COMP 181

Lecture 23

Compiling for modern architectures

November 30, 2006

Prelude

- How many people ride the T?
  - Red line: 210 thousand a day
  - Green line: 208 thousand
  - Orange line: 154 thousand
  - Blue line: 55 thousand
- How many T stops are there in Somerville/Medford?
  - 2 – Davis Square, Wellington
- What is the proposed solution?
  - Green line extension from Lechmere
  - New MBTA map…

Today

- Finish up dataflow analysis
- Discuss issues with modern architectures

Problem 4: Constant Folding

- Compute constant variables at each program point
- Constant variable = variable having a constant value on all program executions
- Dataflow information: sets of constant values
- Example: {x=2, y=3} at program point p
- Is a forward analysis
- Let V = set of all variables in the program, nvar = |V|
- Let N = set of integer numbers
- Use a lattice over the set V x N
- Construct the lattice starting from a lattice for N
- Problem: (N, $\leq$) is not a complete lattice!
  … why?

Constant Folding Lattice

- Second try: lattice ($N \cup \{\top, \bot\}$, $\leq$)
  - Where $\bot \leq n$, for all $n \in N$
  - And $n \leq \top$, for all $n \in N$
  - Is complete!
- Meaning:
  - $v = \top$: don't know if v is constant
  - $v = \bot$: v is not constant

$\top$
…
2
1
0
-1
-2
…
$\bot$

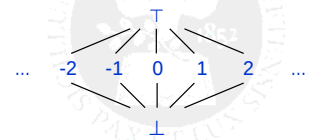
Constant Folding Lattice

- Second try: lattice $(N \cup \{\top, \bot\}, \leq)$
  - Where $\bot \leq n$, for all $n \in N$
  - And $n \leq \top$, for all $n \in N$
  - Is complete!
- Problem:
  - Is incorrect for constant folding
  - Meet of two constants $c \neq d$ is min(c,d)
  - What should meet of different constants be?
- Another problem:
  - has infinite height …

$\top$
…
2
1
0
-1
-2
…
$\bot$

Constant Folding Lattice

- Solution: flat lattice $L = (N \cup \{\top, \bot\}, \sqsubseteq)$
  - Where $\bot \sqsubseteq n$, for all $n \in N$
  - And $n \sqsubseteq \top$, for all $n \in N$
  - And distinct integer constants are not comparable



- Note: meet of any two distinct numbers is $\bot$!

Constant Folding Lattice

- Denote $N^* = N \cup \{\top, \bot\}$
- Use flat lattice $L = (N^*, \sqsubseteq)$
- Constant folding lattice: $L' = (V \rightarrow N^*, \sqsubseteq_C)$
- Where partial order on $V \rightarrow N^*$ is defined as:
  $X \sqsubseteq_C Y$ iff for each variable v: $X(v) \sqsubseteq Y(v)$
- Can represent a function in $V \rightarrow N^*$ as a set of assignments: { {v1=c1}, {v2=c2}, …, {vn=cn} }

CF: Transfer Functions

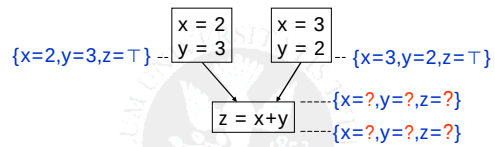
- Transfer function for instruction I:
  $F_I(X) = (X - kill[I]) \cup gen[I]$
  where:
  kill[I] = constants "killed" by I
  gen[I] = constants "generated" by I
- Slightly tricky part: what is $\{v=5\} \cup \{v=\bot\}$?
- If I is v = c (constant): gen[I] = {v=c} kill[I] = {v} x N*
- If I is v = u+w: gen[I] = {v=e} kill[I] = {v} x N*
  where e = X[u] + X[w], if X[u] and X[w] are not $\top, \bot$
  e = $\bot$, if X[u] = $\bot$ or X[w] = $\bot$
  e = $\top$, if X[u] = $\top$ or X[w] = $\top$

CF: Transfer Functions

- Transfer function for instruction I:
  $F_I(X) = (X - kill[I]) \cup gen[I]$
- Here gen[I] is not constant, it depends on X
- However transfer functions are monotonic (easy to prove)
- … but are transfer functions distributive?

