CS 1622: Optimization

Jonathan Misurda
jmisurda@cs.pitt.edu

A "Bad" Name

Optimization is the process by which we turn a program into a better one, for some definition of better.

This is impossible in the general case.

For instance, a *fully optimizing compiler* for size must be able to recognize all sequences of code that are infinite loops with no output, so that it can replace it with a one-instruction infinite loop.

This means we must solve the halting problem.

So, what can we do instead?

Optimization

An optimizing compiler transforms *P* into a program *P'* that always has the same input/output behavior as *P*, and might be smaller or faster.

Optimizations are code or data transformations which typically result in improved performance, memory usage, power consumption, etc.

Optimizations applied naively may sometimes result in code that performs *worse*.

We saw one potential optimization before, ***loop interchange***, where we decide to change the order of loop headers in order to get better cache locality. However, this may result in worse overall performance if the resulting code must do more work and the arrays were small enough to fit in the cache regardless of the order.

Register Allocation

Register allocation is also an optimization as we previously discussed.

On register-register machines, we avoid the cost of memory accesses anytime we can keep the result of one computation available in a register to be used as an operand to a subsequent instruction.

Good register allocators also do coalescing which eliminates move instructions, making the code smaller and faster.

Dataflow Analyses

Reaching Definitions

Does a particular value *t* directly affect the value of *t* at another point in the program?

Given an *unambiguous* definition *d*,

$$t \leftarrow a \oplus b$$

or

$$t \leftarrow M[a]$$

we say that *d reaches* a statement *u* in the program if there is some path in the CFG from *d* to *u* that does not contain any unambiguous definition of *t*.

An *ambiguous* definition is a statement that might or might not assign a value to *t*, such as a call with pointer parameters or globals. MiniJava will not register allocate these, and so we can ignore the issue.

Reaching Definitions

We label every move statement with a definition ID, and we manipulate sets of definition IDs.

We say that the statement

`d1: t ← x ⊕ y`

generates the definition d_1, because no matter what other definitions reach the beginning of this statement, we know that d_1 reaches the end of it.

This statement **kills** any other definition of *t*, because no matter what other definitions of *t* reach the beginning of the statement, they cannot directly affect the value of *t* after this statement.

Reaching Definitions

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

This looks familiar, but is the reverse of our liveness calculations.

We solve it using iteration the same as with liveness.

Available Expressions

An expression:

`x ⊕ y`

is **available** at a node *n* in the flow graph if, on every path from the entry node of the graph to node *n*, `x ⊕ y` is computed at least once *and* there are no definitions of *x* or *y* since the most recent occurrence of `x ⊕ y` on that path.

Any node that computes `x ⊕ y` *generates* `{x ⊕ y}`, and any definition of *x* or *y kills* `{x ⊕ y}`.

A *store* instruction (`M[a] ← b`) might modify any memory location, so it kills any *fetch* expression (`M[x]`). If we were sure that *a* = *x*, we could be less conservative, and say that `M[a] ← b` does not kill `M[x]`. This is called **alias analysis.**

Available Expressions

$$in[n] = \bigcap_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Compute this by iteration.

Define the *in* set of the start node as empty, and initialize all other sets to *full* (the set of all expressions), not empty.

Intersection makes sets *smaller*, not bigger.

Reaching Expressions

We say that an expression:

`t ← x ⊕ y`

(in node *s* of the flow graph) reaches node *n* if there is a path from *s* to *n* that does not go through any assignment to *x* or *y*, or through any computation of *x* ⊕ *y*.

Dataflow Optimizations

Common Subexpression Elim

Compute *reaching expressions*, that is, find statements of the form

```
n: v ← x ⊕ y
```

such that the path from n to s does not compute $x \oplus y$ or define x or y.

Choose a new temporary w, and for such n, rewrite as:

```
n: w ← x ⊕ y
n': v ← w
```

Finally, modify statement s to be:

```
s: t ← w
```

We will rely on copy propagation to remove some or all of the extra assignment quadruples.

