Lecture 22

CMSC 350: COMPILER DESIGN
OPTIMIZATIONS

A high-level tour of a variety of optimizations.
Optimizations

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

• Consider this Tiger program:

```tiger
function foo(w: int) : int =
  let var x := 3 + 5
  var y := x * w
  var z := y - 0 in
  z * 4
end
```
Unoptimized vs. Optimized Output

Hand optimized code:

```
LABEL(foo)
STORE(R1,4,FP)
LOAD(R1,3,FP)
SETLO(Rt,32)
MUL(Rt,Rt,Rt)
STORE(R1,3,FP)
LOAD(R1,4,FP)
LOAD(R1,3,FP)
RETURN(FP_alt,PC_ret)
```

- Function foo may be inlined by the compiler, so this might reduce even further!
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 languages make avoiding redundant computation inconvenient or impossible
  – 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 – they 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
When to apply optimization

<table>
<thead>
<tr>
<th>Level</th>
<th>Node</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>High level</td>
<td>AST</td>
<td>• Inlining</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Function specialization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Constant folding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Constant propagation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Value numbering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Dead code elimination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Loop-invariant code motion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Common sub-expression elimination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Strength Reduction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Constant folding &amp; propagation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Branch prediction / optimization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Register allocation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Loop unrolling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Cache optimization</td>
</tr>
<tr>
<td>Low level</td>
<td>Abstract assembly</td>
<td>• Inlining</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Function specialization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Constant folding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Constant propagation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Value numbering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Dead code elimination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Loop-invariant code motion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Common sub-expression elimination</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Strength Reduction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Constant folding &amp; propagation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Branch prediction / optimization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Register allocation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Loop unrolling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Cache optimization</td>
</tr>
</tbody>
</table>
Where to Optimize?

- Usual goal: improve time performance
- Problem: many optimizations trade space for time
- Example: *Loop unrolling*
  - Idea: rewrite a loop like:
    ```
    for(int i=0; i<100; i=i+1) {
      s = s + a[i];
    }
    ```
  - Into a loop like:
    ```
    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 a much bigger impact on performance than compiler optimizations.
  - Reduce # of operations
  - Reduce 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 programming language semantics.
  – Languages that provide weaker guarantees to the programmer permit more optimizations, but have more ambiguity in their behavior.
  – e.g. In Java tail-call optimization (that turns recursive function calls into loops) is not valid.
  – e.g. In C, loading from initialized memory is undefined, so the compiler can do anything.

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

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

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

- Idea: If operands are known at compile time, perform the operation statically.

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

- 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)
  - $a \times 4 \rightarrow a \ll 2$
  - $a \times 7 \rightarrow (a \ll 3) - a$
  - $a / 32767 \rightarrow (a \gg 15) + (a \gg 30)$

- Note 1: must be careful with floating point (due to rounding) and integer arithmetic (due to overflow/underflow)
- Note 2: iteration of these optimizations is useful... how much?
Constant Propagation

- If the value is known to be a constant, replace the use of the variable by the constant
- Value of the variable must be 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];  int z = a[y];  int z = a[y];  int z = a[10];
  ```

- To be most effective, constant propagation should be interleaved with constant folding
Copy Propagation

- If one variable is assigned to another, replace uses of the assigned variable with the copied variable.
- Need to know where copies of the variable propagate.
- Interacts with the scoping rules of the language.

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

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

- Can make the first assignment to x dead code (that can be eliminated).
Dead Code Elimination

• If a side-effect free statement can never be observed, it is safe to eliminate the statement.

```
x = y * y  // x is dead!
...       // x never used  ➔  ...
x = z * z
```

• A variable is **dead** if it is never used after it is defined.
  – Computing such *definition* and *use* information is an important component of compiler

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

• Basic blocks not reachable by any trace leading from the starting basic block are *unreachable* and 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 call to a function with the body of the function itself with arguments rewritten to be local variables:
- Example:

```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 run-time type is known statically
  - Often just one class implements a method.
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 Java function:
  ```java
  void f(int[] a, int[] b, int[] c) {
      int j = ...; int i = ...; int k = ...;
      b[j] = a[i] + 1; c[k] = a[i];
  }
  ```

- The following optimization that shares the expression `a[i]` is unsafe... why?
  ```java
  void f(int[] a, int[] b, int[] c) {
      int j = ...; int i = ...; int k = ...;
      t = a[i];
      b[j] = t + 1; c[k] = t;
  }
  ```
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. (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 */
}

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

Hoisted loop-invariant expression
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.
  
  ```
  for (int i=0; i<n; i++) { S; }
  ```

  ```
  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 $\frac{3}{4}$ 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?

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

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

Example:
- base time = 1.2s
- optimized time = 0.87s
  \Rightarrow 38\% speedup
Optimization Effectiveness?

- **mem2reg**: promotes allocated 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