Advanced Topics in Compilers

Loops

Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• Loops in LLVM

• Loops in NOELLE

• Loop abstractions in NOELLE
• Target optimization: we need to identify loops
• There is no IR instruction for “loop”
• How to identify an IR loop?

```c
#include <stdio.h>

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

• 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.
void myFunction (){
  1: while (...){
  2:    while (...){ ... }
  3: for (...){
  4:      do {
  5:        while(...) {...}
          } while (...)
  } while (...)
}
Loops in LLVM

Function $\iff$ Natural loops $\iff$ Merged natural loops (loops with the same header are merged)
First loop 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

Diagram:
- Pre-header
- Header
- Body
- Exit

Structure:
- Pre-header
- Header
- Body
- Exit
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**: 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
Further normalizations in LLVM

• Loop representation can be further normalized:
  • \textit{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
  • \texttt{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
Loop pass example

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

## Lcssa normalization

while (){
    d = ...
}

... = phi(d...)

... = d1 op ...
... = d1 op ...
call f(d1)

while (){
    d = ...
    ...
    ...
    ...
    ...

    if (...){
        d2 = ...
    }

    d3 = phi(d,d2)

    d1 = phi(d...)
    ...
    ...
    ...
    ...

    = d1 op ...
    ...
    call f(d1)

while (){
    d = ...
    ...
    ...
    ...
    ...
    ...

    if (...){
        d2 = ...
    }

    d3 = phi(d,d2)

    d1 = phi(d3...)
    ...
    ...
    ...
    ...

    = d1 op ...
call f(d1)
Outline

• Loops in LLVM

• Loops in NOELLE

• Loop abstractions in NOELLE
NOELLE

• All loops in NOELLE are normalized as canonical and in LCSSA form all the time

• Before invoking NOELLE to any IR file, you must normalize that IR
  • noelle-simplification:
    normalizations required by NOELLE

  • noelle-norm:
    normalizations required by NOELLE + fast optimizations that are needed most of the time (e.g., dead code elimination)
Normalizations in NOELLE

noelle-norm MYIR.bc –o IR.bc
noelle-load –load ~/CAT/lib/MYPASS.so –MYPASS IR.bc –o newIR.bc

noelle-simplification MYIR.bc –o IR.bc
noelle-load –load ~/CAT/lib/MYPASS.so –MYPASS IR.bc –o newIR.bc
Get loops of a function with NOELLE

/*
 * Fetch the entry point.
 */
auto fm = noelle.getFunctionsManager();
auto mainF = fm->getEntryFunction();

/*
 * Fetch the loops with only the loop structure abstraction.
 */
auto loopStructures = noelle.getLoopStructures(mainF);

Each loop is an instance of LoopStructure
Flat representation of the loops
Loop forest with NOELLE

/*
 * Fetch the entry point.
 */
auto fm = noelle.getFunctionsManager();
auto mainF = fm->getEntryFunction();

/*
 * Fetch the loops with only the loop structure abstraction.
 */
auto loopStructures = noelle.getLoopStructures(mainF);

/*
 * Fetch the loop forest.
 */
auto loopForest = noelle.organizeLoopsInTheirNestingForest(*loopStructures);
Traversing loop forest with NOELLE

/*
 * Iterate over the trees that compose the forest.
 */
errs() << "Printing the loop forest\n";
for (auto loopTree : loopForest->getTrees()){

    /*
     * Fetch the root of the current tree.
     */
    auto rootLoop = loopTree->getLoop();
    errs() << "======= Tree with root " << *rootLoop->getEntryInstruction() << "\n";
    printTree(loopTree);
    errs() << "\n";
}

Traversing loop forest with NOELLE
Get all program loops with NOELLE

```cpp
/*
 * Fetch the loops with only the loop structure abstraction.
 */
auto loopStructures = noelle.getLoopStructures();
```

```cpp
/*
 * Fetch the loops with only the loop structure abstraction.
 */
auto loopStructures = noelle.getLoopStructures(mainF);
Outline

• Loops in LLVM

• Loops in NOELLE

• Loop abstractions in NOELLE
Loop abstractions in NOELLE

- We saw one abstraction so far: LoopStructure

- LoopStructure describes structural aspects of a loop
  - Entry point
  - Exit points
  - Set of basic blocks that compose the loop
  - Latches
  - Pre-header
  - Nesting level

- LoopStructure is a little more than LLVM’s Loop
LoopDependenceInfo

• In NOELLE:
  LoopStructure is the simplest abstraction that describes a loop

```cpp
/*
 * Fetch the loops with only the loop structure abstraction.
 */
auto loopStructures = noelle.getLoopStructures();
```

• In NOELLE:
  LoopDependenceInfo is the abstraction that describes a loop with the most information

```cpp
/*
 * Fetch the loops with all their abstractions
 * (e.g., Loop Dependence Graph, SCCDAG)
 */
auto loops = noelle.getLoops();
```
From LoopStructure to LoopDependenceInfo

/*
 * Iterate over all loops,
 * and compute the LoopDependenceInfo only for those that we care.
 */
for (auto l : *loopStructures){
    if (l->getNestingLevel() > 1){
        continue;
    }

    /*
    * Get the LoopDependenceInfo
    */
    auto ldi = noelle.getLoop(l);
}
LoopDependenceInfo includes many abstractions

- LoopStructure

```c
/*
 * Print the first instruction the loop executes.
 */
auto LS = loop->getLoopStructure();
auto entryInst = LS->getEntryInstruction();
errs() << "Loop " << *entryInst << "\n";
```

- InductionVariableManager

```c
/*
 * Induction variables.
 */
errs() << " Induction variables\n";
auto IVM = loop->getInductionVariableManager();
```
LoopDependenceInfo includes many abstractions

- InvariantManager
  ```cpp
  /*
   * Invariants.
   */
  errs() << " Invariants\n";
  auto IM = loop->getInvariantManager();
  ```

- Loop dependence Graph
  ```cpp
  /*
   * Dependences.
   */
  auto LDG = loop->getLoopDG();
  ```
LoopDependenceInfo includes many abstractions

- SCCManager

```c
/*
 * Dependences.
 */
auto sccManager = loop->getSCCManager();
auto SCCDAG = sccManager->getSCCDAG();
```

- Object cloner ...
SCCManager

- Get the SCCDAG

```c++
/*
 * Dependences.
 */
auto sccManager = loop->getSCCManager();
auto SCCDAG = sccManager->getSCCDAG();
```

- Iterate over SCCs

```c++
SCCDAG->iterateOverSCCs(sccIterator);
```

```c++
auto sccIterator = [sccManager](SCC *scc) -> bool {
    return false;
};
```
SCC

/*
 * Check if @scc is a single instruction
 */
if (!scc->hasCycle()){
    return false;
}

/*
 * Print the instructions that compose the SCC.
 */
errs() << "    Instructions:\n";
auto mySCCIter = [] (Instruction *i) -> bool {
    errs() << "      " << *i << "\n";
    return false;
};
scc->iterateOverInstructions(mySCCIter);
SCCManager API

/*
 * Fetch the SCC information.
 */
auto sccInfo = sccManager->getSCCArrts(scc);
if (sccInfo->isInductionVariableSCC()){
    errs() << " It is due to the computation of an induction variable\n";
}
else if (sccInfo->canExecuteReducibly()){
    errs() << " It can be reduced\n";
}
else if (sccInfo->canExecuteIndependently()){
    errs() << " It doesn't have loop-carried data dependences\n";
}
else if (sccInfo->mustExecuteSequentially()){
    errs() << " It must be executed sequentially\n";
}
else {
    errs() << " It can run in parallel\n";
}