Loops

Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• Loops

• Identify loops

• Induction variables

• Loop normalization
Impact of optimized code to program

10 seconds

Code transformation

1 second

How much did we optimize the overall program?
- Coverage of optimized code
  - 10% coverage: Speedup=\(~1.10x\) (100->91 seconds)
  - 20% coverage: Speedup=\(~1.22x\) (100->82 seconds)
  - 90% coverage: Speedup=\(~5.26x\) (100->19 seconds)
90% of time is spent in 10% of code

Identify hot code to succeed!!!
Loops ...
... but where are they?
... How can we find them?
Loops in source code

for (i=0; i < 10; i++){
    ...
}

i=0;
do {
    ...
i++;
} while (i < 10);

S={0,1,...,10}
for (i : S){
    ...
}

Is there a LLVM IR instruction “for”? There is no IR instruction for “loop”
• Target optimization: we need to identify loops
• There is no IR instruction for “loop”
• How to identify an IR loop?
Loops in IR

• Loop identification control flow analysis:
  • Input: Control-Flow-Graph
  • Output: loops in CFG
  • Not sensitive to input syntax: a uniform treatment for all loops

• Define a loop in graph terms

• Intuitive properties of a loop
  • Single entry point
  • Edges must form at least a cycle in CFG

• How to check these properties automatically?
Outline
• Loops
  • Identify loops
  • Induction variables
• Loop normalization
Natural loops in CFG

- **Header**: node that dominates all other nodes in a loop
  Single entry point of a loop

- **Back edge**: edge (tail -> head) whose head dominates its tail

- **Natural loop** of a back edge:
  smallest set of nodes that includes the head and tail of that back edge,
  and has no predecessors outside the set, except for the predecessors of the header.
Identify natural loops

① Find the dominator relations in a flow graph

② Identify the back edges

③ Find the natural loop associated with the back edge
Immediate dominators

**Definition:** the immediate dominator of a node $n$ is the unique node that strictly dominates $n$ (i.e., it isn’t $n$) but does not strictly dominate another node that strictly dominates $n$.
Finding back-edges

Definition:
a back-edge is an arc (tail -> head) whose head dominates its tail

(A) Depth-first spanning tree
Spanning tree of a graph

Definition:
A tree \( T \) is a *spanning tree* of a graph \( G \) if
\( T \) is a subgraph of \( G \) that contains all the vertices of \( G \).
Depth-first spanning tree of a graph

Idea:
Make a path as long as possible,
and then go back (backtrack) to add branches also as long as possible.

Algorithm
s = new Stack(); s.add(G.entry); mark(G.entry);
While (!s.empty()){
  1: v = s.pop();
  2: if (v’ = adjacentNotMarked(v, G)){
      3: mark(v’); DFST.add((v, v’));
      4: s.push(v’);
  }
}
Finding back-edges

Definition:
a back-edge is an arc (tail -> head) whose head dominates its tail

(A) Depth-first spanning tree
• Compute retreating edges in CFG:
  • Advancing edges: from ancestor to proper descendant
  • Retreating edges: from descendant to ancestor

(B) For each retreating edge t->h, check if h dominates t
• If h dominates t, then t->h is a back-edge
Finding natural loops

**Definition**: the natural loop of a back edge is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.

Let $t \rightarrow h$ be the back-edge.

A. Delete $h$ from the flow graph.

B. Find those nodes that can reach $t$ (those nodes plus $h$ and $t$ form the natural loop of $t \rightarrow h$)
Natural loop example

For (int i=0; i < 10; i++){
    A();
    while (j < 5){
        j = B(j);
    }
}

Diagram:
- Node 0: i=0
- Node 1: i < 10
- Node 2: A()
- Node 3: j < 5
- Node 4: j = B(j)
- Node 5: i++
- Exit
Identify inner loops

• If two loops do not have the same header
  • They are either disjoint, or
  • One is entirely contained (nested within) the other
    • Outer loop, inner loop
    • Loop nesting relation

• What about if two loops share the same header?

```java
while (a: i < 10){
    b: if (i == 5) continue;
    c: ...
}
```

Graph/DAG/tree? Why?
Loop nesting tree

- **Loop-nest tree**: each node represents the blocks of a loop, and parent nodes are enclosing loops.
- The leaves of the tree are the inner-most loops.

