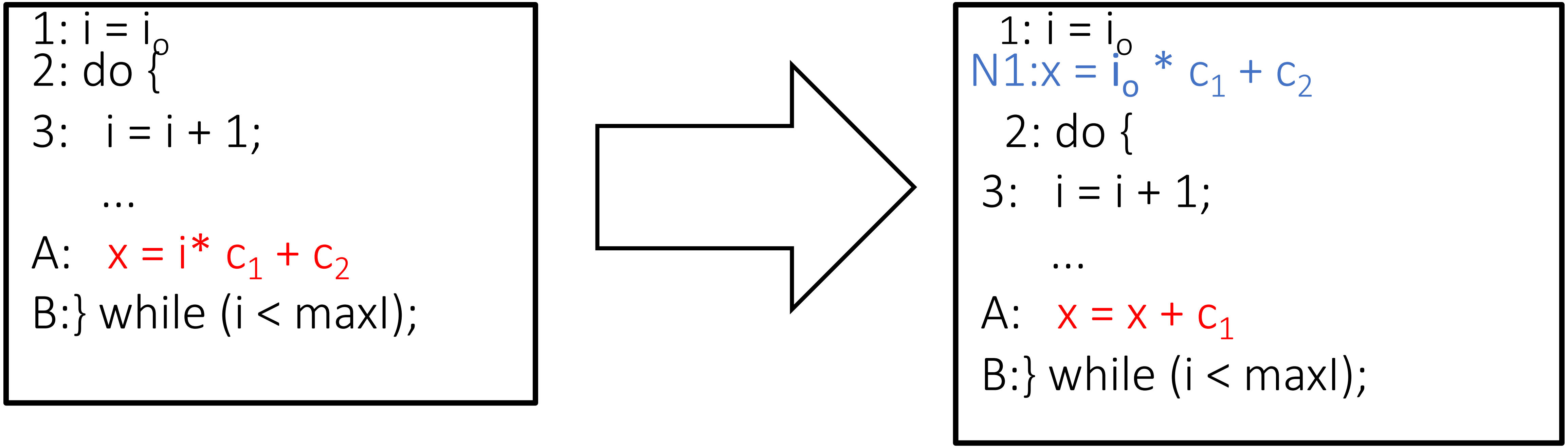
```
1: if (N>5){ k = 1; z = 4;}
2: else {k = 2; z = 3;}

    do {
3:   a = 1;
4:   y = x + N;
5:   b = k + z;
6:   c = a * 3;
7:   if (N < 0){
8:     m = 5;
9:     break;
     }
10:  x++;
11:} while (x < N);
```

Induction variable elimination

- Suppose we have a loop variable
  - i initially set to i_0; each iteration i = i + 1
- and a variable that linearly depends on it
  - $x = i * c_1 + c_2$

    constants
- We can
  - Initialize $x = i_o * c_1 + c_2$
  - Increment x by c_1 each iteration

Is it faster?



```
1: i = $i_o$
2: do {
3:   i = i + 1;
     ...
A:   $x = i * c_1 + c_2$
B:} while (i < maxI);
```

```
 1: i = $i_o$
N1:$x = i_o * c_1 + c_2$
 2: do {
3:   i = i + 1;
      ...
A:   $x = x + c_1$
B:} while (i < maxI);
```

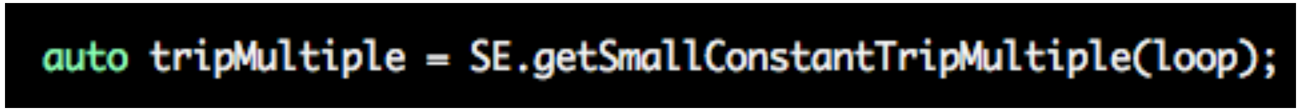
On some hardware, adds are much faster than multiplies

- Strength reduction

Many optimizations rely on IVs

- Like induction variable elimination we have seen before

- or like loop unrolling to compute the trip count



```
auto tripMultiple = SE.getSmallConstantTripMultiple(loop);
```

Induction variable elimination: step 1

①Iterate over IVs

k = j * c1 + c2

- where IV j =(i, a, b), and
- this is the only def of k in the loop, and
- there is no def of i between the def of j and the def of k

```
i = …
…
j = i …
…
k = j …
```

