Lecture 18

COMPILER DESIGN
• **HW4**: due soon

• **HW5**: OAT v. 2.0
  - Records, function pointers, type checking, array-bounds checks, etc.
A high-level tour of a variety of optimizations.
Optimizations

• The code generated by our OAT compiler so far is pretty inefficient
  – Lots of redundant moves
  – Lots of unnecessary arithmetic instructions

• Consider this OAT program

```c
int foo(int w) {
    var x = 3 + 5;
    var y = x * w;
    var z = y - 0;
    return z * 4;
}
```
Unoptimized vs. Optimized Output

Hand optimized

_foo:
  shlq  $5, %rdi
  movq  %rdi, %rax
  ret

Compiler may inline foo, so just one instruction!
Why do we need optimizations?

• To help programmers
  – They write modular, clean, high-level programs
  – Compiler generates efficient, high-performance assembly

• Programmers don’t write optimal code
• High-level PLs make avoiding redundant computation hard
  – e.g. $A[i][j] = A[i][j] + 1$
• Architectural independence
  – Optimal code depends on features not expressed to the programmer
  – Modern architectures assume optimization

• Different kinds of optimizations
  – **Time**: improve execution speed
  – **Space**: reduce amount of memory needed
  – **Power**: lower power consumption (e.g. to extend battery life)
Some caveats

- Optimization are code transformations
  - They can be applied at any stage of the compiler
  - They must be safe --- shouldn’t change the meaning of the program

- In general, optimizations require some program analysis
  - To determine if the transformation really is safe
  - To determine whether the transformation is cost effective

- This course
  - Most common and valuable performance optimizations
  - See Muchnick (optional text) for ~10 chapters about optimization
When to apply optimization

- Inlining
- Function specialization
- Constant folding
- Constant propagation
- Value numbering
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength Reduction
- Constant folding & propagation
- Branch prediction / optimization
- Register allocation
- Loop unrolling
- Cache optimization

Zhendong Su    Compiler Design
Where to Optimize?

- Usual goal: improve time performance
- Problem: many optimizations trade space for time

Example: *Loop unrolling*

```plaintext
for(int i=0; i<100; i=i+1) {
  s = s + a[i];
}
```

```plaintext
for(int i=0; i<99; i=i+2){
  s = s + a[i];
  s = s + a[i+1];
}
```

- Tradeoffs
  - Increasing code space slows down whole program a tiny bit (extra instructions to manage) but speeds up the loop a lot
  - For frequently executed code with long loops, generally a win
  - Interacts with instruction cache and branch prediction hardware

- Complex optimizations may never pay off!
Writing Fast Programs In Practice

- Pick the right algorithms and data structures
  - These have much bigger impact on performance than optimizations
  - Reduce number of operations & memory accesses
  - Minimize indirection – it breaks working-set coherence

- Then turn on compiler optimizations

- Profile to determine program hot spots

- Evaluate whether the algorithm/data structure design works

- ...if so: “tweak” the source code until the optimizer does “the right thing” to the machine code
Safety

• Whether an optimization is **safe** depends on the language semantics
  – Languages with **weaker guarantees** permit more optimizations
    … but have **more ambiguity** in their behavior
  – In Java, tail-call optimization is not (yet) valid/supported
  – In C, loading from uninitialized memory is undefined

• Example: *loop-invariant code motion (LICM)*
  – Idea: hoist invariant code out of a loop

```java
while (b) {
    z = y/x;
    ...
    // x, y not updated
}
```

• Is this more efficient?
• Is this safe?
Constant Folding

• Idea: If operands are statically known, compute value at compile-time

\[
\text{int } x = (2 + 3) \times y \Rightarrow \text{int } x = 5 \times y
\]
\[
b \& \text{false} \Rightarrow \text{false}
\]

• Performed at every stage of optimization, why?
  – Constant expressions can be created by translation or earlier optimizations

• Example: \(A[2]\) might be compiled to

\[
\text{MEM[MEM[A] + 2 \times 4]} \Rightarrow \text{MEM[MEM[A] + 8]}
\]
Constant Folding Conditionals

- if (true) S ➞ S
- if (false) S ➞ ;
- if (true) S else S' ➞ S
- if (false) S else S' ➞ S'
- while (false) S ➞ ;
- if (2 > 3) S ➞ ;
Algebraic Simplification

• More general form of constant folding
  – Take advantage of mathematically sound simplification rules

