Lecture 18

COMPILER DESIGN
• HW5: OAT v. 2.0
  – Records, function pointers, type checking, array-bounds checks, etc.
  – Due: Thursday, November 28th at 23:59

• Final Exam
  – Scheduled for Friday, January 31st, 9-11 AM
interface Incrementable {
    public void inc();
}
class IntCounter implements Incrementable {
    public void add(int);
    public void inc();
    public int value();
}
class FloatCounter implements Incrementable {
    public void inc();
    public void add(float);
    public float value();
}

void add2(Incrementable x) { x.inc(); x.inc(); }
Code Specialization

• Idea: create specialized versions of a function that is called from different places with different arguments.
• Example: specialize function $f$ in:

```java
class A implements I { int m() {...} }
class B implements I { int m() {...} }
int f(I x) { x.m(); }  // don't know which m
A a = new A(); f(a);  // know it's A.m
B b = new B(); f(b);  // know it's B.m
```

• $f_A$ would have code specialized to dispatch to $A.m$
• $f_B$ would have code specialized to dispatch to $B.m$
• You can also inline methods when the run-time type is known statically
  – Often just one class implements a method.

Zhendong Su    Compiler Design
Common Subexpression Elimination

• In some sense it’s the opposite of inlining: fold redundant computations together
• Example:

\[ a[i] = a[i] + 1 \] compiles to:
\[ [a + i*4] = [a + i*4] + 1 \]
Common subexpression elimination removes the redundant add and multiply:
\[ t = a + i*4; \ [t] = [t] + 1 \]
• For safety, you must be sure that the shared expression always has the same value in both places!
Unsafe Common Subexpression Elimination

- Example: consider this OAT function:

```c
unit f(int[] a, int[] b, int[] c) {
    int j = ...; int i = ...; int k = ...;
    b[j] = a[i] + 1;
    c[k] = a[i];
    return;
}
```

- The following optimization that shares the expression `a[i]` is unsafe… why?

```c
unit f(int[] a, int[] b, int[] c) {
    int j = ...; int i = ...; int k = ...;
    t = a[i];
    b[j] = t + 1;
    c[k] = t;
    return;
}
```
LOOP OPTIMIZATIONS
Loop Optimizations

• Program hot spots often occur in loops
  – Especially inner loops
  – Not always: consider operating systems code or compilers vs. a computer game or word processor

• Most program execution time occurs in loops
  – The 90/10 rule of thumb holds here too --- 90% of the execution time is spent in 10% of the code

• Loop optimizations are very important, effective, and numerous
  – Also, concentrating effort to improve loop body code is usually a win
Loop Invariant Code Motion (revisited)

- Another form of redundancy elimination
- If the result of a statement or expression does not change during the loop and it’s pure, it can be hoisted outside the loop body
- Often useful for array element addressing code
  - Invariant code not visible at the source level

```java
for (i = 0; i < a.length; i++) {
    /* a not modified in the body */
}

// Hoisted loop-invariant expression
int t = a.length;
for (i = 0; i < t; i++) {
    /* same body as above */
}
```
Strength Reduction (revisited)

- Strength reduction can work for loops too
- Idea: replace expensive operations (multiplies, divides) by cheap ones (adds and subtracts)
- For loops, create a dependent induction variable:

  Example:
  ```c
  for (int i = 0; i < n; i++) {
      a[i*3] = 1;
  } // stride by 3
  
  int j = 0;
  for (int i = 0; i < n; i++) {
      a[j] = 1;
      j = j + 3; // replace multiply by add
  }
  ```
Loop Unrolling (revisited)

- Branches can be expensive, unroll loops to avoid them

```c
for (int i=0; i<n; i++) { S }
```

```c
for (int i=0; i<n-3; i+=4) {S;S;S;S};
for (       ; i<n; i++) { S }  // left over iterations
```

- With k unrollings, eliminates (k-1)/k conditional branches
  - So for the above program, it eliminates 3/4 of the branches
- Space-time tradeoff
  - Not a good idea for large S or small n
- Interacts with instruction caching, branch prediction
EFFECTIVENESS?
Optimization Effectiveness?

\[
\text{%speedup} = \left( \frac{\text{base time}}{\text{optimized time}} - 1 \right) \times 100\%
\]

Example:
base time = 2s
optimized time = 1s \quad \Rightarrow \quad 100\% \text{ speedup}

Example:
base time = 1.2s
optimized time = 0.87s \quad \Rightarrow \quad 38\% \text{ speedup}

Graph taken from:
Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic.
Formal Verification of SSA-Based Optimizations for LLVM.
In Proc. ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), 2013
Optimization Effectiveness?

