CME 212: Optimization

Code optimization:

- what compilers try to do
- what you can do
- details related to C
- profiling

Overview of optimization

The compiler is a fundamental tool:

- performs several types of optimization
- in general, produces far better assembly code than people
- knows the hardware to some extent

Modern hardware attempts to run code as fast as possible with:

- pipelining
- instruction level parallelism
- out-of-order execution
- caching in fast memory

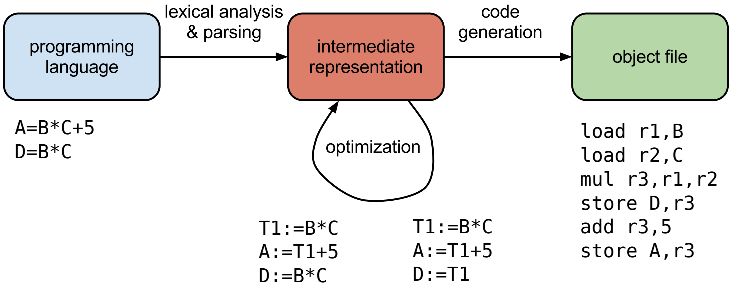
Programmer's job:

- first make the program work
- write code in a way that does not block efforts of compiler and CPU
- use profiler and spend time optimizing inner loops

Keep in mind

- you have to make the program work first, this is most important
- Selecting a faster algorithm will do more than optimizing a slow algorithm. An algorithm that scales linearly with input data will do better than one that scales quadratically
- code readability is extremely important, focus on this for all but computationally intensive loops
- most of the time we just try to avoid certain blockers that limit compiler optimization and hardware features

The compilation process



```
A=B*C+5
D=B*C
```

```
T1:=B*C
A:=T1+5
D:=B*C
```

```
T1:=B*C
A:=T1+5
D:=T1
```

```
load r1,B
load r2,C
mul r3,r1,r2
store D,r3
add r3,5
store A,r3
```

Figure: high level view of compilation process

GCC & IR

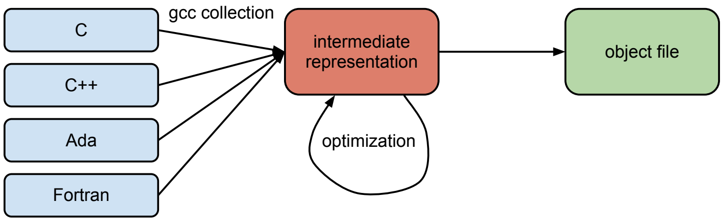


Figure: the gnu compiler collection coverts many front-end languages to an intermediate representation for analysis and optimization

Intermediate representation

IR looks a lot like assembly code, however it keeps more information like definitions and uses of variables. In assembly this information is handled as registers and addresses. A simple model of IR consists of a stream of *quadrupals*:

$$(op, o_1, o_2, r)$$

With:

- $op =$ an operation, like $\times$, $+$, `store`, `load`
- o_1 and o_2 are operands
- r result

Code transformation

Complex statements and expressions need to be broken down into "atomic" components. This often requires temporary variables. For example,

```
A = -B + C * D / E
```

could be converted to the sequence

```
T1 := D / E
T2 := C * T1
T3 := -B
A  := T3 + T2
```

now each statement fits into a quadrupal

Example from HPC book

Loop code:

```
while (j < n) {
  k = k + j * 2;
  m = j * 2;
  j++;
}
```

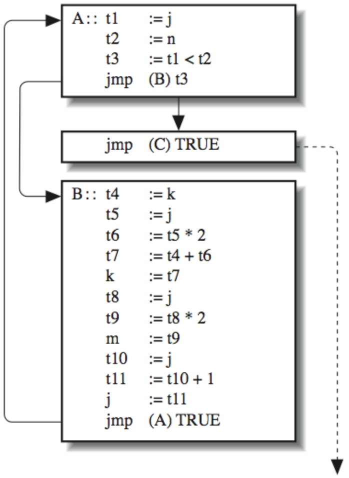
Intermediate representation

```
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::
```

Block structure

Loop code:

```
while (j < n) {
  k = k + j * 2;
  m = j * 2;
  j++;
}
```



Classical compiler optimizations

- Copy propagation
- Constant folding
- Dead code removal
- Strength reduction
- Variable renaming
- Common subexpression elimination
- Loop-invariant code motion
- Induction variable simplification

Copy propagation

Start with:

```
X = Y
Z = 1.0 + X
```

The second statement requires the result of the first, however this is an unnecessary dependency and the compiler can *propagate* a copy of `Y`:

```
X = Y
Z = 1.0 + Y // replace X with Y
```

Now the statements are independent and could be parallelized by the processor.