• Identities
  – $a \times 1 \Rightarrow a$
  – $a \times 0 \Rightarrow 0$
  – $a + 0 \Rightarrow a$
  – $a - 0 \Rightarrow a$
  – $b \mid \text{false} \Rightarrow b$
  – $b \& \text{true} \Rightarrow b$

• Reassociation & commutativity
  – $(a + 1) + 2 \Rightarrow a + (1 + 2) \Rightarrow a + 3$
  – $(2 + a) + 4 \Rightarrow (a + 2) + 4 \Rightarrow a + (2 + 4) \Rightarrow a + 6$

• Strength reduction: (replace expensive op with cheaper op)
  – $a \times 4 \Rightarrow a << 2$
  – $a \times 7 \Rightarrow (a << 3) - a$
  – $a / 32767 \Rightarrow (a >> 15) + (a >> 30)$

• Note
  – Must be careful with floating-point and integer arithmetic
    • Due to rounding and overflow/underflow
  – Iteration of these optimizations is useful, but by how much?
Constant Propagation

• If a variable’s value is a constant, replace its uses by the constant

• Value of a variable is propagated forward from the point of assignment
  – This is a substitution operation

• Example
  int x = 5;
  int y = x * 2; \rightarrow int y = 5 * 2; \rightarrow int y = 10; \rightarrow
  int z = a[y]; \rightarrow int z = a[y]; \rightarrow int z = a[y]; \rightarrow int z = a[10];

• To be most effective, constant propagation and folding interleave
Copy Propagation

• If variable $x$ is assigned to $y$, replace $x$’s uses with $y$
  – Need to know where copies of the variable propagate
  – Interacts with the scoping rules of the language

• Example

```plaintext
x = y;
if (x > 1) {
  x = x * f(x - 1);
}
```

```plaintext
\underline{x = y;}
if (y > 1) {
  x = y * f(y - 1);
}
```

• Can make the first assignment to $x$ dead code, thus eliminated
Dead Code Elimination

• If side-effect free code can never be observed, safe to eliminate it

\[ x = y \times y \quad // \quad x \text{ is dead!} \]

\[ \ldots \quad // \quad x \text{ never used} \quad \rightarrow \quad \ldots \]

\[ x = z \times z \]

• A variable is *dead* if it is never used after it is defined
  – Computing such *def/use* information is an important compiler component

• Dead variables can be created by other optimizations
Unreachable/Dead Code

• Basic blocks unreachable from the entry block can be deleted
  – Performed at the IR or assembly level
  – Improves cache, TLB performance

• Dead code: similar to unreachable blocks
  – A value might be computed but never subsequently used
• Code for computing the value can be dropped

• ... but only if it’s pure, i.e. it has no externally visible side effects
  – Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket
  – Note: Pure functional languages (e.g. Haskell) make reasoning about the safety of optimizations (and code transformations in general) easier
Inlining

• Replace a function call with the body of the function
  (with arguments rewritten to be local variables)

• Example in OAT code
  ```c
  int g(int x) { return x + pow(x); }
  int pow(int a) { int b = 1; int n = 0;
      while (n < a) {b = 2 * b}; return b; }
  ```

  ```c
  int g(int x) { int a = x; int b = 1; int n = 0;
      while (n < a) {b = 2 * b}; tmp = b; return x + tmp;
  }
  ```

• May need to rename variable names to avoid name capture
  – Example of what can go wrong?
• Best done at the AST or relatively high-level IR
• When is it profitable?
  – Eliminates the stack manipulation, jump, etc.
  – Can increase code size
  – Enables further optimizations
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 runtime type is known statically
  - Often just one class implements a method
Common Subexpression Elimination

• In some sense, it is the opposite of inlining:
  fold redundant computations together

• Example

\[
a[i] = a[i] + 1 \quad \text{compiles to} \\
[a + i*4] = [a + i*4] + 1
\]

Common subexp. elimination removes the redundant add and multiply
\[
t = a + i*4; \quad [t] = [t] + 1
\]

• When is it safe?
  – The shared expression must always have 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 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

```