CF: Distributivity

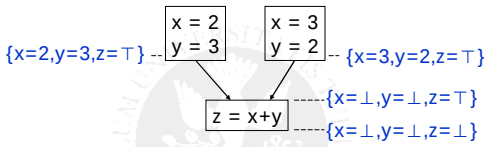
- Example:



- At join point, apply meet operator
- Then use transfer function for z=x+y

CF: Distributivity

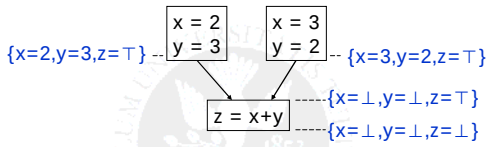
- Example:



- Dataflow result (MFP) at the end: {x=⊥,y=⊥,z=⊥}
- MOP solution at the end?

CF: Distributivity

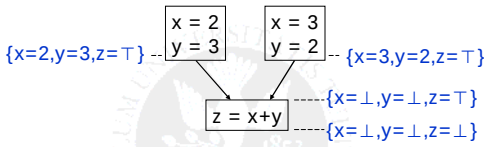
- Example:



- Dataflow result (MFP) at the end: {x=⊥,y=⊥,z=⊥}
- MOP solution at the end: {x=⊥,y=⊥,z=5} !

CF: Distributivity

- Example:



- Reason for MOP ≠ MFP:
  transfer function F of z=x+y is not distributive!

  $F(X1 \sqcap X2) \neq F(X1) \sqcap F(X2)$

  where X1 = {x=2,y=3,z=⊤} and X2 = {x=3,y=2,z=⊤}

Classification of Analyses

- Forward analyses: information flows from
  - CFG entry block to CFG exit block
  - Input of each block to its output
  - Output of each block to input of its successor blocks
  - Examples: available expressions, reaching definitions, constant folding
- Backward analyses: information flows from
  - CFG exit block to entry block
  - Output of each block to its input
  - Input of each block to output of its predecessor blocks
  - Example: live variable analysis

Another Classification

- "may" analyses:
  - information describes a property that MAY hold in SOME executions of the program
  - Usually: ⊓=∪, ⊤=∅
  - Hence, initialize info to empty sets
  - Examples: live variable analysis, reaching definitions
- "must" analyses:
  - information describes a property that MUST hold in ALL executions of the program
  - Usually:⊓=∩, ⊤=S
  - Hence, initialize info to the whole set
  - Examples: available expressions

Next topic

Compiling for modern architectures

- "Old" architectures
  - Single-issue, in-order, no speculation, etc.
  - Compiler: just generate shortest sequence of instructions
- Modern architectures
  - Performance is exposed
  - Compiler needs to take many issues into account

Main Problems

Issues:

- Pipelined machines: scheduling to expose instructions which can run in parallel in the pipeline, whithout stalls
- Superscalar, VLIW: scheduling to expose instruction which can run fully in parallel
- Symmetric multiprocessors (SMP): transformations to expose coarse-grain parallelism
- Memory hierarchies: transformations to improve memory system performance
- Need knowledge about dependencies between instructions

Book: "Optimizing Compilers for Modern Architectures", by Kennedy, Allen

Pipelined Machines

- Example pipeline:
  - Fetch
  - Decode
  - Execute
  - Memory access
  - Write back

Fetch	Dec	Exe	Mem	WB

- Simultaneously execute stages of different instructions

Instr 1	Fetch	Dec	Exe	Mem	WB		
Instr 2		Fetch	Dec	Exe	Mem	WB	
Instr 3			Fetch	Dec	Exe	Mem	WB

Stall the Pipeline

- It is not always possible to pipeline instructions
- Example 1: branch instructions