How to compute the loop-nest tree?
void myFunction (){ 
  1: while (...){
  2:    while (...){ ... }
  }
  ...
  3: for (...){
  4:    do {
  5:      while(...) {...}
      } while (...)
  }
}
Loops in LLVM

Function \leftrightarrow \text{Natural loops} \leftrightarrow \text{Merged natural loops (loops with the same header are merged)}
Identify loops in LLVM

• Rely on other passes to identify loops

```cpp
#include "llvm/Analysis/LoopInfo.h"

void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<LoopInfoWrapperPass>();
    AU.setPreservesAll();
}
```

• Fetch the result of the LoopInfoWrapperPass analysis

```cpp```
LoopInfo & LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
```cpp```

• Iterate over outermost loops

```cpp```
for (auto i : LI) {
    Loop *loop = &*i;
    ...
}
```cpp```
Loops in LLVM: sub-loops

- Iterate over sub-loops of a loop

```cpp
void myFunction (){
  1: while (...){
     2:    while (...){ ... }
   }
    ...
  3: for (...){
     4:    do {
         5:      while(...) {...}
     } while (...)
   }
}
Defining loops in graphic-theoretic terms

Is it good? Bad? Implications?

L1: ...
   if (X < 10) goto L2;
goto L1;
L2: ...

if (...) goto L1;
...
do {
   ...
   L1: ...
} while (X < 10);

The good

The bad

Implications?
Outline

• Loops

• Identify loops

• Induction variables

• Loop normalization
Code example

```c
int myF (int k) {
    int i;
    int s = 0;
    for (i = 0; i < 100; i++) {
        s = s + k;
    }
    return s;
}
```

Is adding “k” to “s” for every loop iteration really needed?
Code example

```c
int myF (int k){
    int i;
    int s = 0;
    for (i=0; i < 100; i++){
        s = s + k;
    }
    return s;
}
```

Value of \( k \)
- 0
- \( k \)
- 2\( k \)
- 3\( k \)
- 4\( k \)
- ...
- \( 100k \)
int myF (int k) {
    int i;
    int s = 0;
    s = k * 100;
    return s;
}

Code example

```c
int myF (int k){
    int i;
    int s = 0;
    for (i=0; i < 100; i++){
        s = s + k;
    }
    return s;
}
```

```c
int myF (int k){
    int i;
    int s = 0;
    s = k * 100;
    return s;
}
```
int myF (int k) {
    int i;
    int s = 5;
    for (i=0; i < 100; i++) {
        s = s + k;
    }
    return s;
}
int myF (int k){
    int i;
    int s = 5;
    for (i=0; i < 100; i++){
        s = s + k;
    }
    return s;
}
Code example 2

```c
int myF (int k){
    int i;
    int s ;
    s = k * 100;
    s = s + 5;
    return s;
}
```
int myF (int k) {
    int i;
    int s = 5;
    for (i=0; i < 100; i++){
        s = s + k;
    }
    return s;
}

int myF (int k) {
    int i;
    int s;
    s = k * 100;
    s = s + 5;
    return s;
}
int myF (int k, int iters){
    int i;
    int s = 5;
    for (i=0; i < iters; i++){
        s = s + k;
    }
    return s;
}
Code example 3

```c
int myF (int k, int iters){
    int i;
    int i;
    int s ;
    s = k * iters;
    s = s + 5;

    return s;
}
```
Code example 3

```c
int myF (int k, int iters){
    int i;
    int s = 5;
    for (i=0; i < iters; i++){
        s = s + k;
    }
    return s;
}
```

```c
int myF (...){
    int i;
    int s;
    s = k * iters;
    s = s + 5;
    return s;
}
```
Important information about variable evolution

```c
int myF (int k){
    int i;
    int s = 0;
    for (i=0; i < 100; i++){
        s = s + k;
    }
    return s;
}
```

```c
int myF (int k){
    int i;
    int s = 5;
    for (i=0; i < 100; i++){
        s = s + k;
    }
    return s;
}
```

