COMP3048: Lecture 17

Code Optimisations

Matthew Pike

University of Nottingham, Ningbo

This Lecture: Optimization

- ***Code improvement*** or ***optimization***: what is it?
- High-level, intermediate-level, and low-level optimization.
- Time and space trade-offs.
- Specific optimizations; e.g.
  - Constant folding
  - Common subexpression evaluation
  - Inlining
- Interaction among Optimizations

Code Improvement (1)

The code generated by a compiler

- ***must*** be correct
  (i.e., semantics-preserving translation)
- ***should*** also
  - run fast
  - be small
  - use as little space as possible

Code improvement is the process of improving the time and/or space behaviour of generated code ***without*** changing its functional behaviour;
i.e. correctness ***must*** be preserved.

Code Improvement (2)

Consider:

```
w := 42;
i := 0;
while (i < 100) do begin
    j := 0;
    while (j < 200) do begin
        x := (w * 10) * a[i];
        y := y + x + b[j];
        j := j + 1;
    end;
    i := i + 1;
end
```

How might this code fragment be changed to (likely) make it run faster?

Code Improvement (3)

Example: Replacing the code fragment

```
f(x) + f(x)
```

by

```
2 * f(x)
```

saves a function call; likely reduces execution time.

Any caveat???

Only correct if `f` does ***not*** have any side effects!

Code Improvement (4)

Consider:

```
var x: Integer;
...
fun f (y: Integer): Integer =
    begin
        x = x + 1;
        return x + y
    end
...
x = 2;
putint(f(2) + f(2))
```

This code fragment would print `11`, whereas the result of printing `2 * f(2)` would be `10`.

Code Improvement (5)

Note: “Side effect” includes:

- Updates of variables, data-structures
- I/O and other changes to the system state
- Exceptions
- Non-termination

Optimization?

Code improvement usually referred to as “optimization”. However:

- Hardly ever possible to ***guarantee*** optimality under any mathematical measure.
- Not even always an improvement: not known what is going to happen at run-time, so “optimizing” for the ***average expected*** case.
- Careful and extensive ***benchmarking*** is often the only way to verify that an optimization indeed does improve generated code most of the time.

At What Level? (1)

Code improvement can be done at different levels:

- High level: source-to-source (AST) transformations.
- Intermediate level: transformations on intermediate representation, e.g.:
  - “bare-bones” high-level language
  - control/data flow graph representation
- Low level: transformations on machine code.

Each level suitable for different kinds of optimization. Improve at all levels!

At What Level? (2)

Consider this code fragment:

```
if x then
    if y then
        putint(1)
    else
        putint(2)
else
    putint(3)
```

Anything that obviously could be improved at this level; i.e. the ***source code*** level?

At What Level? (3)

Resulting TAM code might be:

```
        LOAD      [SB + 12]      #2: LOADL   2
        JUMPIFZ #0                   CALL    putint
        LOADL     [SB + 13]      #3: JUMP    #1
        JUMPIFZ #2               #0: LOADL   3
        LOADL     1                  CALL    putint
        CALL      putint         #1:
        JUMP      #3
```

Now anything that could be improved;
i.e., at the ***machine code*** level?

At What Level? (4)

Information that was implicit in the high-level representation might become explicit at the intermediate level, thus enabling/facilitating certain optimizations.

Consider array indexing. High-level code fragments:

```
var x, y: array[1..100] Integer;
...
a := x[i] + y[i];
```

At What Level? (4)

Intermediate (C-like) code with explicit pointer arithmetic:

```
if (i < 1 || i > 100) then raise index_bounds;
t1 := ^(x + 4 * (i - 1));
if (i < 1 || i > 100) then raise index_bounds;
t2 := ^(y + 4 * (i - 1));
a := t1 + t2
```

(^ is the pointer dereferencing operator.)