Branch	Fetch	Dec	Exe	Mem	WB			
Target				Fetch	Dec	Exe	Mem	WB

- Example 2: load instructions

Load	Fetch	Dec	Exe	Mem	WB		
Use			Fetch	Dec	Exe	Mem	WB

Pipelined Machines

- Instructions cannot be executed concurrently in the pipeline because of ***hazards***:
  - Control hazard: target of branch not known in the early stages of the pipeline, cannot fetch next instruction
  - Data hazard: results of an instruction not available for a subsequent instruction
  - Structural hazard: machine resources restrict the number of possible combinations of instructions in the pipeline
- Hazards produce pipeline stalls
- Instruction scheduling = reorder instructions to avoid hazards

Instruction Scheduling

- Instruction scheduling = reorder instructions to improve the parallel execution of instructions
- Essentially, compiler detects parallelism in the code
- Instruction Level Parallelism (ILP) = parallelism between individual instructions
  - Instruction scheduling: reorder instructions to expose ILP

Instruction Scheduling

- Many techniques for instruction scheduling
- List scheduling
  - Build dependence graph
  - Schedule an instruction if all its predecessors have been scheduled
  - Many choices at each step: need heuristics
- Scheduling across basic blocks
  - Move instructions past control flow split/join points
  - Move instruction to successor blocks
  - Move instructions to predecessor blocks

Superscalar, VLIW

- Processor can issue multiple instructions in each cycle
- Need to determine instructions which don't depend on each other
  - VLIW: programmer/compiler finds independent instructions
  - Superscalar: hardware detects if instructions are independent; but compiler must maximize independent instructions close to each other
- Out-of-order superscalar: burden of instruction scheduling and ILP detection is partially moved to the hardware
- Must detect and reorder instructions to expose fully independent instructions

Symmetric Multiprocessors

- Multiple processing units (as in VLIW)
- …which execute asynchronously (unlike VLIW)
- Problems:
  - Overhead of creating and starting threads of execution
  - Overhead of synchronizing threads
- Conclusion:
  - Inefficient to execute single instructions in parallel
  - Need coarse grain parallelism (not ILP)
  - Compiler must detect larger pieces of code (not just instructions) which are independent

Memory Hierarchies

- Memory system is hierarchically structured: register, L1 cache, L2 cache, RAM, disk
- Top the hierarchy: faster, but fewer
- Bottom of the hierarchy: more resources, but slower
- Memory wall problem: processor speed increases at a higher rate than memory latency
- Effect: memory accesses have a bigger impact on the program efficiency
- Need compiler optimizations to improve memory system performance (e.g. increase cache hit rate)

Data Dependencies

- Compiler must reason about dependence between instructions
- Three kinds of dependencies:
  - True dependence:

```
(s1)  x = ...
(s2)  ... = x
```

  - Anti dependence:

```
(s1)  ... = x
(s2)  x = ...
```

  - Output dependence:

```
(s1)  x = ...
(s2)  x = ...
```

- Cannot reorder instructions in any of these cases!

Data Dependences

- In the context of hardware design, dependences are another kind of hazard:
  - True dependence = RAW hazard (read after write)
  - Anti dependence = WAR hazard (write after read)
  - Output dependence = WAW hazard (write after write)
- A transformation is correct if it preserves all dependences in the program
- How easy is it to determine dependences?
- Trivial for scalar variables (variables of primitive types)

  x = …

  … = x

Problem: Pointers

- Data dependences not obvious for pointer-based accesses
- Pointer-based loads and stores:

```
(s1)  *p = ...
(s2)  ... = *q
```

- s1, s2 may be dependent if $Ptr(p) \cap Ptr(q) \neq \varnothing$
- Need pointer analysis to determine dependent instructions!
- More precise analyses compute smaller pointer sets, can detect (and parallelize) more independent instructions

Problem: Arrays

- Array accesses also problematic:

```
(s1)  a[i] = ...
(s2)  ... = a[j]
```

- s1, s2 may be dependent if i=j in some execution of the program
- Usually, array elements accessed in nested loops, access expressions are linear functions of the loop indices
- Lot of existing work to formalize the array data dependence problem in this context

Next time…

- Memory management
- Then: the last class!