CSE P 501 – Compilers

Optimizing Transformations
Hal Perkins
Spring 2018
Agenda

• A closer look at some common optimizing transformations

• More details and examples later when we look at analysis algorithms
Optimizations in a Compiler

Source → Front End → ‘Middle End’ → Back End → Target

Scan → Parse → Convert → Optimize → Select Instructions → Allocate Registers → Emit → Machine Code

chars → tokens → AST → IR → IR → IR → IR → IR

AST = Abstract Syntax Tree
IR = Intermediate Representation

UW CSE P 501 Spring 2018
Role of Transformations

• Dataflow analysis discovers opportunities for code improvement

• Compiler rewrites the (IR) to make these improvements
  – Transformation may reveal additional opportunities for further optimization
  – May also block opportunities by obscuring information
Organizing Transformations in a Compiler

- Typically middle end consists of many phases
  - Analyze IR
  - Identify optimization
  - Rewrite IR to apply optimization
  - And repeat (50 phases in a commercial compiler is typical)
- Each individual optimization is supported by rigorous formal theory
- But no formal theory for what order or how often to apply them(!)
  - Some rules of thumb and best practices
  - May apply some transformations several times as different phases reveal opportunities for further improvement
Each optimization requires a 'pass' (linear scan) over the IR
IR may sometimes shrink, sometimes expand
Some optimizations may be repeated
'Best' ordering is heuristic
Don't try to beat an optimizing compiler - you will lose!

Note: not all programs are written by humans!
Machine-generated code can pose a challenge for optimizers
  eg: a single function with 10,000 statements, 1,000+ local variables, loops nested 15 deep, spaghetti of "GOTOs", etc
A Taxonomy

• Machine Independent Transformations
  – Mostly independent of target machine
    (e.g., loop unrolling will likely make it faster regardless of target)
  – “Mostly”? – e.g., vectorize only if target has SIMD ops
  – Worthwhile investment – applies to all targets

• Machine Dependent Transformations
  – Mostly concerned with instruction selection & scheduling, register allocation
  – Need to tune for different targets
  – Most of this in the back end, but some in the optimizer
Machine Independent Transformations

- Dead code elimination
  - unreachable or not actually used later
- Code motion
  - “hoist” loop-invariant code out of a loop
- Specialization
- Strength reduction
  - $2 \times x \Rightarrow x + x$; $@A+((i \times \text{numcols} + j) \times \text{eltsize}) \Rightarrow p += 4$
- Enable other transformations
- Eliminate redundant computations
  - Value numbering, GCSE
Machine Dependent Transformations

• Take advantage of special hardware
  – e.g., expose instruction-level parallelism (ILP)
  – e.g., use special instructions (VAX polyf; x86 sqrt, strings)
  – e.g., use SIMD (vector) instructions and registers

• Manage or hide latencies
  – e.g., tiling/blocking and loop interchange
  – Improves cache behavior – hugely important

• Deal with finite resources - # functional units

• Compilers generate for a vanilla machine, e.g., SSE2
  – But provide switches to tune (arch:AVX, arch:IA32)
  – JIT compiler knows its target architecture!
Optimizer Contracts

• **Prime directive**
  – No optimization will change observable program behavior!
  – This can be subtle. e.g.:
    • What is "observable"? (via IO? to another thread?)
    • Dead-Code-Eliminate a *throw*?
    • Language Reference Manual may be ambiguous/undefined/negotiable for edge cases

• **Avoid harmful optimizations**
  – If an optimization does not improve code significantly, don't do it: it harms throughput
  – If an optimization degrades code quality, don't do it
Is this *hoist* legal?

```java
for (int i = start; i < finish; ++i) a[i] += 7;
```

```java
i = start
loop:
    if (i >= finish) goto done
    if (i < 0 || i >= a.length) throw OutOfBounds
    a[i] += 7
    ++i
    goto loop
done:
```

```java
if (start < 0 || finish >= a.length) throw OutOfBounds
i = start
loop:
    if (i >= finish) goto done
    a[i] += 7
    ++i
    goto loop
done:
```

Another example: "volatile" pretty much kills all attempts to optimize
Dead Code Elimination

• If a compiler can prove that a computation has no external effect, it can be removed
  – Unreachable operations – always safe to remove
  – Useless operations – reachable, may be executed, but results not actually required

• Dead code often results from other transformations
  – Often want to do DCE several times
Dead Code Elimination

• Classic algorithm is similar to garbage collection
  – Pass I – Mark all useful operations
    • Instructions whose result does, or can, affect visible behavior:
      – Input or Output
      – Updates to object fields that might be used later
      – Instructions that may throw an exception (e.g.: array bounds check)
      – Calls to functions that might perform IO or affect visible behavior
    – (Remember, for many languages, compiler does not process entire program at one time – but a JIT compiler might be able to)
    • Mark all useful instructions
    • Repeat until no more changes
  – Pass II – delete all unmarked operations
Code Motion

- Idea: move an operation to a location where it is executed less frequently
  - Classic situation: *hoist* loop-invariant code: execute once, rather than on every iteration

- Lazy code motion & *partial* redundancy

```
b = b + 1
a = b * c

a = b * c = a . . .
```

Replicate, so `a` need not be re-calculated

```
b = b + 1
a = b * c

= a . . .
```

`a` must be re-calculated - wasteful if control took right-hand arm

Replicate, so `a` need not be re-calculated
Specialization I

• Idea: Replace general operation in IR with more specific
  – Constant folding:
    • feet_per_minute = mph * feet_per_mile/minutes_per_hour
    • feet_per_minute = mph * 5280 / 60
    • feet_per_minute = mph * 88
  – Replacing multiplications and division by constants with shifts (when safe)
  – Peephole optimizations
    • movl $0,%eax     =>  xorl %eax,%eax
Specialization:2 - Eliminate Tail Recursion

