Cours M2: Compilation avancée et optimisation de programmes

Alain Darte

CNRS, Compsys
Laboratoire de l’Informatique du Parallélisme
École normale supérieure de Lyon

Back-end code optimizations
Outline

1 Code representations
   - Control-flow graph
   - Loop-nesting forest
   - Static single assignment

2 Out-of-SSA translation
   - Translation with copy insertions: pitfalls and solution
   - Improving code quality and ease of implementation
   - Fast implementation with reduced memory footprint

3 SSA properties and liveness
   - Dominance, liveness, interferences, and chordal graphs
   - Construction of liveness sets in reducible CFGs for strict SSA
   - Extensions to irreducible CFGs and for checking liveness
Back-end code analysis

**Control-flow analysis** determines control flow and control structure of a program and build a program representation.

- Basic block
- Control-flow graph
- Loop-nesting forest
- Static single assignment

**Data-flow analysis** determines the flow of scalar variables, their live-ranges, and possibly their values.

- Constant propagation
- Redundancy elimination, dead-code elimination
- Code motion and scheduling
- Register allocation

Analysis: local, intra-procedural, or inter-procedural.
Basic block sequence of consecutive statements in any execution: single entry & single exit.

Control-flow graph directed graph:
- nodes are basic blocks
- edges represent control flow (jumps or fall-through), i.e., paths that may be taken
- block/edge frequencies

Vocabulary
- DFS, back-edge, cross-edge
- loop, entry node, join node
- reducible and irreducible graph
- critical edge (in red)
Dominance relation

- a single entry node \( r \).
- each node reachable from \( r \).
- \( a \) dominates \( b \) if every path from \( r \) to \( b \) contains \( a \).
Dominance, post-dominance, control dependences

**Dominance relation**
- a single entry node \( r \).
- each node reachable from \( r \).
- \( a \) dominates \( b \) if every path from \( r \) to \( b \) contains \( a \).
Dominance, post-dominance, control dependences

**Dominance relation**
- a single entry node \( r \).
- each node reachable from \( r \).
- \( a \) dominates \( b \) if every path from \( r \) to \( b \) contains \( a \).
Dominance, post-dominance, control dependences

Dominance relation

- a single entry node $r$.
- each node reachable from $r$.
- $a$ dominates $b$ if every path from $r$ to $b$ contains $a$. 

2 dominates 4?

Properties

- The dominance relation induces a tree.
- With tree labeling, testing if $a$ dominates $b$ takes $O(1)$.

Alain Darte  
Cours M2: Compilation avancée et optimisation de programmes
Dominance, post-dominance, control dependences

**Dominance relation**

- A single entry node $r$.
- Each node reachable from $r$.
- $a$ dominates $b$ if every path from $r$ to $b$ contains $a$. 

![Dominance relation diagram]

2 dominates 4? NO
Dominance, post-dominance, control dependences

**Dominance relation**
- a single entry node \( r \).
- each node reachable from \( r \).
- \( a \) dominates \( b \) if every path from \( r \) to \( b \) contains \( a \).

**Properties**
- The dominance relation induces a **tree**.
- With tree labeling, testing if \( a \) dominates \( b \) takes \( O(1) \).
Dominance, post-dominance, control dependences

**Dominance relation**
- a single entry node \( r \).
- each node reachable from \( r \).
- \( a \) dominates \( b \) if every path from \( r \) to \( b \) contains \( a \).

**Properties**
- The dominance relation induces a tree.
- With tree labeling, testing if \( a \) dominates \( b \) takes \( O(1) \).

Similar for post-dominance, used for defining control dependences: \( b \) is control-dependent on \( a \) if there is a path from \( a \) to \( b \) and \( b \) does not strictly post-dominate \( a \).
Loop nesting forest

Construction (minimal properties)

- Partition the CFG into its strongly connected components (SCCs). A SCC with at least one edge is called a loop.
Loop nesting forest

Construction (minimal properties)

- Partition the CFG into its strongly connected components (SCCs). A SCC with at least one edge is called a loop.
- For each loop \( L \), select a subset of nodes in \( L \) not dominated by any other node in \( L \): \( \text{loop-headers of } L \). Remove all edges in \( L \) that lead to a loop-header: \( \text{loop-edges of } L \).
Loop nesting forest

Construction (minimal properties)

- Partition the CFG into its strongly connected components (SCCs). A SCC with at least one edge is called a **loop**.
- For each loop \( L \), select a subset of nodes in \( L \) not dominated by any other node in \( L \): **loop-headers** of \( L \). Remove all edges in \( L \) that lead to a loop-header: **loop-edges** of \( L \).
- Repeat this partitioning recursively for every SCC.
**Loop nesting forest**

**Construction (minimal properties)**

- Partition the CFG into its strongly connected components (SCCs). A SCC with at least one edge is called a loop.
- For each loop \( L \), select a subset of nodes in \( L \) not dominated by any other node in \( L \): loop-headers of \( L \). Remove all edges in \( L \) that lead to a loop-header: loop-edges of \( L \).
- Repeat this partitioning recursively for every SCC.

**Corresponding loop-nesting forest**

- Leaves are the nodes of the CFG.
- Internal nodes, labeled by loop-headers, correspond to loops.
- The children of a loop’s node represent all inner loops it contains as well as the regular basic blocks of the loop’s body.
As the CFG is not reducible, several loop forests are possible, with loop headers 5 and/or 6. Also, in general, the depth of a loop forest is not uniquely defined.
Tarjan’s algorithm for detecting loops (reducible case)

procedure collapse(loopBody, loopHeader)
    for every z ∈ loopBody do
        loop-parent(z) := loopHeader; LP.union(z, loopHeader)
    endfor

procedure findloop(potentialHeader)
    loopBody = {}
    worklist = {LP.find(y) | y → potentialHeader is a back-edge} \ {potentialHeader}
    while (worklist is not empty) do
        remove an arbitrary element y from worklist; add y to loopBody
        for every predecessor z of y such that (z, y) is not a back-edge do
            if (LP.find(z) ∉ (loopBody ∪ {potentialHeader} ∪ worklist)) then
                add LP.find(z) to worklist
            endif
        endfor
    endwhile
    if (loopBody is not empty) then collapse(loopBody, potentialHeader)

procedure TarjanAlgorithm(G)
    for every vertex x of G do loop-parent(x) := NULL; LP.add(x); endfor
    for every vertex x of G in reverse-DFS-order do findloop(x); endfor
Ramalingam’s modified Havlak’s algorithm (general case)

```
procedure markIrreducibleLoops(z)
    t := loop-parent(z)
    while (t ≠ NULL) do
        u = RLH.find(t); mark u as irreducible-loop-header
        t := loop-parent(u)
        if (t ≠ NULL) then RLH.union(u, t)
    endwhile

procedure processCrossFwdEdges(x)
    for every edge (y, z) in CrossFwdEdges[x] do
        add edge (find(y), find(z)) to the graph; markIrreducibleLoops(z)
    endfor

procedure ModifiedHavlakAlgorithm(G)
    for every vertex x of G do
        loop-parent(x) := NULL; crossFwdEdges[x] := {}; LP.add(x); RLH.add(x);
    endfor
    for every forward edge and cross edge (y, x) of G do
        remove (y, x) from G and add it to crossFwdEdges[LCA(y, x)]
    endfor
    for every vertex x of G in reverse-DFS-order do
        processCrossFwdEdges(x)
        findloop(x) /* same procedure as for Tarjan’s algorithm */
    endfor
```