

COMP3320/COMP6464: Optimizing Compilers

Alistair Rendell

"In order to write high level language code that produces efficient machine code, you really need to understand how compilers and linkers translate high-level source statements into executable machine code"

Write Great Code, Vol 2, Randall Hyde, Chapter 5

See also *High Performance Computing*, Dowd and Severance, Chapter 5

COMP3320 Lecture 10-0 Copyright © 2010 The Australian National University

10.1 Compilers

Aim: To convert the Higher Level Language (HLL) to machine code with:

- Correct answers (although accuracy may be affected)
- Fast code

Use:

- **No optimisation:** just to make the code work!
- **Basic optimisation:** when code is working, generally decreases binary size
 - simplifies code
 - throws away extraneous computations
 - shares intermediate results
- **Advanced optimisation:** to gain speed, may increase binary size
 - move/restructure the code

COMP3320 Lecture 10-1 Copyright © 2010 The Australian National University

10.2 Architectural Issues

- Number of registers
- Number, size (total and line), and associativity of caches
- Rules for issuing instructions (grouping)
- Cost of instructions (clock ticks)
- Latency of instructions (when results appear)
- What operations can be combined (eg add with mult)

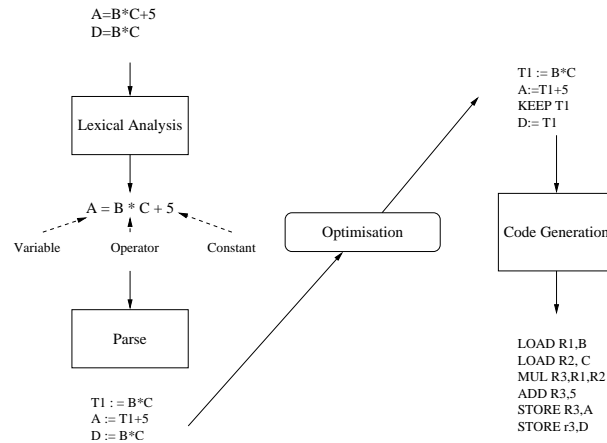
COMP3320 Lecture 10-2 Copyright © 2010 The Australian National University

10.3 The Compilation Process

- Precompilation/Preprocessing phase (eg CPP preprocessor)
- Lexical analysis: source translated into tokens such as variables, constants, comments, or language elements
- Parsing phase: input checked for syntax and translated into *intermediate language* (something between the HLL and assembler)
- Intermediate language optimisation: basic block analysis etc
- Object code generator: generation of assembler, register assignment, pipelining

COMP3320 Lecture 10-3 Copyright © 2010 The Australian National University

10.4 Basic Compilation Process



10.6 Intermediate Language: Example

```

do while (j < n)
  k = k + j * 2
  m = j * 2
  j = j + 1
enddo

A:: t1 := j
    t2 := n
    t3 := t1 < t2
    jmp (B) t3

    jmp (C) TRUE

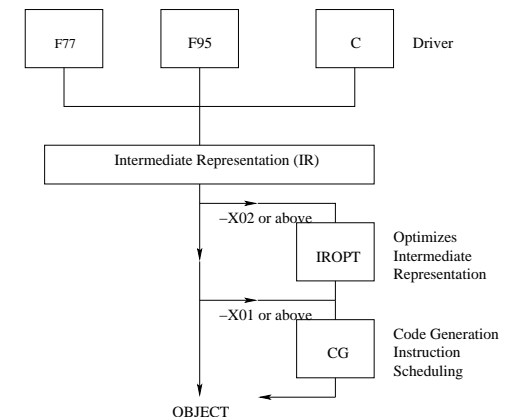
B:: t4 := k
    t5 := j
    t6 := t5 * 2
    t7 := t4 + t6
    k := t7
    t8 := j
    t9 := t8 * 2
    m := t9
    t10 := j
    t11 := t10 + 1
    j := t11
    jmp (A) TRUE

C::
  
```

10.5 The Intermediate Language

- Usually common to Fortran, C, C++ compilers
- Incorporates the core features of the (RISC) machine, eg
 - instructions consist of one operation and at most two operands
 - all memory references are explicit load from or store to "temporaries" (registers)
 - logical values used in branches are calculated separately from actual branch
 - jumps go to absolute addresses (or labels)
- Intermediate code then cut into basic blocks:- portions of code with one entry (at the top) and one exit (at the bottom).
 - jumps are made from the end of one basic block to the beginning of another.
- Optimisation is then either within or between basic blocks

10.7 The Sun Compilation Process



10.8 Optimisation Levels

No optimisation: Translate exactly into machine code. Maybe very slow but should be correct! Primarily for debugging.

Basic Optimisation: Constant folding, strength reduction etc as discussed on following slides

Interprocedural analysis: Optimisation beyond the boundary of a single routine, eg inline a function

Floating point analysis: IEEE 754 specifies precisely how floating point operations are performed and their side effect. Compiler may relax this to increase speed, eg replace divide by reciprocal and multiply

Runtime profile analysis: use information from a previous profile to enhance optimisation

Dataflow analysis: identify potential areas that can be run in parallel

Sun compiler has 5 levels of optimisation. Currently "-O" defaults to "-O3". Do "man cc" for details.