```c
int myF (int k, int iters){
    int i;
    int s = 5;
    for (i=0; i < iters; i++){
        s = s + k;
    }
    return s;
}
```
• It is important to understand the evolution of variables
• Important transformations are possible only when variable evolutions are analyzed
• Variables with a specific type of evolution (described next) are called “induction variables”
  • “s” was an induction variable in all prior examples
Induction variable observation

• **Observation:**
  Some variables change by a constant amount on each loop iteration
  - x initialized at 0; increments by 1
  - y initialized at N; increments by 2
  - These are all induction variables

• **Definition of induction variable (IV):**
  An IV is a variable that
  - increases or decreases by a fixed amount on every iteration of a loop or
  - it is a linear function of another IV

• How can we identify IVs automatically?

```c
x = 0; y = N;
While (...){
  x++;  
  x = y + 2;
}
```
Identify induction variables

**Idea**

We find induction variables incrementally.

First: we identify the basic cases.

Second: we identify the complex cases.

Iterate the analysis until we cannot add new IVs
Induction variables

• Basic induction variables
  • $i = i \text{ op } c$
  • $c$ is loop invariant
  • a.k.a. independent induction variable

• Derived induction variables

What is a loop-invariant?
Loop-invariant computations

• Let $d$ be the following definition
  
  $t = x$

• $d$ is a loop-invariant of a loop $L$ if
  (assuming $x$ does not escape)
  • $x$ is constant or
  • All reaching definitions of $x$ are outside the loop, or
  • Only one definition of $x$ reaches $d$,
    and that definition is loop-invariant
Loop-invariant computations

• Let \( d \) be the following definition
  \[
  (d) \ t = x \; \text{op} \; y
  \]

• \( d \) is a loop-invariant of a loop \( L \) if
  (assuming \( x, y \) do not escape)
  • \( x \) and \( y \) are constants or
  • All reaching definitions of \( x \) and \( y \) are outside the loop, or
  • Only one definition of \( x \) (or \( y \)) reaches \( d \), and that definition is loop-invariant
Loop-invariant computations

• Let \( d \) be the following definition
  \[
  (d) \ t = \text{load}(x)
  \]

• \( d \) is a loop-invariant of a loop \( L \) if
  (assuming \( x \) does not escape)
  • The memory location pointed by \( x \), \( \text{mem}[x] \), is constant or
  • All reaching definitions of \( \text{mem}[x] \) are outside the loop, or
  • Only one definition of \( \text{mem}[x] \) reaches \( d \),
    and that definition is loop-invariant
Loop example

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

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

\textit{d} is a loop-invariant of a loop \( L \) if
\begin{itemize}
  \item \( x \) and \( y \) are constants or
  \item All reaching definitions of \( x \) and \( y \) are outside the loop, or
  \item Only one definition reaches \( x \) (or \( y \)), and that definition is loop-invariant
\end{itemize}
Loop-invariant computations in LLVM

```c++
for (auto bbi = loop->block_begin(); bbi != loop->block_end(); ++bbi) {
    BasicBlock *bb = bbi;
    for (auto &instr_iter : *bb) {
        auto instr = &instr_iter;
    }
}
```
Induction variables

• Basic induction variables
  • $i = i \text{ op } c$
  • $c$ is loop invariant
  • this definition is executed exactly once per iteration
  • a.k.a. independent induction variable

• Derived induction variables
  • $j = i * c_1 + c_2$
  • $c_1$ and $c_2$ are loop invariants
  • this definition is executed exactly once per iteration
  • $i$ is an IV
  • a.k.a. dependent induction variable
Identify induction variables: step 1

**Find the basic IVs**

① Scan loop body for defs of the form

\[ x = x + c \]

where \( c \) is loop-invariant and this definition is executed exactly once per iteration

② Record these basic IVs as

\[ x = (x, 1, c) \]

this represents the IV: \( x = x \times 1 + c \)

How can we do?
Can we exploit SSA?
Identify induction variables: step 2

Find derived IVs

① Scan for derived IVs of the form
   \[ k = i \times c1 + c2 \]
   where \( i \) is a basic IV and
   this is the only definition of \( k \) in the loop and
   this definition is executed exactly once per iteration

② Record as \( k = (i, c1, c2) \)
   We say \( k \) is in the family of \( i \)
**Code example**