Constant Propagation

Suppose we have a statement d:

```
t ← c
```

where c is a constant,

and another statement n that uses t:

```
y ← t ⊕ x
```

We know that t is constant in n if d reaches n, and no other definitions of t reach n.

In this case, we can rewrite n as:

```
y ← c ⊕ x
```

Copy Propagation

This is like constant propagation, but instead of a constant c we have a variable z.

Suppose we have a statement:

```
d: t ← z
```

and another statement n that uses t, such as:

```
n: y ← t ⊕ x
```

If d reaches n, and no other definition of t reaches n, and there is no definition of z on any path from d to n (including a path that goes through n one or more times), then we can rewrite n as:

```
n: y ← z ⊕ x
```

Dead-code Elimination

If there is a quadruple

```
s : a ← b ⊕ c
```

or

```
s : a ← M[x]
```

such that a is not *live-out* of s, then the quadruple can be deleted.

Some instructions have implicit side effects such as raising an exception on overflow or division by zero. The deletion of those instructions will change the behavior of the program.

The optimizer shouldn't always do this. Optimizations that eliminate even seemingly harmless runtime behavior cause unpredictable behavior of the program. A program debugged with optimizations on may fail with them disabled.

Loop Optimizations

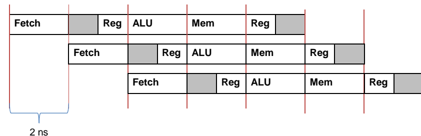
Single Cycle Implementation

Each instruction can be done with a single 8 ns cycle

Time between the first and fourth instruction: 24 ns

For three Load instructions, it will take 3 * 8 = 24 ns

Pipelined Implementation



Each step takes 2 ns (even register file access) because the slowest step is 2 ns

Time between 1st and 4th instruction starting: 3 * 2 ns = 6 ns

Total time for the three instructions = 14 ns

Control Hazards

Control hazard: attempt to make a decision before condition is evaluated.

Branch instructions:

```
beq $1,$2,L0
add $4,$5,$6
...
L0: sub $7,$8,$9
```

Which instruction do we fetch next?

Make a **guess** that the branch is **not taken**. If we're right, there's no problem (no stalls). If we're wrong…?

What would have been stalls if we waited for our comparison are now "wrong" instructions. We need to cancel them out and make sure they have no effect. These are called **bubbles**.

Branch Prediction

Attempt to predict the outcome of the branch before doing the comparison.

- Predict branch taken (fetch branch target instruction)
- Predict branch not taken (fetch fall through)

If wrong, we'll need to squash the mispredicted instructions by setting their control signals to zero (no writes). This turns them into nops.

Times to do prediction:

- Static
  - Compiler inserts hints into the instruction stream
  - CPU predicts forward branches not taken and backwards branches taken
- Dynamic
  - Try to do something in the CPU to guess better

Dynamic Branch Prediction

Use a branch's history to predict the next time it is executed.

Consider a loop that executes 10 times. The first 9 iterations, we can statically predict that the backwards edge of our loop is taken correctly. However, on the final iteration, we take the fall through and our forward branch is mispredicted. Our accuracy is 90%.

A 1-bit predictor remembers the taken/not-taken status of a branch in the past. It uses that to predict the branch the next time it is encountered. In our example, this would work the same as a static prediction.

A Branch Target Buffer might remember more history. For instance, a 16-entry branch target buffer in our previous example could store all 10 iterations. If we encounter the same loop again, we will predict it with 100% accuracy.