At What Level? (5)

This could be optimimzed by reusing common subexpressions and eliminating redundant array bounds checks:

```
if (i < 1 || i > 100) then raise index_bounds;
t0 := 4 * (i - 1)
t1 := ^(x + t0);
t2 := ^(y + t0);
a := t1 + t2;
```

Time vs. Space (1)

Time and space optimizations are often in conflict. Consider representing an array of Booleans:

- Each Boolean represented by one machine word:
  - fast access
  - wastes space.
- Each Boolean represented by a single bit:
  - space efficient
  - access requires extra operations (shifting and masking): takes time (and some ***instruction*** space)!

Time vs. Space (2)

In other cases, small is fast as well:

- Basic observation: accessing memory is slow. The fewer instructions and the fewer pieces of data, the fewer memory accesses, and the faster the execution.
- It is highly desirable to keep inner loops small so that they fit in the first-level ***instruction*** cache.
- It is desirable to keep the set of “currently accessed” memory locations small so that they fit in the first-level ***data*** cache.

Time vs. Space (3)

But then again, since memory access is very slow, avoiding a memory access could sometimes be worth a few extra instructions!

(Reason: Instruction fetching is typically much faster than data fetching because it is more predictable.)

Conclusion: the trade-off between time and space is a highly complicated issue!

In practice, one often have to make en educated guess, then verify by benchmarking.

Common Optimization Techniques

Applicable at the source-code (AST) level and/or intermediate level:

- Constant Folding
- Common Subexpression Elimination
- Algebraic Identities
- Copy Propagation
- Dead Code Elimination
- Strength Reduction
- Code Motion
- Loop Unrolling
- Inlining

Constant Folding (1)

Idea: evaluate (sub)expressions at compile-time where possible:

```
const pi: Double = 3.1416;
var volume, radius: Double;
...
volume := 4/3 * pi * radius^3;
```

`4/3 * pi` can be evaluated at compile-time:

```
const pi: Double = 3.1415;
var volume, radius: Double;
...
volume := 4.1888 * radius^3;
```

Constant Folding (2)

Not only applicable to ***declared*** constants:

```
x := 3;
y := x + 1;
x := x * 2;
```

can be optimized to

```
x := 3;
y := 4;
x := 6;
```

Constant Folding (3)

In general, ***flow analysis*** required:

```
x := 3;
y := x + 1;
while (x < z) begin
    x := x * 2
end
```

Unless `z` is known, we can only optimize to:

```
x := 3;
y := 4;
while (x < z) begin
    x := x * 2
end
```

Common Subexpression Elimination (1)

Idea: avoid evaluating the “same expression” more than once.

```
x1 := y1 + 7 * z + 42;
x2 := y2 + 7 * z + 42;
```

can be optimized to

```
t := 7 * z + 42;
x1 := y1 + t;
x2 := y2 + t;
```

Common subexpressions often appear in ***address computations*** in intermediate code.

Common Subexpression Elimination (2)

The expressions must not only be ***syntactically*** the same; they must also ***mean*** the same thing:

- Scope rules must be taken into account; consider Haskell-like `let`-expressions (i.e., functional code, ***no*** side effects):

  ```
  let x = y * 17 in
      let y = 13 in
          let z = y * 17
  ```

  The innermost `y * 17` ***cannot*** be replaced by `x`.

Common Subexpression Elimination (3)

- Side effects must be taken into account (flow analysis):

```
x := y * 17 + 3;
y := y + 1;
z := y * 17 + 3;
```

Here, the two instances of `y * 17 + 3` do ***not*** compute the same value.

Indeed, the expressions themselves could have side effects (C-like increment operator):

```
x := y++ * 17 + 3;
z := y++ * 17 + 3;
```

Algebraic Identities (1)

Algebraic identities can be exploited to:

- simplify expressions: `1 * x - 0` $\Rightarrow$ `x`
- expose further opportunities for e.g. common subexpression evaluation:

  ```
  x := (2 + z) * i;
  y := (z + 2) * j;
  ```

  can be transformed into

  ```
  t := z + 2;
  x := t * i;
  y := t * j;
  ```

Algebraic Identities (2)

However, standard algebraic identities do not always hold!

Is it safe to assume that `x + (y + z)` has the same meaning as `(x + y) + z`?

