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, \(\text{ivar} = |V|\)
- Let \(N\) = set of integer numbers
- Use a lattice over the set \(V \times N\)
- Construct the lattice starting from a lattice for \(N\)
- Problem: \((N, \leq)\) is not a complete lattice!
  - … why?

Constant Folding Lattice

- Second try: lattice \((N, \{\bot, 1\}, \leq)\)
  - Where \(\bot \leq n\), for all \(n \in N\)
  - And \(n \leq 1\), for all \(n \in N\)
  - Is complete!
- Meaning:
  - \(v = 1\): don’t know if \(v\) is constant
  - \(v = \bot\): \(v\) is not constant
**Constant Folding Lattice**

- **Second try:** lattice \((\mathbb{N} \cup \{\top, \perp\}, \leq)\)
  - Where \(\perp \leq n\), for all \(n \in \mathbb{N}\)
  - And \(n \leq \top\), for all \(n \in \mathbb{N}\)
  - 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 …

**Solution:** flat lattice \(L = (\mathbb{N} \cup \{\perp\}, <)\)

- Where \(\perp < n\), for all \(n \in \mathbb{N}\)
- And \(n < \top\), for all \(n \in \mathbb{N}\)
- And distinct integer constants are not comparable

Note: meet of any two distinct numbers is \(\perp\)!

**Constant Folding Lattice**

- **Denote** \(\mathbb{N}^* = \mathbb{N} \cup \{\top, \perp\}\)
- Use flat lattice \(L = (\mathbb{N}^*, <)\)

**Constant folding lattice:** \(L' = (V \rightarrow \mathbb{N}^*, <_C)\)

Where partial order on \(V \rightarrow \mathbb{N}^*\) is defined as:

- \(X \subseteq_C Y\) iff for each variable \(v\): \(X(v) \subseteq Y(v)\)

Can represent a function in \(V \rightarrow \mathbb{N}^*\) as a set of assignments: \(\{\{v_1 = c_1\}, \{v_2 = c_2\}, ..., \{v_n = c_n\}\}\)

**CF: Transfer Functions**

- **Transfer function** for instruction \(I:\)
  \[F_I(X) = (X - \text{kill}[I]) \cup \text{gen}[I]\]
  
  - Where \(\text{kill}[I]\) = constants "killed" by \(I\)
  
  - \(\text{gen}[I]\) = constants "generated" by \(I\)

  - Slightly tricky part: what is \(\{v=5\} \cup \{v=\perp\}\)?

  - If \(I\) is \(v = c\) (constant):
    \[\text{gen}[I] = \{v\} \quad \text{kill}[I] = \{v\} \times \mathbb{N}\]

  - If \(I\) is \(v = u + w\):
    \[\text{gen}[I] = \{v\} \quad \text{kill}[I] = \{v\} \times \mathbb{N}\]

  - Where \(e = X[u] + X[w]\), if \(X[u]\) and \(X[w]\) are not \(\top, \perp\)

  - \(e = \perp\), if \(X[u] = \perp\) or \(X[w] = \perp\)

  - \(e = \top\), if \(X[u] = \top\) or \(X[w] = \top\)

**CF: Distributivity**

- **Example:**

  \[
  \{x=2, y=3, z=\top\} \\
  \{x=3, y=2, z=\top\} \\
  \{x=3, y=2, z=\top\}
  \]

  - At join point, apply meet operator
  
  - Then use transfer function for \(z = x \times y\)
**CF: Distributivity**

- Example:
  \[
  (x=2, y=3, z=\top) \quad \Rightarrow \quad x = 3 \quad y = 2 \quad \Rightarrow \quad \{x=\bot, y=\bot, z=\top\}
  \]

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

**Example:**

\[
\begin{align*}
  x &= 2 \\
  y &= 3 \\
  z &= x + y
\end{align*}
\]

\[
\{x=3,y=2,z=\top\} \quad \Rightarrow \quad \{x=\bot,y=\bot,z=\bot\}
\]

**Reason for MOP \neq MFP:**

- Transfer function \(F\) of \(z=x+y\) is not distributive!

\[
F(X_1 \cap X_2) \neq F(X_1) \cap F(X_2)
\]

where \(X_1 = \{x=2,y=3,z=\top\}\) and \(X_2 = \{x=3,y=2,z=\top\}\)

**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: \(\cap, \cup = \emptyset\)
  - 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: \(\cap, \cup = S\)
  - 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, whitout 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
- Simultaneously execute stages of different instructions
  - Instr 1
  - Instr 2
  - Instr 3

Stall the Pipeline

- It is not always possible to pipeline instructions
- Example 1: branch instructions
  - Branch
  - Target
- Example 2: load instructions
  - Load
  - Use

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: \((s1) \ x = \ldots \ (s2) \ \ldots = x\)
  - Anti dependence: \((s1) \ \ldots = x \ (s2) \ x = \ldots\)
  - Output dependence: \((s1) \ x = \ldots \ (s2) \ x = \ldots\)
- Cannot reorder instructions in any of these cases!

Problem: Pointers
- Data dependences not obvious for pointer-based accesses
- Pointer-based loads and stores:
  \[
  \begin{align*}
  (s1) \ p &= \ldots \\
  (s2) \ \ldots &= q
  \end{align*}
  \]
- \(s1, s2\) may be dependent if \(\text{Ptr}(p) \cap \text{Ptr}(q) \neq \emptyset\)
- 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:

  \[
  \begin{align*}
  (s1) & \quad a[i] = \ldots \\
  (s2) & \quad \ldots = a[j] \\
  \end{align*}
  \]

- \(s1, s2\) 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!