```c
int myF1 (int start, int end){
    int i = start;
    while (i < end){
        j = i * 8 + 4;
        i++;
    }
    return j;
}

int myF2 (int start, int end){
    int i = start;
    while (i < end){
        j = i * 8;
        while (j > 0){
            k = j * 42 + i;
            j--;
        }
        i++;
    }
    return j;
}
```
Identified induction variables

A forest of induction variables
Induction variables in LLVM

• scalar-evolution:
  • Scalar evolution analysis
  • Represent scalar expressions (e.g., $x = y \text{ op } z$)
    • It supports induction variables (e.g., $x = x + 1$)
  • It lowers the burden of explicitly handling the composition of expressions
Induction variable vs. scalar evolution

• Basic IV (BIV):
  It increases or decreases by a fixed amount on every iteration of a loop

• IV:
  A BIV or a linear function of another IV

• Generalized IV (GIV):
  It increases or decreases by an amount
  It can depend non linearly on other BIVs/GIVs
  It can have multiple update
Chain of recurrences

It is a formalism to analyse expressions in BIV and GIV expressing them as Recurrences

\[ n! = 1 \times 2 \times \ldots \times n \quad \leftrightarrow \quad n! = (n-1)! \times n \]

\[ f(n) = 1 \times 2 \times \ldots \times n \quad \leftrightarrow \quad f(n) = f(n-1) \times n \]
Basic recurrences

```java
int f = k0;
for (int j=0; j < n ; j++){
    ... = f;
    f = f + k1
}
```

Assuming k0 and k1 to be loop invariants

\[
f(i) = \begin{cases} 
  k0 & \text{if } i = 0 \\ 
  f(i-1) + k1 & \text{if } i > 0 
\end{cases}
\]

Basic recurrence = \{k0, +, k1\}

Starts with k0, and it increments by k1 every time
Chain of recurrences

```java
int f = g = k0;
for (int j=0; j < n ; j++){
    ... = f;
    g = g + f;
    f = f + k1
}

f(i) = \begin{cases} 
  k0 & \text{if } i == 0 \\
  f(i-1) + k1 & \text{if } i > 0 
\end{cases}

Basic recurrence = \{k0, +, k1\}

Chain of recurrence = \{k0, +, \{k0, +, k1\}\} = \{k0, +, k0, +, k1\}
```

```java
g(i) = \begin{cases} 
  k0 & \text{if } i == 0 \\
  g(i-1)+f(i-1) & \text{if } i > 0 
\end{cases}
```

```java
```
Chain of recurrences

for (int x=0; x < n ; x++){
    p[x] = x*x*x + 2*x*x + 3*x + 7;
}

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>7</td>
<td>13</td>
<td>29</td>
<td>61</td>
<td>115</td>
<td>197</td>
</tr>
</tbody>
</table>
Chain of recurrences

for (int x=0; x < n; x++){
    p[x] = x*x*x + 2*x*x + 3*x + 7;
}

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[x]</td>
<td>7</td>
<td>13</td>
<td>29</td>
<td>61</td>
<td>115</td>
<td>197</td>
</tr>
</tbody>
</table>
chain of recurrences

for (int x=0; x < n ; x++){
    p[x] = x*x*x + 2*x*x + 3*x + 7;
}

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[x]</td>
<td>7</td>
<td>13</td>
<td>29</td>
<td>61</td>
<td>115</td>
<td>197</td>
</tr>
<tr>
<td>D</td>
<td>-</td>
<td>6</td>
<td>16</td>
<td>32</td>
<td>54</td>
<td>82</td>
</tr>
</tbody>
</table>
Chain of recurrences