- Not if overflow/underflow is ***trapped***: if `x` and `y` are large positive numbers, and `z` is a large negative number, then `(x + y) + z` might result in a trap, while `x + (y + z)` doesn't.
- ***Floating point addition*** is not associative!

Copy Propagation (1)

Idea: After an assignment that ***copies*** a value, like `x := y` (often result of earlier optimization), use `y` in place of `x` wherever possible:

```
x := y;
v := x * 17;
w := x + 19;
```

can be transformed to

```
x := y;
v := y * 17;
w := y + 19;
```

Copy Propagation (2)

It may then turn out that the assigned variable is ***never used again***. In that case, the assignment is ***dead code*** and can be eliminated.

```
x := y;
v := y * 17;
w := y + 19;
```

can be optimized to

```
v := y * 17;
w := y + 19;
```

if `x` is never used again.

Dead Code Elimination (1)

Idea: It may be possible to ***statically*** determine that certain parts of the code

- will never be reached
- will not have any effect

The former is called ***unreachable*** code, the latter ***dead*** code.

Sometimes unreachable code is also referred to as dead code.

Either way, both are examples of ***useless*** code that can be ***removed*** without changing the meaning of the program.

Dead Code Elimination (2)

Consider the following Java fragment:

```
debug = false;
...
if (debug) {
    System.out.println("Got here!");
}
```

After constant folding, we have

```
if (false) {
    System.out.println("Got here!");
}
```

and the print statement is manifestly unreachable.

Dead Code Elimination (3)

In the copy propagation example, we saw that an assignment like

```
x := y;
```

could be removed if `x` is never used again as it has no effect and thus is dead code.

However, care needed: even if the assigned variable is never used, execution of the assignment statement itself might have an effect, meaning it ***cannot*** be removed (in its entirety):

```
x := y++;
```

Strength Reduction (1)

Idea: replace “expensive” operations by cheaper ones. Simple examples:

- Addition and shifting might be cheaper than multiplication:

  ```
  5 * x ⇒ x « 2 + x
  ```

- Multiplication might be cheaper than exponentiation:

  ```
  x^2 ⇒ x * x
  z := x^5
   ⇒ x2 := x * x; z := x2 * x2 * x
  ```

  Only applies when ***known*** integral power.

Strength Reduction (2)

A loop may have a number of ***induction variables*** that remain in ***lock step***:

```
i := 10;
while (i > 0) do begin
    i := i - 1;
    t := 4 * i;
    a[i] := b[t]
end
```

Here, `i` and `t` are induction variables.

Strength Reduction (3)

All that is going on is that `t` decreases by 4 each time round the loop. We can rephrase as follows:

```
i := 10;
t := 4 * i;
while (i > 0) do begin
    i := i - 1;
    t := t - 4;
    a[i] := b[t]
end
```

An potentially expensive multiplication has been replaced by a subtraction ***inside*** a loop.

Code Motion (1)

Idea: code that is ***loop invariant***, i.e. evaluate to the same value at each loop iteration, should be moved outside the loop.

```
for (i := 0; i <= m - 1; i++) do
    for (j := 0; j <= n - 1; j++) do
        x := x + a[i * 10 + j]
```

- `m - 1` and `n - 1` invariant in the outer loop
- `i * 10` invariant in the inner loop.

Code Motion (2)

Thus we can transform to:

```
t1 := m - 1;
t2 := n - 1;
for (i := 0; i <= t1; i++) do begin
    t3 := i * 10;
    for (j := 0; j <= t2; j++) do
        x := x + a[t3 + j]
end
```

Array address computations and bounds checks often introduce loop invariant code fragments.