Loop Unrolling

A tight loop may perform better if it is unrolled: where multiple loop iterations are replaced by multiple copies of the body in a row.

```
int x;
for (x = 0; x < 100; x++)
{
    printf("%d\n", x);
}
```

```
int x;
for (x = 0; x < 100; x+=5)
{
     printf("%d\n", x);
     printf("%d\n", x+1);
     printf("%d\n", x+2);
     printf("%d\n", x+3);
     printf("%d\n", x+4);
 }
```

Loop Unrolling

Benefits:

- Reduce branches and thus potentially mispredictions
- More instruction-level parallelism

Drawbacks:

- Code size increase can cause instruction cache pressure
- Increased register usage may result in spilling

Duff's Device

```
do {
    *to = *from++;
    /* Note that the 'to' pointer
is NOT incremented */
} while(--count > 0);
```

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch(count % 8) {
        case 0: do { *to=*from++;
        case 7:      *to=*from++;
        case 6:      *to=*from++;
        case 5:      *to=*from++;
        case 4:      *to=*from++;
        case 3:      *to=*from++;
        case 2:      *to=*from++;
        case 1:      *to=*from++;
                   } while(--n>0);
    }
}
```

Strength Reduction

Certain machine instructions are more expensive than others. One classic example is multiplication being 30ish cycles while addition and shifting are just 1 cycle.

Strength reduction replaces a more expensive operation with a cheaper one.

In the simplest case, we can replace multiplications with shifts (taking care to deal with the rounding of negative numbers correctly).

In the general case, we can accumulate a multiplied value each iteration in a method reminiscent of Horner's method.

Method Optimizations

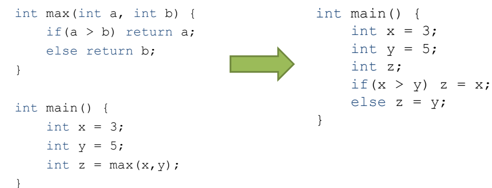
Method Inlining

Method inlining replaces a function call site with the body of the callee.

Example:

```
int max(int a, int b) {
    if(a > b) return a;
    else return b;
}

int main() {
    int x = 3;
    int y = 5;
    int z = max(x,y);
}
```



```
int main() {
    int x = 3;
    int y = 5;
    int z;
    if(x > y) z = x;
    else z = y;
}
```

Method Inlining

Benefits:

- Less dynamic instructions
  - Call site removed, prologue and epilogue code eliminated
- Smaller dynamic memory needs since the activation record is eliminated
- Removal of control flow transfer helps eliminate branch penalties and improves instruction cache locality
- After inlining is performed, more code is available to the optimizer to improve

Disadvantages:

- Static code size increase is likely
- Code growth can impact instruction cache performance
- May increase register pressure

Method Inlining

In languages like C++, there is a keyword `inline` that hints to the compiler that a method should be inlined during compilation.

In C, this is one of the benefits of using a parameterized `#define` macro.

In OOPLs, we often have very small methods (usually acting as accessors and mutators) that can be inlined.

Inlining is not always the right thing to do, and so the compiler must use heuristics to decide to apply it or not.

Unknown depth recursion makes inlining difficult.

Tail Recursion Elimination

A **tail call** is a function call site that appears as the last statement in a function.

Example:

```
int factorial(int x) {
    if(x < 2) return 1;
    return x * fact(x-1);
}
```

Tail calls can be implemented without adding a new activation record to the stack.

The activation record of the original call is reused, substituting in the new parameter values as appropriate. The tail call is then replaced with a jump to the beginning of the function.

Runtime Optimization

Just-in-time Compilation

Just-in-time (JIT) compilers are software dynamic translators that convert one language into another at runtime, when a segment of code (often a method) is needed.

JIT compilers can be used to support dynamic languages which are not traditionally compiled such as JavaScript or to support languages compiled into platform-independent bytecode, like Java.

Since the JIT compiler serves as a runtime environment, we can access dynamic properties of the program in order to better optimize it.

Java

```
class EvenOdd {
    public static void main(String args[]) {
        long evenSum=0, oddSum=0;
        for(int i=0;i<1000000;i++) {
            if(i%2 == 0) {
                evenSum+=i;
            }
            else {
                oddSum+=i;
            }
        }
        System.out.println("Even sum: " + evenSum);
        System.out.println("Odd sum: " + oddSum);
    }
}
```

Bytecode

