Advanced Topics in Compilers

Dependences

Simone Campanoni
simone.campanoni@northwestern.edu
Outline

• Program Dependence Graph at the instruction granularity

• SCCDAG

• Semantics of dependences
PDG* is provided by NOELLE

```cpp
/*
 * Fetch the PDG
 */
auto PDG = noelle.getProgramDependenceGraph();
```

This PDG is at the instruction granularity

• A dependence is either
  • Between two instructions or
  • Between an instruction and a function parameter

[⁎] Jeanne Ferrante, Karl J. Ottenstein, Joe D. Warren.
The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and System 1987
NOELLE’s PDG at the instruction granularity

• Dependences are clustered by function
• Dependences between instructions in two functions:

```c
declare void @f1 (int8 *%0){
  ...
  store 4, %0
  call @f2(%0)
  ..
}
```

```c
declare void @f2 (int8 *%0){
  ...
  %a = load %0
  ..
}
```
NOELLE’s Function Dependence Graph (FDG)

```cpp
/*
 * Fetch the PDG
 */
auto PDG = noelle.getProgramDependenceGraph();

/*
 * Fetch the FDG of "main"
 */
auto fm = noelle.getFunctionsManager();
auto mainF = fm->getEntryFunction();
auto FDG = noelle.getFunctionDependenceGraph(mainF);
```

Different instances of the same C++ class (PDG)
PDG: iterating over dependences

* Iterate over the dependences

```c
auto iterF = [] (Value *src, DGEEdge<Value> *dep) -> bool {
  6 lines: errs() << " " << *src << " " ;------------------------
  return false;
};
```

Do you want to stop iterating?

```c
for (auto& inst : instructions(mainF)){
  errs() << "Instruction "" << inst << ":\" depends on\n";
  FDG->iterateOverDependencesTo(&inst, true, true, true, iterF);
}
```
PDG: iterating over dependences

```c
/*
 * Iterate over the dependences
 */

auto iterF = [] (Value *src, DGE<Value> *dep) -> bool {
  6 lines: errs() << " " << *src << " " ;
      return false;
};
```

```c
for (auto& inst : instructions(mainF)){
  errs() << "Instruction " << inst << " depends on\n"
  FDG->iterateOverDependencesTo(&inst, true, true, true, iterF);
}
```
PDG: iterating over dependences

Iterating over outgoing edges
PDG: iterating over dependences

```c
for (auto& inst : instructions(mainF))
    for (auto& inst2 : instructions(mainF))
        for (auto dep : FDG->getDependences(&inst, &inst2))
            if (dep->isControlDependence())
                errs() << " CONTROL ";
        if (dep->isDataDependence())
            if (dep->isRAWDependence())
                errs() << " RAW " ;
            if (dep->isWARDDependence())
                errs() << " WAR " ;
            if (dep->isWAWDependence())
                errs() << " WAW " ;
        if (dep->isMemoryDependence()) {
            if (dep->isMustDependence())
                errs() << " must ";
            else {
                errs() << " may ";
            }
        }
    errs() << " MEMORY ";
```
NOELLE provides SCCDAG

• NOELLE provides:
  • Program Dependence Graph (PDG)
  • Function Dependence Graph (FDG)
  • Loop Dependence Graph (LDG) (see NOELLE_loops slides/talk)

• All dependence graphs are instances of the same class llvm::noelle::PDG

• Because of importance of loops, NOELLE provides a rich class for them called llvm::noelle::LoopDependenceInfo

• LoopDependenceInfo includes:
  • LDG
  • SCCDAG
  • And much more (see NOELLE_loops slides/talk)
Memory alias analysis: the problem (from 323)

• We want to
  • Execute \(j\) in parallel with \(i\) (extracting parallelism)
  • Move \(j\) before \(i\) (code scheduling)

• Does \(j\) depend on \(i\)?

  \[
  \begin{align*}
  i: (*p) &= \text{varA} + 1 \\
  j: \text{varB} &= (*q) \times 2 \\
  \end{align*}
  \]

  \[
  \begin{align*}
  i: \text{obj1.f} &= \text{varA} + 1 \\
  j: \text{varB} &= \text{obj2.f} \times 2 \\
  \end{align*}
  \]

• Do \(p\) and \(q\) point to the same memory location?
  • Does \(q\) alias \(p\)?
Memory alias analyses included in NOELLE

• NOELLE relies on ~40 memory alias analyses to compute its PDG

• Most analyses are included in the following 3 frameworks:
  • SCAF: https://github.com/PrincetonUniversity/SCAF
  • SVF: https://github.com/SVF-tools/SVF
  • LLVM: http://llvm.org

• NOELLE includes an extra alias analysis as well to capture corner cases that alias analyses above do not
  • We see alias analysis to be used by NOELLE, rather than for NOELLE to provide
  • Hence, when another alias infrastructure will capture them, this NOELLE’s AA will be removed
Outline

• Program Dependence Graph at the instruction granularity

• SCCDAG

• Semantics of dependences
NOELLE’s Hierarchical SCCDAG

• From the PDG

• To the SCC identifications
NOELLE’s Hierarchical SCCDAG

• From the PDG

• To the SCC identifications

• To the SCCDAG
NOELLE’s Hierarchical SCCDAG

• From the PDG

• To the SCC identifications

• To the SCCDAG

```c
/*
 * Compute the SCCDAG of the FDG of "main"
 */
auto mainSCCDAG = new SCCDAG(FDG);
```
Outline

• Program Dependence Graph at the instruction granularity

• SCCDAG

• Semantics of dependences
Dependences

• Control dependences
• Data dependences
  • Variable
  • Memory
Dependences

• Control dependences

• Data dependences
  • Variable
  • Memory
Post-Dominators

**Assumption:** Single exit node in CFG

**Definition:** Node $d$ post-dominates node $n$ in a graph if every path from $n$ to the exit node goes through $d$
Control dependences

A node $Y$ control-depends on another node $X$ if and only if

1. There is a path from $X$ to $Y$ such that every node in that path other than $X$ is post-dominated by $Y$

2. $X$ is not strictly post-dominated by $Y$

B: while (par1 > 5)
C: varX = par1 + 1
C2: ...
D: print(varX)
Dependences

• Control dependences

• Data dependences
  • Variable
  • Memory
Data dependences

• A variable dependence is a def-use chain in LLVM
• A memory dependence from instruction $i_1$ to instruction $i_2$ exists iff *:
  • the footprint of operation $i_1$ may-alias the footprint of $i_2$ (alias);
  • at least one of the two instructions writes to memory (update);
  • there is a feasible path of execution $P$ from $i_1$ to $i_2$ (feasible-path) such that no operation in $P$ overwrites the common memory footprint (no-kill).

Footprint refers to the memory locations accessed (read or written) by an instruction.

[*) Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I. August
The (LLVM) memory model

myObject0 = call malloc(4)
myObject1 = call malloc(10)
p = myObject0 + 4

Can p alias myObject1?
The (LLVM) memory model

myObject0 = call malloc(4)
myObject1 = call malloc(10)
p = myObject0 + 4

Can \( p \) alias myObject1?
Always have faith in your ability

Success will come your way eventually

Best of luck!