Code Motion (3)

Of course, we have to be careful if there are side effects. Consider:

```
for (i := 0; i < n; i++) do
    x := x + f(17);
```

The function call `f(17)` might look like loop invariant code at a first glance, but it could have side effects, in which case it is wrong to move it out of the loop:

```
f(n) = begin z := z + n; return z end;
```

Loop Unrolling (1)

As loops carry certain overheads (evaluation of loop condition, jumps), it can be beneficial to unroll loops that are known to be short. Consider:

```
for (i := 0; i < 5; i++) do
    a[i] := b[4 - i] * 2^i;
```

Loop unrolling yields:

```
a[0] := b[4 - 0] * 2^0;
a[1] := b[4 - 1] * 2^1;
a[2] := b[4 - 2] * 2^2;
a[3] := b[4 - 3] * 2^3;
a[4] := b[4 - 4] * 2^4;
```

Loop Unrolling (2)

The resulting code can often be further improved; e.g. by constant folding:

```
a[0] := b[4] * 1;
a[1] := b[3] * 2;
a[2] := b[2] * 4;
a[3] := b[1] * 8;
a[4] := b[0] * 16;
```

Loop Unrolling (3)

Caveats:

- Loop unrolling can cause the code to grow considerably: space vs. time trade off.
- Impact of cache memories:
  - The instructions for a short loop will fit into the instruction cache and can thus be fetched again very quickly for each iteration.
  - Each instruction for an unrolled loop has to be fetched from main memory.

Loop Unrolling (4)

Loops where the bounds are statically unknown can sometimes still be partially unrolled:

```
for (i := 0; i < n; i++) do
    a[i] := b[i] + c[i];
```

can for example be transformed into (integer div.!):

```
for (i := 0; i < (n/2)*2; i := i+2) do begin
    a[i] := b[i] + c[i];
    a[i + 1] := b[i + 1] + c[i + 1]
end;
if (i < n) then begin
    a[i] := b[i] + c[i];
    i++
end;
```

Loop Unrolling (5)

Benefits:

- Number of iterations reduced (here, roughly halved).
- Increased size of loop body may open up for further improvements; e.g. constant folding, CSE, strength reduction as discussed earlier (in particular for index address calculations).

Inlining (1)

Idea: Avoid overhead of function/procedure call by instantiating the body with the actual parameters and copying the result to the call site.

Also called ***procedure integration***.

- Inlined procedures/functions should be ***small***, or size of code might blow up!
- Careful with ***recursion***! Otherwise the ***compiler*** might get stuck in a loop.
- Can make sense to unfold recursive procedures/ functions a few times: similar to loop unrolling.

Inlining (2)

```
fun f (x: Integer): Integer =
    begin
        return (x + 17) * 123
    end
...
x := f(a + 3);
y := f(x * 3);
```

Inlining would result in the last fragment becoming:

```
x := ((a + 3) + 17) * 123;
y := ((x * 3) + 17) * 123;
```

Inlining (3)

Consider:

```
fun fib (x : Integer) : Integer =
begin
    return (x<2 ? x : fib(x-1) + fib(x-2))
end
```

Recursion! Care needed!

If we blindly inline `fib` everywhere just because it initially looks small, compiler will get stuck in a loop (exhausting the memory eventually)!

Interaction among Optimizations (1)

One optimization might generate opportunities for other optimizations:

```
const level: Integer = 4;
const debugging: Boolean = true;
func debug(severity: Integer) =
begin
    return debugging && severity > level
end
...
x := 10;
if debug(3) then begin
    print "Oops! Well, got here.";
    x := x + 1
end;
y := x + 10;
```

Interaction among Optimizations (2)

Inlining yields:

```
const level: Integer = 4;
const debugging: Boolean = true;
...
x := 10;
if debugging && 3 > 4 then begin
    print "Oops! Well, got here.";
    x := x + 1
end;
y := x + 10;
```

Interaction among Optimizations (3)

Constant folding yields:

```
x := 10;
if false then begin
    print "Oops! Well, got here.";
    x := x + 1
end;
y := x + 10;
```

Interaction among Optimizations (4)

Dead (unreachable) code elimination yields:

```
x := 10;
y := x + 10;
```

And now we can do further constant folding!

```
x := 10;
y := 20;
```

And then, if `x` never used again, more dead code elimination!

```
y := 20;
```

Interaction among Optimizations (5)

- In general hard to pick a “best” order among the optimizations.
- Compilers often carry out optimizations iteratively until no further improvements can be made.