```java
for (int x=0; x < n ; x++) {
    p[x] = x*x*x + 2*x*x + 3*x + 7;
}
```

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[x]</td>
<td>7</td>
<td>13</td>
<td>29</td>
<td>61</td>
<td>115</td>
<td>197</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>6</td>
<td>16</td>
<td>32</td>
<td>54</td>
<td>82</td>
</tr>
<tr>
<td>D^2</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>16</td>
<td>22</td>
<td>28</td>
</tr>
</tbody>
</table>
Chain of recurrences

for (int x=0; x < n ; x++){
    p[x] = x*x*x + 2*x*x + 3*x + 7;
}

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[x]</td>
<td>7</td>
<td>13</td>
<td>29</td>
<td>61</td>
<td>115</td>
<td>197</td>
</tr>
<tr>
<td>D</td>
<td>-</td>
<td>6</td>
<td>16</td>
<td>32</td>
<td>54</td>
<td>82</td>
</tr>
<tr>
<td>D²</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>16</td>
<td>22</td>
<td>28</td>
</tr>
<tr>
<td>D³</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Chain of recurrence = \{7, +, 6, +, 10, +, 6\}
Chain of recurrences

Chain of recurrence = \{7, +, 6, +, 10, +, 6\}

```
3 void myF (int *p, int n){
4   for (int x=0; x < n; x++){
5     p[x] = x*x*x + 2*x*x + 3*x + 7;
6   }
7 }
```

And if you run scalar evolution of LLVM:
Instruction  \%16 = add nsw i32 \%15, 7 is SCEVAddRecExpr
SCE: \{7,+,6,+,10,+,6\}<%7>
LLVM scalar evolution example

• SCEV: \{A, B, C\}<\text{flag}>*<\%D>
  • A: Initial; B: Operator; C: Operand; D: basic block where it get defined
LLVM scalar evolution example

• SCEV: \{A, B, C\}<flag>*<%D>
  • A: Initial; B: Operator; C: Operand; D: basic block where it get defined

```
#include <stdio.h>
int main (int argc, char *argv[]){
  for (int i=0; i < argc; i++){
    printf("ciao\n");
  }
  return 0;
}
```

```
cat-c program.bc -c -emit-llvm -o loop_0.bc
Function: main
New loop
Instruction  %i.0 = phi i32 [ 0, %0 ], [ %6, %5 ] is SCEVAddRecExpr
  SCE: {0,+1}<nuw><nsw><%1>
Instruction  %6 = add nsw i32 %i.0, 1 is SCEVAddRecExpr
  SCE: {1,+1}<nuw><nsw><%1>
```
void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<LoopInfoWrapperPass>();
    AU.addRequired<ScalarEvolutionWrapperPass>();
    AU.setPreservesAll();
}
bool runOnFunction(Function &F) override {
    LoopInfo & LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
    ScalarEvolution *SE = &getAnalysis<ScalarEvolutionWrapperPass>().getSE();

    errs() << "Function: " << F.getName() << "\n";
    for (auto i = LI.begin(); i != LI.end(); ++i) {
        Loop *loop = *i;
        errs() << " New loop\n";
        for (auto bbi = loop->block_begin(); bbi != loop->block_end(); ++bbi) {
            BasicBlock *bb = bbi;
            for (BasicBlock::iterator i = bb->begin(); i != bb->end(); ++i) {
                Instruction *inst = &*i;
                const SCEV *S = SE->getSCEV(inst);
                if (auto *AR = dyn_cast<SCEVAddRecExpr>(S)) {
                    errs() << " Instruction ";
                    i->print(errs());
                    errs() << " is SCEVAddRecExpr\n";

                    errs() << " SCE: ";
                    AR->print(errs());
                    errs() << "\n";
                }
            }
        }
    }
    return false;
}
Scalar evolution in LLVM

• Analysis used by
  • Induction variable substitution
  • Strength reduction
  • Vectorization
  • …

• SCEVs are modeled by the `llvm::SCEV` class
  • There is a sub-class for each kind of SCEV (e.g., `llvm::SCEVAddExpr`)

• A SCEV is a tree of SCEVs
  • Leaves:
    • Constant: `llvm::SCEVConstant` (e.g., 1)
    • Unknown: `llvm::SCEVUnknown` (e.g., `%v = call rand()`)
  • To iterate over a tree: `llvm::SCEVVisitor`
Outline

• Loops

• Identify loops

• Induction variables

• Loop normalization
```c
#include <stdio.h>

int main (){
    for (int i=0; i < 10; i++){
        printf("Hello world\n");
    }
    return 0;
}
```

Code before a new iteration
We need to normalize loops so CATs can expect a single pre-defined shape!