②Record as k = (i, a*c1, b*c1+c2)

Induction variable elimination: step 2

For an induction variable k = (i, c1, c2)

① Initialize k = i * c1 + c2 in the pre-header

② Replace k’s def in the loop by k = k + c1

- Make sure to do this after i’s definition

Outline

- Simple loop transformations

- Loop invariants based transformations

- Induction variables based transformations

- Complex loop transformations

Loop transformations

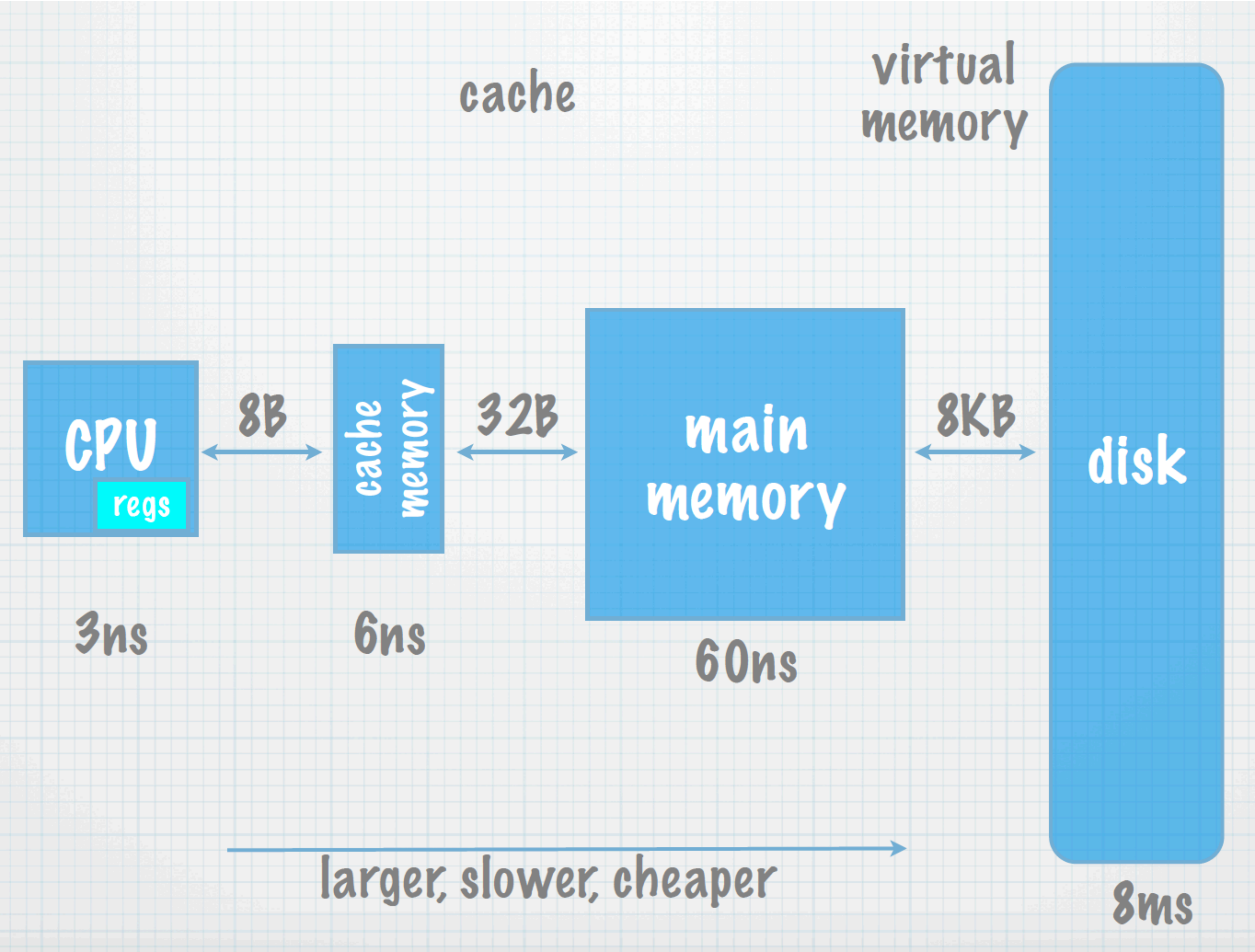
- Restructure a loop to expose more optimization opportunities and/or transform the “loop overhead”
  - Loop unrolling, loop peeling, …