- **mem2reg**: promotes alloca’ed stack slots to temporaries to enable register allocation

- **Analysis**
  - mem2reg alone (+ back-end optimizations like register allocation) yields ~78% speedup on average
  - -O1 yields ~100% speedup
    (so all the rest of the optimizations combined account for ~22%)
  - -O3 yields ~120% speedup

- **Hypothetical program that takes 10 sec. (base time):**
  - Mem2reg alone: expect ~5.6 sec
  - -O1: expect ~5 sec
  - -O3: expect ~4.5 sec
CODE ANALYSIS
Motivating Code Analyses

- Many things might influence the safety/applicability of an optimization
  - What algorithms and data structures can help?

- How do we know what is a loop?
- How do we know an expression is invariant?
- How do we know if an expression has no side effects?
- How do we keep track of where a variable is defined?
- How do we know where a variable is used?
- How do we know if two reference values may be aliases of one another?
Moving Toward Register Allocation

• The OAT compiler currently generates as many temp. variables as it needs
  – These are the %uids you should be very familiar with by now

• Current compilation strategy
  – Each %uid maps to a stack location
  – This yields programs with many loads/stores to memory
  – Very inefficient

• Ideally, we’d like to map as many %uid’s as possible into registers
  – Eliminate the use of the alloca instruction?
  – Only 16 max registers available on 64-bit X86
  – %rsp, %rbp are reserved; some have special semantics, so only 10 or 12 available
  – This means that a register must hold more than one slot

• When is this safe?
Liveness

• Observation: %uid1 and %uid2 can be assigned to the same register if their values will not be needed at the same time
  – What does it mean for an %uid to be “needed”?
  – Ans: its contents will be used as a source operand in a later instruction
• Such a variable is called “live”
• Two variables can share the same register if they are not live at the same time
Scope vs. Liveness

• We can already get some coarse liveness information from variable scoping
• Consider the following OAT program
  ```c
  int f(int x) {
    var a = 0;
    if (x > 0) {
      var b = x * x;
      a = b + b;
    }
    var c = a * x;
    return c;
  }
  ```

• Note that due to OAT’s scoping rules, variables b and c can never be live at the same time
  – c’s scope is disjoint from b’s scope
• So, we could assign b and c to the same alloca’ed slot and potentially to the same register
But Scope is too Coarse

• Consider this program

```c
int f(int x) {
    int a = x + 2;
    int b = a * a;
    int c = b + x;
    return c;
}
```

• The scopes of a, b, c, x all overlap – they’re all in scope at the end of the block.

• But, a, b, c are never live at the same time
  – So they can share the same stack slot / register
• A variable $v$ is *live* at a program point if $v$ is defined before the program point and used after it.

• Liveness is defined in terms of where variables are *defined* and where variables are *used*.

• Liveness analysis: Compute the live variables between each statement
  – May be *conservative* (i.e. it may claim a variable is live when it isn’t) so because that’s a safe approximation
  – To be useful, it should be more *precise* than simple scoping rules

• Liveness analysis is one example of *dataflow analysis*
  – Other examples: Available Expressions, Reaching Definitions, Constant-Propagation Analysis, …
• For the purposes of dataflow analysis, we use the control-flow graph (CFG) intermediate form.

• Recall that a basic block is a sequence of instructions such that:
  – There is a distinguished, labeled entry point (no jumps into the middle of a basic block)
  – There is a (possibly empty) sequence of non-control-flow instructions
  – The block ends with a single control-flow instruction (jump, conditional branch, return, etc.)

• A control flow graph
  – Nodes are blocks
  – There is an edge from B1 to B2 if the control-flow instruction of B1 might jump to the entry label of B2
  – There are no “dangling” edges – there is a block for every jump target.

• Note: the following slides are intentionally a bit ambiguous about the exact nature of the code in the control flow graphs:
  – at the x86 assembly level
  – an “imperative” C-like source level
  – at the LLVM IR level
  – Same general idea, but the exact details will differ
    • e.g. LLVM IR doesn’t have “imperative” update of %uid temporaries.
    • In fact, the SSA structure of the LLVM IR makes some of these analyses simpler.
Dataflow over CFGs

• For precision, it is helpful to think of the “fall through” between sequential instructions as an edge of the control-flow graph too.
  – Different implementation tradeoffs in practice…

Basic block CFG

“Exploded” CFG

Fall-through edges

in-edges

out-edges
Liveness is Associated with *Edges*

- This is useful so that the same register can be used for different temporaries in the same statement.
- Example: \( a = b + 1 \)

- Compiles to:

  ```
  Mov a, b
  Live: b

  Add a, 1
  Live: a

  Mov eax, eax
  Live: a (maybe)
  ```

  Register Allocate:
  \[ a \to eax, b \to eax \]

  ```
  Add eax, 1
  ```
Uses and Definitions

• Every instruction/statement uses some set of variables
  – i.e. reads from them

• Every instruction/statement defines some set of variables
  – i.e. writes to them

• For a node/statement \( s \) define:
  – use\([s]\) : set of variables used by \( s \)
  – def\([s]\) : set of variables defined by \( s \)

• Examples:
  – \( a = b + c \)
    use\([s]\) = \{b,c\}  def\([s]\) = \{a\}
  – \( a = a + 1 \)
    use\([s]\) = \{a\}  def\([s]\) = \{a\}
A variable $v$ is live on edge $e$ if:

There is
- a node $n$ in the CFG such that $\text{use}[n]$ contains $v$, and
- a directed path from $e$ to $n$ such that for every statement $s'$ on the path, $\text{def}[s']$ does not contain $v$

The first clause says that $v$ will be used on some path starting from edge $e$.