```java
for (i = 0; i < t; i++) {
    /* same body as above */
}
```

Zhendong Su    Compiler Design
Strength Reduction (revisited)

- Strength reduction can work for loops too
  replace expensive ops (*, /) by cheap ones (+, -)

- 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
int i;
for (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 ¾ 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?

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

Example:
- base time = 2s
- optimized time = 1s
  \[\Rightarrow\] 100% speedup

Example:
- base time = 1.2s
- optimized time = 0.87s
  \[\Rightarrow\] 38% 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

Zhendong Su  Compiler Design
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 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 reserved; some have special semantics, so only 10-12 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.

• Q: What does it mean for an \%uid to be “needed”?
  A: Its contents will be used as a source operand in a later instruction.

• Such a variable is called “live”.

• Two variables can share a 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
  
  ```
  int f(int x) {
      var a = 0;
      if (x > 0) {
          var b = x * x;
          a = b + b;
      }
      var c = a * x;
      return c;
  }
  ```

- Due to OAT's scoping rules, b can c can never be live the same time
  - c's scope is disjoint from b's scope
- So, can assign b, c to the same alloca'ed slot & potentially to 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 --- 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
Live Variable Analysis

• Variable $v$ is *live* at a program point $L$ if
  – $v$ is defined before $L$
  – $v$ is used after $L$

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

• Liveness analysis: *Compute the live variables between each statement*
  – May be *conservative* (i.e. it may claim a variable live when it isn’t) 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, …
Control-flow Graphs Revisited

• For 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
  – A block ends with a single control-flow instruction: jump, branch, return, etc.

• A control flow graph
  – Nodes are blocks
  – An edge from B1 to B2 if B1’s control-flow instruction may 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
Liveness is Associated with *Edges*

- This is useful as the same register can be used for different temporaries in the same statement

- Example: \( a = b + 1 \)

Compiles to

\[
\begin{align*}
\text{Mov } a, b & \quad \text{Live: } b \\
\text{Add } a, 1 & \quad \text{Live: } a \\
\text{Add eax, 1} & \quad \text{Register Allocate: } \ a \rightarrow \text{eax}, \ b \rightarrow \text{eax}
\end{align*}
\]
Uses and Definitions

• Every instruction/statement \textit{uses} some set of variables
  \hspace{1cm} i.e. reads from them
• Every instruction/statement \textit{defines} some set of variables
  \hspace{1cm} i.e. writes to them

• For a node/statement \( s \) define
  \hspace{1cm} \text{use}[s] : \text{set of variables used by } s
  \hspace{1cm} \text{def}[s] : \text{set of variables defined by } s

• Examples:
  \hspace{1cm} a = b + c \hspace{1cm} \text{use}[s] = \{b,c\} \hspace{1cm} \text{def}[s] = \{a\}
  \hspace{1cm} a = a + 1 \hspace{1cm} \text{use}[s] = \{a\} \hspace{1cm} \text{def}[s] = \{a\}
Liveness, Formally

• 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 $s^\prime$ on the path, $\text{def}[s^\prime]$ does not contain $v$

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

• The second 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 (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 backward through the control-flow graph until either $v$ is defined or a previously visited node is 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 hold 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 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: \( \text{in}[n] \supseteq \text{out}[n] \setminus \text{def}[n] \)
  - “If a variable is live on exit from \( n \), and \( n \) doesn’t define it, it is live on entry to \( n \)”
  - Note: here ‘\( \setminus \)’ 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
  - \( \text{in}[n] \supseteq \text{use}[n] \)
  - \( \text{in}[n] \supseteq \text{out}[n] \setminus \text{def}[n] \)
  - \( \text{out}[n] \supseteq \text{in}[n'] \) if \( n' \in \text{succ}[n] \)

- Idea: iteratively re-compute \( \text{in}[n] \) & \( \text{out}[n] \) where forced to by constraints
  - Each iteration will add variables to the sets \( \text{in}[n] \) & \( \text{out}[n] \)
    (i.e. the live variable sets will increase monotonically)
- We stop when \( \text{in}[n] \) & \( \text{out}[n] \) satisfy these equations
  (which are derived from the constraints above)
  - \( \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \)
  - \( \text{out}[n] = \bigcup_{n' \in \text{succ}[n]} \text{in}[n'] \)
Complete Liveness Analysis Algorithm

for all $n$, $in[n] := \emptyset$, $out[n] := \emptyset$
repeat until no change in ‘in’ and ‘out’
  for all $n$
    $out[n] := \bigcup_{n' \in \text{succ}[n]} in[n']$
    $in[n] := \text{use}[n] \cup (out[n] \setminus \text{def}[n])$
  end
end

• Finds a fixpoint of the in & out equations
  – The algorithm is guaranteed to terminate, why?
• Why do we start with $\emptyset$?