- Reorganize a loop to improve memory utilization
  - Cache blocking, skewing, loop reversal

- Distribute a loop over cores/processors
  - DOACROSS, DOALL, DSWP, HELIX

Loo
for r

- Ho


cache
virtual memory
CPU
regs
8B
cache memory
32B
main memory
8KB
disk
3ns
6ns
60ns
8ms
larger, slower, cheaper

Goal: improve cache performance

- **Temporal locality**

  A resource that has just been referenced
  will more likely be referenced again in the near future

- **Spatial locality**

  The likelihood of referencing a resource is higher
  if a resource near it was just referenced

- Ideally, a compiler generates code
  with high temporal and spatial locality
  for the target architecture
  - What to minimize: bad replacement decisions

What a compiler can do

- Time:
    - When is an object accessed?

- Space:
    - Where does an object exist in the address space?

- These are the two “knobs” a compiler can manipulate

Manipulating time and space

- Time: reordering computation
  - Determine when an object will be accessed, and predict a better time to access it

- Space: changing data layout
  - Determine an object’s shape and location, and determine a better layout

First understand cache behavior ...

- When do cache misses occur?
  - Use locality analysis

- Can we change the visitation order to produce better behavior?
  - Evaluate costs

- Does the new visitation order still produce correct results?
  - Use dependence analysis

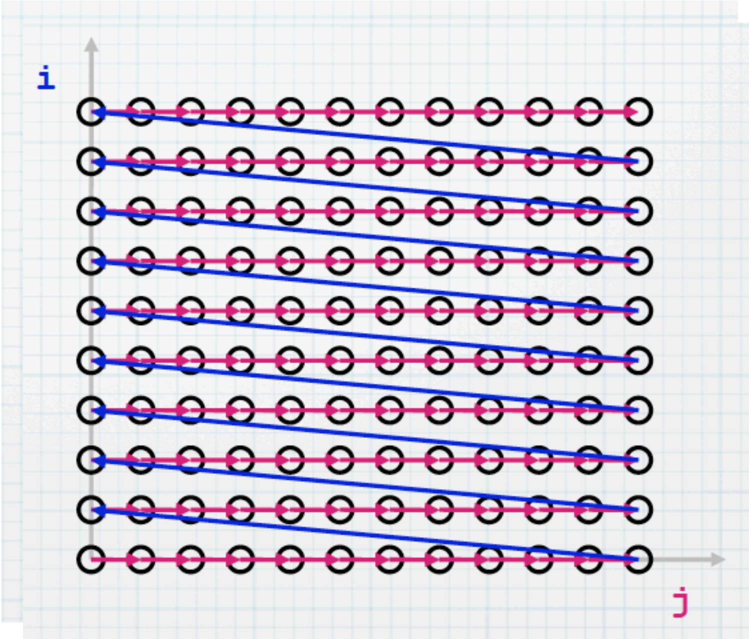
… and then rely on loop transformations

- loop interchange
- cache blocking
- loop fusion
- loop reversal
- …

Code example

```
double A[N][N], B[N][N];
...
for i = 0 to N-1{
  for j = 0 to N-1{
     ... = A[i][j] ...
  }
}
```