• Factorial - recursive
  
  int fac(n) = if (n <= 2) return 1; else return n * fac(n - 1);

• 'accumulating' Factorial - tail-recursive
  facaux(n, r) = if (n <= 2) return r; else return facaux(n - 1, n*r)
  call facaux(n, 1)

• Optimize-away the call overhead; replace with simple jump
  facaux(n, r) = if (n <= 2) return r;
  else n = n - 1; r = n*r; jump back to start of facaux
  – So replace recursive call with a loop and just one stack frame

• Issue?
  – Avoid stack overflow - good! - "observable" change?
Strength Reduction

• Classic example: Array references in a loop
  ```
  for (k = 0; k < n; k++) a[k] = 0;
  ```

• Naive codegen for a[k] = 0 in loop body
  ```
  movl $4,%eax // elemsize = 4 bytes
  imull offset \_k(%rbp),%eax // k * elemsize
  addl offset \_a(%rbp),%eax // &a[0] + k * elemsize
  mov $0,(%eax) // a[k] = 0
  ```

• Better!
  ```
  movl offset \_a(%rbp),eax // &a[0], once-off

  movl $0,(%eax) // a[k] = 0
  addl $4,%eax // eax = &a[k+1]
  ```

Note: *pointers* allow a user to do this directly in C or C++

Eg: \[ \text{for} \ (p = a; p < a + n; ) \ *p++ = 0; \]
Implementing Strength Reduction

• Idea: look for operations in a loop involving:
  – A value that does not change in the loop, the *region constant*, and
  – A value that varies systematically from iteration to iteration, the *induction variable*

• Create a new induction variable that directly computes the sequence of values produced by the original one; use an addition in each iteration to update the value
Other Common Transformations

• Inline substitution (procedure bodies)

• Cloning / Replicating

• Loop Unrolling

• Loop Unswitching
Inline Substitution - "inlining"

Class with trivial **getter**
```java
class C {
    int x;
    int getx() { return x; }
}
```

Method **f** calls **getx**
```java
class X {
    void f() {
        C c = new C();
        int total = c.getx() + 42;
    }
}
```

Compiler **inlines** body of **getx** into **f**
```java
class X {
    void f() {
        C c = new C();
        int total = c.x + 42;
    }
}
```

- Eliminates **call** overhead
- Opens opportunities for more optimizations
- Can be applied to large method bodies too
- Aggressive optimizer will inline 2 or more deep
- Increases total code size (memory & cache issues)
- With care, is a huge win for OO code
Code Replication

Original

```c
if (x < y) {
    p = x + y;
} else {
    p = z + 1;
}
q = p * 3;
w = y + x;
```

Replicated code

```c
if (x < y) {
    p = x + y;
    q = p * 3;
    w = y + x;
} else {
    p = z + 1;
    q = p * 3;
    w = y + x;
}
```

- + : extra opportunities to optimize in larger basic blocks (eg: LVN)
- - : increase total code size - may impact effectiveness of I-cache
Loop Unrolling

• Idea: Replicate the loop body
  – More opportunity to optimize loop body
  – Increases chances for good schedules and instruction level parallelism
  – Reduces loop overhead (reduce test/jumps by 75%)

• Catches
  – must ensure unrolled code produces the same answer: "loop-carried dependency analysis"
  – code bloat
  – don't overwhelm registers
Loop Unroll Example

Original

```c
for (i = 1, i <= n, i++) {
    a[i] = a[i] + b[i];
}
```

Unrolled

```c
i = 1;
while (i + 3 <= n) {
    a[i]   = a[i]   + b[i];
    a[i+1] = a[i+1] + b[i+1];
    a[i+2] = a[i+2] + b[i+2];
    a[i+3] = a[i+3] + b[i+3];
    i += 4;
}
while (i <= n) {
    a[i] = a[i] + b[i];
    i++;
}
```
Loop Unswitching

• Idea: if the condition in an if-then-else is loop invariant, rewrite the loop by pulling the if-then-else out of the loop and generating a tailored copy of the loop for each half of the new conditional

  – After this transformation, both loops have simpler control flow – more chances for rest of compiler to do better
Loop Unswitch Example

Original

```java
for (i = 1, i <= n, i++) {
    if (x > y) {
        a[i] = b[i]*x;
    } else {
        a[i] = b[i]*y;
    }
}
```

Unswitched

```java
if (x > y) {
    for (i = 1; i <= n; i++) {
        a[i] = b[i]*x;
    }
} else {
    for (i = 1; i <= n; i++) {
        a[i] = b[i]*y;
    }
}
```

- **IF** condition does not change value in this code snippet
- No need to check $x > y$ on every iteration
- Do the **IF** check once!
Summary

• Just a sampler
  – 100s of transformations in the literature
  – Will examine several in more detail, particularly involving loops

• Big part of engineering a compiler is:
  – decide which transformations to use
  – decide in what order
  – decide if & when to repeat each transformation

• Compilers offer options:
  – optimize for speed
  – optimize for codesize
  – optimize for specific target micro-architecture
  – optimize for power consumption(!)

• Competitive bench-marking will investigate many permutations
What’s next

• Careful look at several analysis and transformation algorithms
• Value numbering / dominators
• Dataflow
• Loops, loops, loops
  – Dominators – discovering loop structures
  – Loop-invariant code
  – Loop Transformations

• And an hour on (simple) code gen for the project