Constant folding

Start with:

```
const int i = 100;
int k = 200;
int j = i+k;
```

Here `i` is obviously a constant. A good compiler will notice that `k` does not change before `j` is defined, so it can just set `int j = 300`. The compiler will trace variable definitions back to the souce. It will do basic arithmetic based on what it sees. For another example, look at:

```
X = X * Y;
```

The compiler will generate different code if it can determine that `Y` is 0,1,2, or some other arbitrary number.

Dead code removal

Look at:

```
main () {
  int i,k;
  i = k = 1;
  i += 1;
  k += 2;
  printf ("%d\n",i);
  if ( 1==0 ) printf("I␣am␣not␣going␣to␣be␣printed\n");
}
```

Here, `k` is never used, so anything related to it can be thrown out. Also the second `printf` statement can never be called and can be discarded.

Note that certain steps in the compilation or optimization process may introduce dead code, so it is important to clean it up.

Strength reduction

It is sometimes possible to replace an expensive calculation with a cheaper one. Take a simple Fortran example:

```
REAL X,Y
Y = X**2
J = K*2
```

- In Fortran `**` is the exponentiation operator. The compiler can replace the second statement with `Y = X*X`, which is much cheaper.
- In C we have to use the `pow` function in `math.h`.
- For the third statement, the compiler can do `J = K+K`, and avoid a multiplication.

Variable renaming

The compiler may rename variables to keep statements separate. Start with:

```
x = y * z;
q = r + x + x;
x = a + b;
```

Rename x with x0 in the first two statements:

```
x0 = y * z;
q = r + x0 + x0;
x = a + b;
```

Now the third statement is independent of the other two.

Common subexpression elimination

The following has a common subexpression `(A+B)`:

```
D = C * (A + B)
E = (A + B)/2.0
```

The compiler could find this and produce:

```
temp = A + B
D = C * temp
E = temp/2.0
```

Note:

- compiler can only go so far. In FP $A + B + C \neq B + C + A$.
- common subexpressions often show up in address calculations for array elements!

Loop-invariant code motion

Move expressions that are independent of the loop outside. Start with:

```
for (int i=0; i != n; i++) {
  a[i] = b[i] + c * d;
  e = g[k];
}
```

Move loop-invariant expressions outside:

```
temp = c * d;
for (int i=0; i != n; i++) {
  a[i] = b[i] + temp;
}
e = g[k];
```

Again, this is important for address arithmetic.

Induction variable simplification

Induction variables change as a linear function of the loop iteration counter. For example

```
for (int i=0; i != n; i++) {
  k = i*4 + m; // at least 3 instructions
  ...
}
```

could be simplified to

```
k = m-4;
for (int i=0; i != n; i++) {
  k += 4; // one instruction
  ...
}
```

This is exactly what's done for the address calculation involved in array indexing.

Optimization in `gcc`

Some tidbits from the online documentation:

- **Without any optimization** option, the compiler's goal is to **reduce the cost of compilation** and to **make debugging produce the expected results**.
- **Turning on optimization** flags makes the compiler attempt to improve the performance and/or code size at the **expense of compilation** time and possibly the **ability to debug the program**.
- Most optimizations are only enabled if an `-O` level is set on the command line. Otherwise they are disabled, even if individual optimization flags are specified.

`gcc` optimization levels

Different optimization levels are access with different `-0#` flags. For example

```
$ gcc -02 ...
```

The flags are

- `-O0`: default, extremely limited optimization, minimize compilation time and produce correct debugging results
- `-O` or `-O1`: reduce code size and execution time without large increase in compilation time
- `-O2`: optimize more
- `-O3`: optimize most! Will increase code size with function inlining
- `-Os`: optimize for size
- `-0fast`: disregard standards compliance (IEEE FP)

gcc optimization flags

The `-O#` flags control a set of `-fflag` flags. See the documentation for more details. For example:

```
-fforward-propagate
```

> *Perform a forward propagation pass on RTL. The pass tries to combine two instructions and checks if the result can be simplified. If loop unrolling is active, two passes are performed and the second is scheduled after loop unrolling.*

Different installs and versions may do different things. To figure out what is happening on your system, see

```
$ gcc -Q --help=optimizers -O1
```

Basic profiling with `gprof`

Let's see a simple example of `gprof` with code that I used to time $y = \exp(x)$. To enable profiling, use the `-pg` flag:

```
$ make run_gettimeofday
gcc -Wall -std=c99 -lm -O1 -pg \
    -c -o vec.o vec.c
gcc -Wall -std=c99 -lm -O1 -pg \
    run_gettimeofday.c vec.o   -o run_gettimeofday
```