Iteration space for A

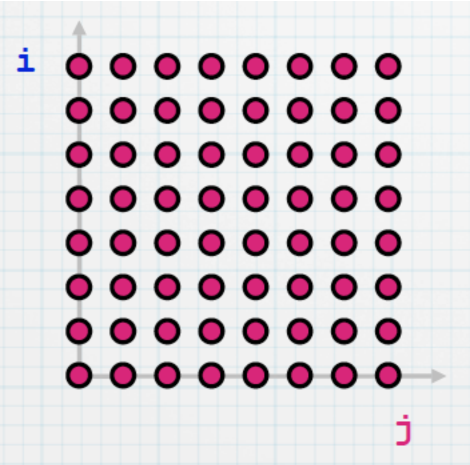


Loop interchange

```
for i = 0 to N-1
  for j = 0 to N-1
    … = A[j][i] …
```

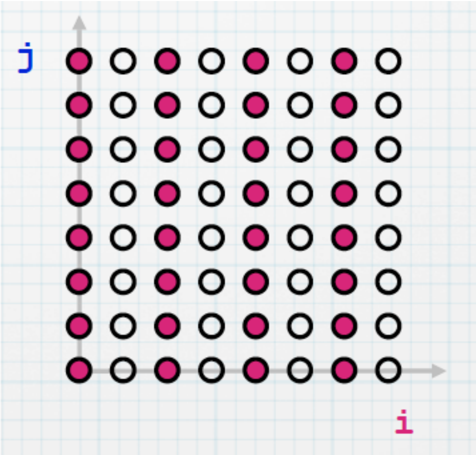
```
For j = 0 to N-1
  for i = 0 to N-1
    … = A[j][i] …
```

Assumptions: N is large; A is row-major; 2 elements per cache line



○ Hit
● Miss

A[][] in C? Java?



Java (similar in C)

To create a matrix:

~~double [][] A = new double[3][3];~~

A is an array of arrays

A is not a 2 dimensional array!

Java (similar in C)

To create a matrix:

```
double [][] A = new double[3][];
A[0] = new double[3];
A[1] = new double[3];
A[2] = new double[3];
```

Java (similar in C)

To create a matrix:

```
double [][] A = new double[3][];
A[0] = new double[10];
A[1] = new double[5];
A[2] = new double[42];
```