10.9 Compiler Optimisation: Basic

1. **Constant Folding:** evaluate expr'ns as much as possible at compile time

eg. given: `#define CACHESIZE 8192`
`n = CACHESIZE / 8; ⇒ n = 1024;`

2. **Dead Code Removal:** Removal of code that has no effect on answers

```
main(){
    int i,k;
    i = k = 1;
    i += 1;
    k += 2;
    printf("%d\n",i);
}
```

Be very careful when benchmarking!

Compiler Optimisation: Basic#2

3. **Strength Reduction:** replace expensive op'n with an equivalent cheaper one

eg. `k * CACHESIZE ⇒ ISHFT(k,13)`

eg. `y = x**2 ⇒ y = x*x`

4. **Common Sub-expression Elimination**

- Can recognize these & evaluate only once
- One of the most important: can easily arise through macros eg. given:

```
#define CUBE(x) ((x)*(x)*(x))
x = CUBE(a(i)/b(i)); ⇒
t = a(i)/b(i); x = t*t*t;
```

- Can often work on larger expr'ns within a block of code (provided not too complex...)

Compiler Optimisation: Basic#3

5. **(loop) Induction Variable Simplification:** Like 3, but uses the context of a loop. Especially important for RISC processors. Eg

```
for(i=0; i<N; i++){c[(i-1)*LdC+j] = c[(i-1)*LdC+j] + a[i] * (b[j] + 1.0);}
```

⇒

```
t = 0
for(i=0; i<N; i++){
    // t = (i-1)*LdC; invariant
    c[t+j] = c[t+j] + a[i] * (b[j]+1.0);
    t = t + LdC;
}
```

6. **Loop Invariant Code Motion:** detect which calculations are the same on each iteration, do once before loop starts, eg (previous example) ⇒

```
t = 0; bj = b[j]+1.0;
for(i=0; i<N; i++){
    c[t+j] = c[t+j] + a[i] * bj;
    t = t + LdC;
}
```

Compiler Optimisation: Basic#4

6. Register Variable Detection: which variables can (safely) be kept in registers

- Can reduce load/stores, eg

```
for(i=0; i<N; i++){c[j] = c[j] + a[i] * b;}
```

⇒

```
register double rc; // implement as a register
```

```
rc = c[j];
```

```
for(i=0; i<N; i++){
```

```
    rc = rc + a[i] * b;
```

```
}
```

```
c[j] = rc;
```

10.10 Compiler Optimisation: Advanced

7. **Optimal instr'n scheduling** (*Software Pipelining*: aim to overlap instructions so they can be run in parallel given a superscalar machine architecture - more in next lecture). This occurs through the analysis of dependencies (also the basis of vectorizing and ||izing compilers):

- if event A must occur before B , there is a dependency between B and A

- ie. B cannot occur in || with A (a 'barrier' to ||ism)

- flow dependencies: B reads variable set by A , eg

```
A = X + Y  
B = A + C
```

- anti-dependencies: B over-writes a variable that A uses (B cannot precede A)

- can generally eliminate by variable re-naming eg

```
X = A/B           | Xt = A / B  
Y = X + 2.0       | Y = Xt + 2.0  
X = D - I         | X = D - I
```

10.11 Compiler Optimisation: Advanced#2

8. Procedure inlining (for small, internal procedures)

- Remove overhead of procedure call (ie a jump in instruction stack)
- Procedures in loops inhibit pipelining

9. Loop unrolling (innermost)

- Reduce overheads of loop
- Increase possibility for software pipelining

10.12 Compiler Optimisation: Comments

Do *not* expect transformations that:

- May expose lots of ||ism, eg. outer loop unrolling, loop interchange
- Optimize memory access patterns
- Require run-time knowledge of the important execution conditions

This is partly because compilers must not make optimizations that are unsafe, ie must preserve serial nature of program (even if not intended by the programmer...)

- eg. (in C) `for (i=0; i<N; i++) y[i] = y[i] + a * x[i];`

?? ⇒??)

```
for (i=0; i<N; i+=2) {  
    double x0 = x[i], x1 = x[i+1];  
    y[i] = y[i] + a * x0; y[i+1] = y[i+1] + a * x1; }  
ie. is memory overlapping - illegal in FORTRAN!
```

10.13 Compiler Optimisation: Comments#2

- **Ambiguous memory references:** cannot determine whether dependency exists at compile-time

- eg. (inter-iteration) flow dependency if $k = j$

```
for (i=1; i<N; i++){  
    X[i+1][j] = X[i][k] + y[i];  
}
```

must in general be treated conservatively!

- Also compiler must:
 - Consider if it will always speed it up?
 - Balance optimizations with potential code bloat, ie. a space-time tradeoff

10.14 Compiler Optimisation: Summary

- Compiler optimizations can have dramatic effects
- May need some help from programmer, esp. to decide what optimizations are important
- May require code changes to help the compiler to generate fast code.