```c
#include <stdio.h>

int main (){
    i=0;
    do {
        printf("Hello world!
        i++;
    } while (i < 10);
    return 0;
}
```
First normalization: adding a pre-header

- Optimizations often require code to be executed once before the loop
- Create a pre-header basic block for every loop
Common loop normalization

Pre-header

Header

Body

exit

Pre-header

Header

Body

exit
Common loop normalization

Diagram:

- Pre-header
- Header
- Body
- Exit

Diagram:

- Pre-header
- Header
- Body
- Exit
Loop normalization in LLVM

• The loop-simplify pass normalize natural loops
• Output of loop-simplify:
  • **Pre-header**: the only predecessor of the header
Loop normalization in LLVM

- The loop-simplify pass normalize natural loops
- Output of loop-simplify:
  - **Pre-header**: the only predecessor of the header
  - **Latch**: node executed just before starting a new loop iteration
Loop normalization in LLVM

• The loop-simplify pass normalize natural loops
• Output of loop-simplify:
  • **Pre-header**: the only predecessor of the header
  • **Latch**: single node executed just before starting a new loop iteration
  • **Exit node**: ensures it is dominated by the header
Loop normalization in LLVM

- The loop-simplify pass normalize natural loops
- Output of loop-simplify:
  - **Pre-header**: the only predecessor of the header
  - **Latch**: single node executed just before starting a new loop iteration
  - **Exit node**: ensures it is dominated by the header
Definition:
A **critical edge** is an edge in the CFG which is neither the only edge leaving its source block, nor the only edge entering its destination block.

These edges must be split: a new block must be created and inserted in the middle of the edge, to insert computations on the edge without affecting any other edges.
Loop normalization in LLVM

- **Pre-header**: `llvm::Loop::getLoopPreheader()`
- **Header**: `llvm::Loop::getHeader()`
- **Latch**: `llvm::Loop::getLoopLatch()`
- **Exit**: `llvm::Loop::getExitBlocks()`

```
opt -loop-simplify bitcode.bc -o normalized.bc
```
Further normalizations in LLVM

• Loop representation can be further normalized:
  • `loop-simplify` normalize the shape of the loop
  • What about definitions in a loop?

• Problem: updating code in loop might require to update code outside loops for keeping SSA
Loop pass example

A pass needs to add a conditional definition of \( d \)
This is not in SSA anymore: we must fix it
Further normalizations in LLVM

- Loop representation can be further normalized:
  - *loop-simplify* normalize the shape of the loop
  - What about definitions in a loop?

- Problem: updating code in loop might require to update code outside loops for keeping SSA
  - Keeping SSA form is expensive with loops
  - Loop-closed SSA form: no var is used outside of the loop in that it is defined
  - lcssa insert phi instruction at loop boundaries for variables defined in a loop body and used outside
  - Isolation between optimization performed in and out the loop
  - Faster keeping the SSA form
    - Propagation of code changes outside the loop blocked by phi instructions
Loop pass example

```
while (){
    d = ...
}
...
... = d op ...
... = d op ...
call f(d)

Lcssa normalization

while (){
    d = ...
}
    d1 = phi(d...)
... = d1 op ...
... = d1 op ...
call f(d1)

while (){
    d = ...
    ...
    if (...){
        d2 = ...
    }
    d3 = phi(d,d2)
    d1 = phi(d...)
... = d1 op ...
... = d1 op ...
call f(d1)

while (){
    d = ...
    ...
    if (...){
        d2 = ...
    }
    d3 = phi(d,d2)
    d1 = phi(d3...)
... = d1 op ...
... = d1 op ...
call f(d1)
```
Loop-closed SSA form in LLVM

```
opt -lcssa bitcode.bc -o transformed.bc

LLVM::Loop::isLCSSAForm(DT)

formLCSSA(...)```

Further normalizations in LLVM

Last loop-related normalization:
Scalar evolution normalization