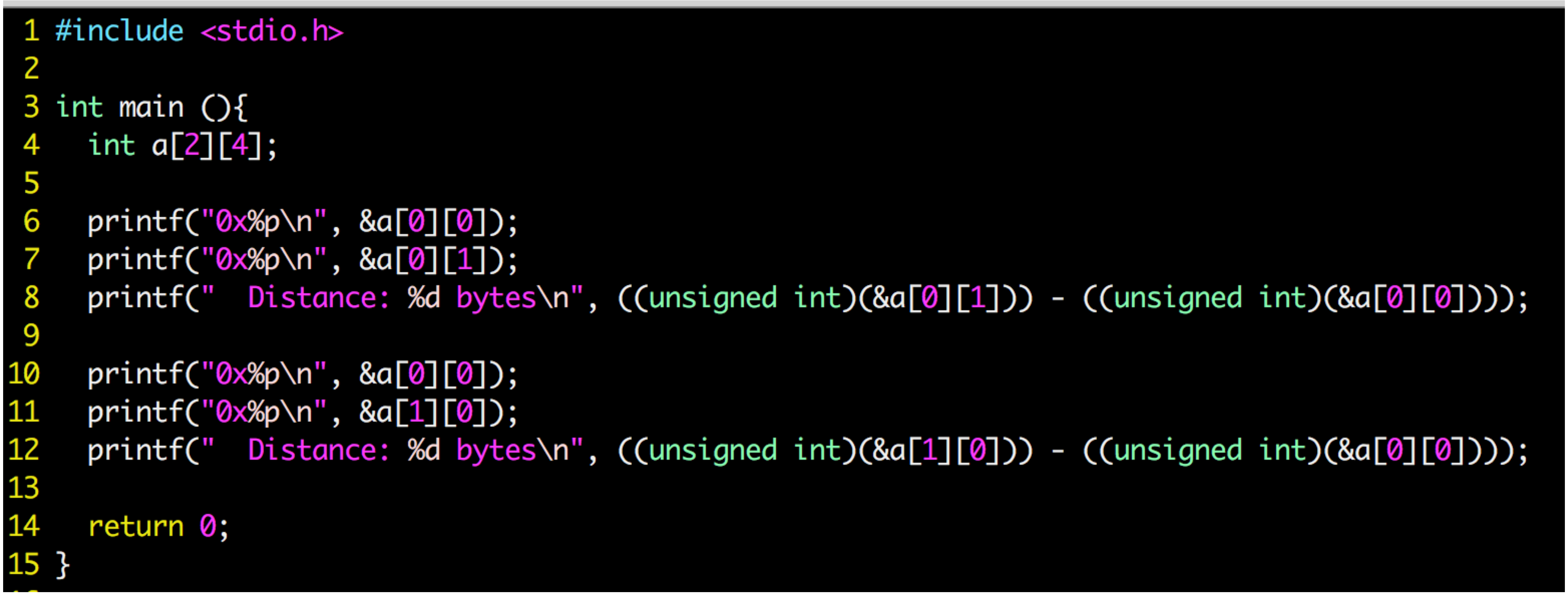
A is a jagged array

C#: [][] vs. [,]

```
double [][] A = new double[3][];
A[0] = new double[3];
A[1] = new double[3];
A[2] = new double[3];
```

```
double [,] A = new double[3,3];
```

The compiler can easily choose between raw-major vs. column-major



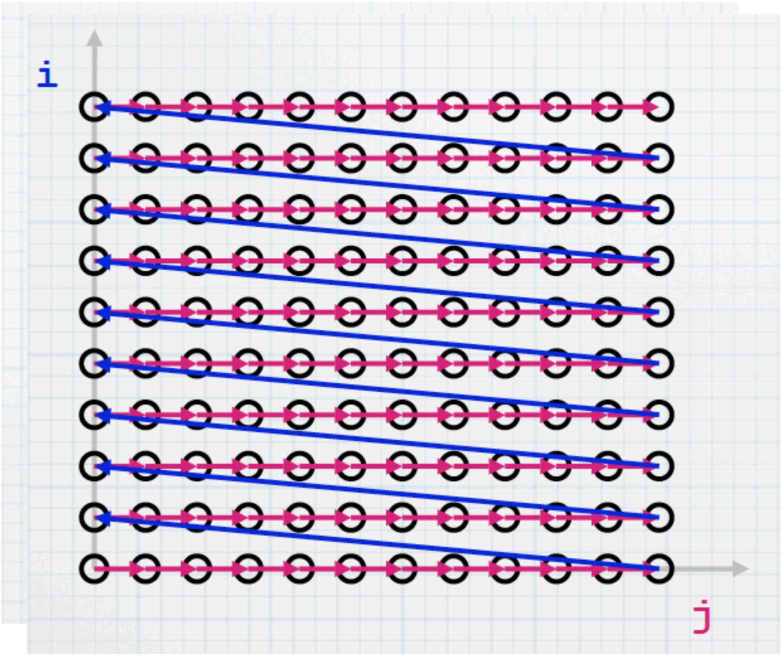
```c
#include <stdio.h>

int main (){
  int a[2][4];

  printf("0x%p\n", &a[0][0]);
  printf("0x%p\n", &a[0][1]);
  printf("  Distance: %d bytes\n", ((unsigned int)(&a[0][1])) - ((unsigned int)(&a[0][0])));

  printf("0x%p\n", &a[0][0]);
  printf("0x%p\n", &a[1][0]);
  printf("  Distance: %d bytes\n", ((unsigned int)(&a[1][0])) - ((unsigned int)(&a[0][0])));

  return 0;
}
```

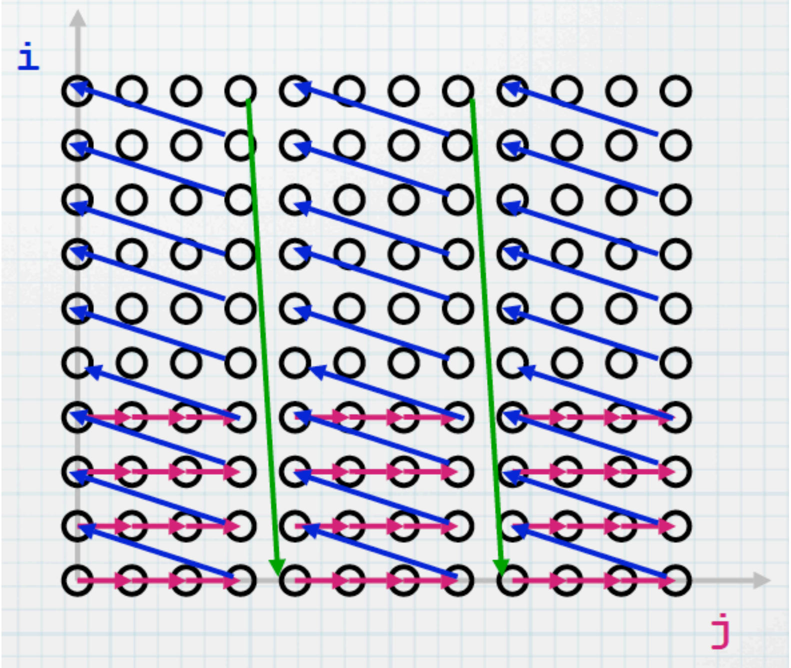
Cache blocking (a.k.a. tiling)

```
for i = 0 to N-1
  for j = 0 to N-1
    f(A[i], A[j])
```