Now run the program and produce the profiling report:

```
$ ./run_gettimeofday
$ gprof run_gettimeofday > report.txt
```

The report

It starts with a flat profile:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 94.91      1.05     1.05      800     1.32     1.32  vec_exp
  5.42      1.11     0.06        8     7.53     7.53  vec_range
  0.00      1.11     0.00      800     0.00     0.00  elapsed_time
  0.00      1.11     0.00       24     0.00     0.00  vec_alloc
  0.00      1.11     0.00       24     0.00     0.00  vec_free
  0.00      1.11     0.00        8     0.00   139.21  run_test
  0.00      1.11     0.00        8     0.00     0.00  vec_max
  0.00      1.11     0.00        8     0.00     0.00  vec_mean
  0.00      1.11     0.00        8     0.00     0.00  vec_min
  0.00      1.11     0.00        8     0.00     0.00  vec_sum
```

and follows with more information. . .

Call graphs

One useful thing to do is look at a call graph:

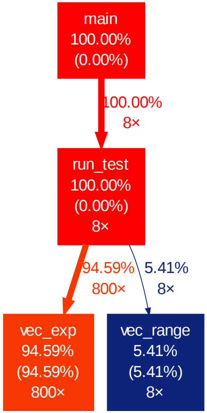


Figure: call graph generated with `gprof2dot.py` and `graphviz`

Utilities

Graph visualization software:

`www.graphviz.org`

Python script to construct visual call graph from `gprof` output:

`code.google.com/p/jrfonseca/wiki/Gprof2Dot`

Sharp profile

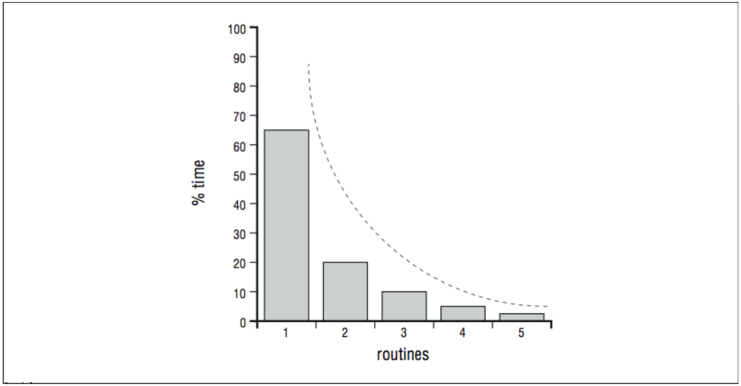


Figure: An example of a sharp profile. Work is dominated by a single routine.

Flat profile

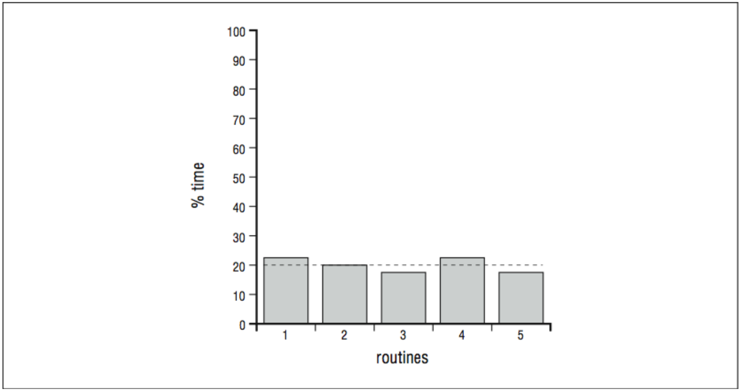


Figure: An example of a flat profile. Work is spread evenly through all routines.

Speed-up

The term **speed-up** usually refers to the quantity

$$s = \frac{\text{old time}}{\text{new time}}.$$

So if you manage to half the runtime of a program you get

$$s = \frac{100 \text{ sec}}{50 \text{ sec}} = 2.$$

We'd say that's a 2 *times* speed-up.

Amdahl's law

Amdahl's law can tell us the expected overall speed-up of a program due to the speed-up of a component. Say P is the fraction of time taken by the unoptimized component. Let S be the speed-up for the component after optimization. The over all speed-up is

$$s_T = \frac{1}{(1-P) + \frac{P}{S}}.$$

For example say a component takes 50% of the program time. After some work we can speed that component up 2 times. Then

$$s_T = \frac{1}{(1-.5) + \frac{.5}{2}} = 1.33.$$

This is useful in figuring out what to optimize and how much speed-up to expect.