The second clause says that $v$ won’t be redefined on that path before the use.

Questions:
- How to compute this efficiently?
- How to use this information (e.g. for register allocation)?
- How does the choice of IR affect this? (e.g. LLVM IR uses SSA, so it doesn’t allow redefinition $\Rightarrow$ simplify liveness analysis)
Simple, inefficient algorithm

• “A variable \( v \) is live on an edge \( e \) if there is a node \( n \) in the CFG using it and a directed path from \( e \) to \( n \) passing through no def of \( v \).”

• Backtracking Algorithm:
  – For each variable \( v \)
  – Try all paths from each use of \( v \), tracing backwards through the control-flow graph until either \( v \) is defined or a previously visited node has been reached.
  – Mark the variable \( v \) live across each edge traversed.

• Inefficient because it explores the same paths many times (for different uses and different variables)
Dataflow Analysis

• **Idea:** compute liveness information for all variables simultaneously.
  – Keep track of sets of information about each node

• **Approach:** define *equations* that must be satisfied by any liveness determination.
  – Equations based on “obvious” constraints.

• **Solve the equations by iteratively converging on a solution.**
  – Start with a “rough” approximation to the answer
  – Refine the answer at each iteration
  – Keep going until no more refinement is possible: a *fixpoint* has been reached

• This is an instance of a general framework for computing program properties: dataflow analysis
Dataflow Value Sets for Liveness

- Nodes are program statements, so:
  - $\text{use}[n]$ : set of variables used by $n$
  - $\text{def}[n]$ : set of variables defined by $n$
  - $\text{in}[n]$ : set of variables live on entry to $n$
  - $\text{out}[n]$ : set of variables live on exit from $n$

- Associate $\text{in}[n]$ and $\text{out}[n]$ with the “collected” information about incoming/outgoing edges

- For Liveness: what constraints are there among these sets?
  - Clearly:
    $$\text{in}[n] \supseteq \text{use}[n]$$

- What other constraints?
Other Dataflow Constraints

• We have: \( \text{in}[n] \supseteq \text{use}[n] \)
  – “A variable must be live on entry to \( n \) if it is used by \( n \)”

• Also: \( \text{in}[n] \supseteq \text{out}[n] - \text{def}[n] \)
  – “If a variable is live on exit from \( n \), and \( n \) doesn’t define it, it is live on entry to \( n \)”
  – Note: here ‘-’ means “set difference”

• And: \( \text{out}[n] \supseteq \text{in}[n'] \) if \( n' \in \text{succ}[n] \)
  – “If a variable is live on entry to a successor node of \( n \), it must be live on exit from \( n \)”
Iterative Dataflow Analysis

• Find a solution to those constraints by starting from a rough guess.
• Start with: $\text{in}[n] = \emptyset$ and $\text{out}[n] = \emptyset$
• They don’t satisfy the constraints:
  – $\text{in}[n] \supseteq \text{use}[n]$
  – $\text{in}[n] \supseteq \text{out}[n] - \text{def}[n]$
  – $\text{out}[n] \supseteq \text{in}[n']$ if $n' \in \text{succ}[n]$

• Idea: iteratively re-compute $\text{in}[n]$ and $\text{out}[n]$ where forced to by the constraints.
  – Each iteration will add variables to the sets $\text{in}[n]$ and $\text{out}[n]$
    (i.e. the live variable sets will increase monotonically)
• We stop when $\text{in}[n]$ and $\text{out}[n]$ satisfy these equations:
  (which are derived from the constraints above)
  – $\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$
  – $\text{out}[n] = \bigcup_{n'\in\text{succ}[n]} \text{in}[n']$
for all \( n \), \( \text{in}[n] := \emptyset \), \( \text{out}[n] := \emptyset \)
repeat until no change in ‘in’ and ‘out’
  for all \( n \)
    \[
    \text{out}[n] := \bigcup_{n' \in \text{succ}[n]} \text{in}[n']
    \]
  \[
  \text{in}[n] := \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
  \]
end
end

• Finds a fixpoint of the \text{in} and \text{out} equations.
  – The algorithm is guaranteed to terminate… Why?
• Why do we start with \( \emptyset \)?