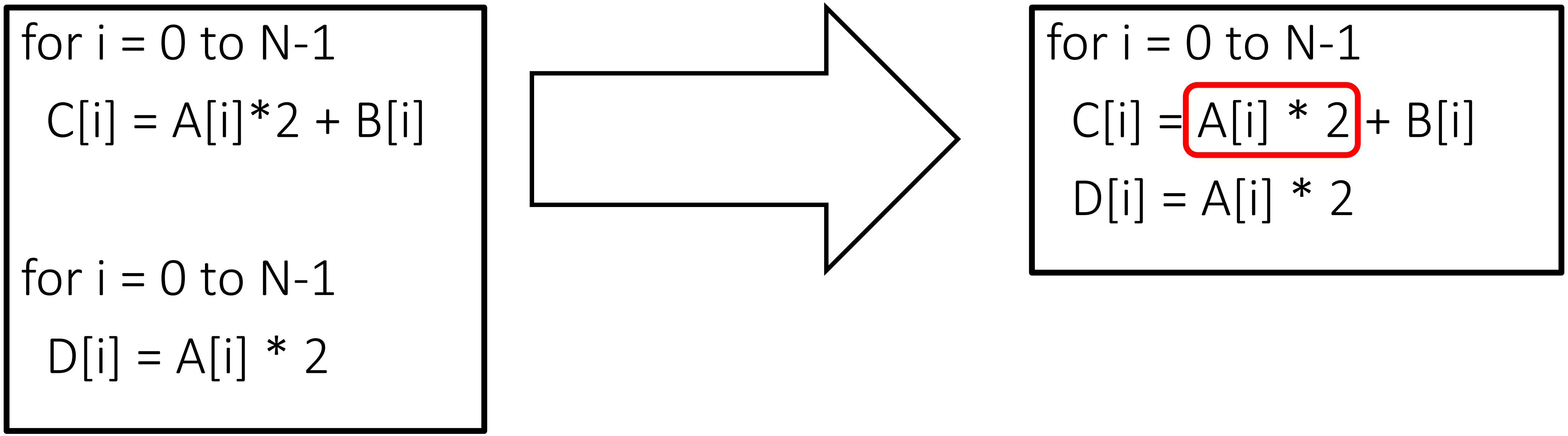
→

```
for JJ = 0 to N-1 by B
  for i = 0 to N-1
    for j = JJ to min(N-1,JJ+B-1)
      f(A[i], A[j])
```





Loop fusion



```
for i = 0 to N-1
  C[i] = A[i]*2 + B[i]

for i = 0 to N-1
  D[i] = A[i] * 2
```

```
for i = 0 to N-1
  C[i] = A[i] * 2 + B[i]
  D[i] = A[i] * 2
```

- Reduce loop overhead
- Improve locality by combining loops that reference the same array
- Increase the granularity of work done in a loop

Locality analysis

- Reuse:
  Accessing a location that has been accessed previously
- Locality:
  Accessing a location that is in the cache

- Observe:
  - Locality only occurs when there is reuse!
  - … but reuse does not imply locality

Steps in locality analysis

- Find data reuse
- Determine “localized iteration space”
    - Set of inner loops where the data accessed by an iteration is expected to fit within the cache
- Find data locality
    - Reuse ∩ localized iteration space ⇒ locality