Loops

Code analysis and transformation

Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• Loops

• Identify loops

• 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

Cold code

Loop
Hot code

Identify hot code to succeed!!!
Loops ...

... but where are they?

... How can we find them?
Loops in source code

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

• 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

\[
s = \text{new Stack()};\ s.\text{add}(G.\text{entry});\ \text{mark}(G.\text{entry});
\]
\[
\text{While (!s.\text{empty}())}{
1: \ v = s.\text{pop}();
2: \ \text{if (v' = adjacentNotMarked(v, G))}{
3: \ \text{mark}(v');\ \text{DFST.add}((v, v'));
4: \ s.\text{push}(v');
}\}
\]
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
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);
    }
}
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?
while (a: i < 10){
  b: if (i == 5) continue;
  c: ...
}
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 (...)
         
    }
}
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();
```

• Iterate over **outermost** loops

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

```cpp
void myFunction (){
1: while (...){
    2: while (...){ ... }
    }
    ...
3: for (...){
    4:  do {
        5:  while(...) {...}
            } while (...)
        }
    }
}
```
Loops in LLVM: sub-loops

• Iterate over sub-loops of a loop

```cpp
vector<Loop *> subLoops = loop->getSubLoops();
for (auto j : subLoops)
    Loop *subloop = &*j;
...
```

```cpp
void myFunction (){  
1: while (...){ 
2:    while (...}{...} 
3:    } 
4:    } 
5:    while (...) {...} 
6:    } 
7: }
```
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);
Outline

• Loops

• Identify loops

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

int main (){
    for (int i=0; i < 10; i++){
        printf("Hello world\n");
    }
    return 0;
}
```
We need to normalize loops so CATs can expect a single pre-defined shape!
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
Common loop normalization
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**: 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**: node executed just before starting a new loop iteration
  • **Exit node**: ensures it is dominated by the header
(Critical edges)

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 SSA add a conditional definition of d

Changes to code outside our loop
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-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: Induction variable normalization