UPPSALA UNIVERSITET

Methods of Programming DV2

Optimisation and profiling

Efficient programs

- Efficiency is not a prime concern when you start development.
- Good code is!
- There is a tradeoff between clear and efficient code.
- It is difficult to know in advance where the bottlenecks are.
- Don't spend extra time writing efficient code that doesn't need to be efficient.

What if it turns out that the program does need to go faster?

- Buy a faster computer!
  (Seriously. Hardware is cheap. Programmers are expensive.)
- Turn on compiler optimisations

Optimisation flags

…for the SUN C-compiler (cc).

`-O[1|2|3|4|5]`

1 Do basic local optimization (peephole).

2 Do basic local and global optimization.

…induction variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination and complex expression expansion.

In general, the `-O2` level results in minimum code size.

3 ...this also optimizes references and definitions for external variables. Loop unrolling and software pipelining are also performed.

4 ...this also does automatic inlining of functions contained in the same file; this usually improves execution speed.

5 Generate the highest level of optimization, suitable only for the small fraction of a program that uses the largest fraction of computer time.

Uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time. Optimization at this level is more likely to improve performance if it is done with profile feedback.

What to improve?

Suppose you do need to modify the code to make it go faster.

- Improve the code
- Change data structures
- Change algorithms
- Change program structure

...but make sure you know *where* the bottlenecks are.

Get statistics!

Compilers and other tool can help you with *profiling*.

1) How often are different parts of the program executed?
2) How much time is spent in different parts of the program?

Hardware simulators can also be used! (E.g. Virtutech's SIMICS.)

Profiling on SUN Unix systems

"Flat" profiling at the procedural level:

```
cc -p program.c -o program
./program              (creates the data file mon.out)
prof program           (generates the report)
```

"Hierarchical" profiling at the procedural level:

(works with `gcc` as well, using the option `-pg`)

```
cc -xpg program.c -o program
./program              (creates the data file gmon.out)
gprof program          (generates the report)
```

Profiling at the basic block level:

```
cc -xa program.c -o program (creates the data file prog.d)
./program              (updates the data file prog.d)
tcov program.c         (generates the report file prog.tcov)
```


Programs for profiling examples

```
void bar(){
  int i;
  for(i=1; i <= 100; i++) {
  };
}
void foo(){
  bar();
}
void main(){
  int i,j;
  for(j=1; j<=100000; j++) {
    foo();
    for(i=1; i<= 10; i++)
      bar();
  };
  return;
}
```

Flat profiling with `gprof`

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 72.6      0.53     0.53  1100000     0.00     0.00  bar [3]
  9.6      0.60     0.07           prof_call_graph_lookup_edge [4]
  8.2      0.66     0.06                    mcount_single [5]
  6.8      0.71     0.05                    _mcount (2239)
  2.7      0.73     0.02        1    20.00   550.00  main [1]
  0.0      0.73     0.00   100000     0.00     0.00  foo [6]
```

```
void bar(){
  int i;
  for(i=1; i <= 100; i++) {
  };
}
void foo(){
  bar();
}
void main(){
  int i,j;
  for(j=1; j<=100000; j++) {
    foo();
    for(i=1; i<= 10; i++)
      bar();
  };  return;  }
```


Effect of changing runtime distribution

Note that the profiler *samples* the running program.

```
(j <= 10000)
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.0      0.04     0.04   110000     0.00     0.00  bar [3]
 25.0      0.06     0.02                    _mcount (2239)
 12.5      0.07     0.01        1    10.00    50.00  main [1]
 12.5      0.08     0.01                prof_call_graph_lookup_edge [4]
  0.0      0.08     0.00    10000     0.00     0.00  foo [5]

(j <= 100000)
 72.6      0.53     0.53  1100000     0.00     0.00  bar [3]
  9.6      0.60     0.07                prof_call_graph_lookup_edge [4]
  8.2      0.66     0.06                    mcount_single [5]
  6.8      0.71     0.05                    _mcount (2239)
  2.7      0.73     0.02        1    20.00   550.00  main [1]
  0.0      0.73     0.00   100000     0.00     0.00  foo [6]