```
00 : lconst_0
01 : lstore_1
02 : lconst_0
03 : lstore_3
04 : iconst_0
05 : istore                  local.05
07 : iload                   local.05
09 : ldc                     1
0B : if_icmpge               pos.2A
0E : iload                   local.05
10 : iconst_2
11 : irem
12 : ifne                    pos.1E
15 : lload_1
16 : iload                   local.05
18 : i2l
19 : ladd
1A : lstore_1
1B : goto                    pos.24
1E : lload_3
1F : iload                   local.05
21 : i2l
22 : ladd
```

Bytecode

```
23 : lstore_3
24 : iinc                    local.05, 1
27 : goto                    pos.07
2A : getstatic               java.io.PrintStream java.lang.System.out
2D : new                     java.lang.StringBuilder
30 : dup
31 : invokespecial           void java.lang.StringBuilder.<init>()
34 : ldc                     "Even sum: "
36 : invokevirtual           java.lang.StringBuilder java.lang.StringBuilder.append(java.lang.String)
39 : lload_1
3A : invokevirtual           java.lang.StringBuilder java.lang.StringBuilder.append(long)
3D : invokevirtual           java.lang.String java.lang.StringBuilder.toString()
40 : invokevirtual           void java.io.PrintStream.println(java.lang.String)
43 : getstatic               java.io.PrintStream java.lang.System.out
46 : new                     java.lang.StringBuilder
49 : dup
4A : invokespecial           void java.lang.StringBuilder.<init>()
4D : ldc                     "Odd sum: "
4F : invokevirtual           java.lang.StringBuilder java.lang.StringBuilder.append(java.lang.String)
52 : lload_3
53 : invokevirtual           java.lang.StringBuilder java.lang.StringBuilder.append(long)
56 : invokevirtual           java.lang.String java.lang.StringBuilder.toString()
59 : invokevirtual           void java.io.PrintStream.println(java.lang.String)
5C : return
```

Adaptive Optimization

Consider a JIT-compiler as part of a Java Virtual Machine (JVM).

Java Bytecode is a stack-oriented machine language. The JVM can use interpretation to implement the virtual CPU for Java Bytecode.

However, this interpretation is slow.

One option is to use JIT compilation to convert the bytecode into machine code.

However, compilation can be slow as well.

We have two ideas:
1. Apply cheap yet effective optimizations
2. Apply optimization only to regions that will benefit from it

Cost-Benefit Analysis

Let us only apply optimization O to method M if the time gained from the optimization is greater than the cost of doing the optimization.

cost(M' = O(M)) < (cost(M) - cost(M'))

We need models of cost for both the optimization and the optimized method. How can we know?

For the cost of the optimization, we can base this on the complexity of the method and the average-case complexity of the algorithm O.

For the cost of the optimized method M', we must be able to predict the future.

To do this, we can look at the past.

Profiling

We can generate a profile of method M to identify how "hot" it is. That is, how much time has been spent in the method so far by inserting instrumentation into the method code during compilation or interpretation to see how much time is spent in that code.

Execution time typically follows the **Pareto Principle** (the 80/20 or 90/10 rule):

90%(80%) of the time is spent in 10%(20%) of the code.

Let us only optimize those methods where we are likely to gain the most.

Choice of Optimizations

Let us then define an adaptive optimization scheme. Each time a method reaches a certain level of hotness, recompile it at the next level.

For instance, a Java Adaptive Optimizer might have 4 or 5 levels:

Level 0: no optimization, potentially even interpretation
Level 1: Baseline JIT, naïve code that still mimics the stack-based nature of the bytecode
Level 2: Do register allocation to remove as many of the operand stack loads and stores as possible
Level 3: Apply some basic control and dataflow optimizations
Level 4: Aggressively optimize the code

This is how the Java HotSpot compiler and Jikes Research Virtual Machine work.