(j <= 1000000)
 74.0      5.47     5.47 11000000     0.00     0.00  bar [3]
 11.8      6.34     0.87                    mcount_single [4]
 10.3      7.10     0.76                prof_call_graph_lookup_edge [5]
  1.9      7.24     0.14        1   140.00  5630.00  main [1]
  1.8      7.37     0.13                    _mcount (2239)
  0.3      7.39     0.02  1000000     0.00     0.00  foo [6]
```


Hierarchical profiling with gprof

```
                                called/total       parents
index  %time    self descendents  called+self    name           index
                                 called/total       children
                0.01        0.61       1/1           _start [2]
[1]     84.9    0.01        0.61       1         main [1]
                0.55        0.00 1000000/1100000      bar [3]
                0.01        0.05  100000/100000       foo [4]
-----------------------------------------------
                                                  <spontaneous>
[2]     84.9    0.00        0.62                  _start [2]
                0.01        0.61       1/1            main [1]
-----------------------------------------------
                0.05        0.00  100000/1100000      foo [4]
                0.55        0.00 1000000/1100000      main [1]
[3]     82.2    0.60        0.00 1100000          bar [3]
-----------------------------------------------
                0.01        0.05  100000/100000       main [1]
[4]      8.8    0.01        0.05  100000          foo [4]
                0.05        0.00  100000/1100000      bar [3]
-----------------------------------------------
                                                  <spontaneous>
[5]      8.2    0.06        0.00                  mcount_single [5]
-----------------------------------------------
                                                  <spontaneous>
[6]      6.8    0.05        0.00            prof_call_graph_lookup_edge [6]
```


Basic block profiling using `tcov`

```
            void bar(){
              int i;
1100000 ->    for(i=1; i <= 100; i++) {
              };
            }
            void foo(){
 100000 ->    bar();
            }
            void main(){
              int i,j;
 100001 ->    for(j=1; j<=100000; j++) {
 100000 ->      foo();
                for(i=1; i<= 10; i++)
1000000 ->        bar();
              };
      1 ->    return;
            }
```


Basic block profiling using `tcov` (cont'd)

```
                Top 10 Blocks

                Line      Count

                   3    1100000
                  14    1000000
                  11     100001
                   7     100000
                  12     100000
                  16          1


         6   Basic blocks in this file
         6   Basic blocks executed
    100.00   Percent of the file executed

   2400002   Total basic block executions
 400000.34   Average executions per basic block
```


How to use profiling

Choose different kind/sizes of test data to see if the bottlenecks differ and what the time complexity of the program is.

If profiling shows that a few parts of the program dominates execution time, it clear where to put your work.

If no part of the program dominates, you will have to try a ”holistic” approach – more difficult but not impossioble.

How to improve?

Aim at improving performace without sacrificing readbility

- Understand your problem (and program)
- Don't be afraid of throwing code away
- Don't do unnecessary computations
- Use the processor efficiently

Code level optimisation

Avoid breaking processor pipelines:

- Avoid procedure calls and jumps – particularly conditional jumps.

Avoid cache misses:

- Use small amounts of code in the body of inner loops – avoid procedure calls.
- Data which is used together should be kept together

Avoid redundant computations:

- Make tests for unlikely cases after tests for likely cases
- Don't compute a value unless you know it will be used
- Don't make invariant computations inside a loop
- "Strength reduction" in loops.

Programs for optimisation examples

```
#define X_SIZE 30
#define Y_SIZE 60

int matrix[X_SIZE][Y_SIZE]

void init_matrix() {
  int x,y;
  for (y = 0; y < Y_SIZE; y++) {
    for (x = 0; x < X_SIZE; x++) {
      matrix[x][y] = 0;
    }
  }
}
```


Optimisation (1)

Put frequently used variables in registers:

```
#define X_SIZE 30
#define Y_SIZE 60

int matrix[X_SIZE][Y_SIZE]

void init_matrix() {
  register int x,y;
  for (y = 0; y < Y_SIZE; y++) {
    for (x = 0; x < X_SIZE; x++) {
      matrix[x][y] = 0;
    }
  }
}
```


Optimisation (2)

- Put the loop with the most iterations innermost
- Step the array pointer one step at a time. Improves cache hits, may improve code — consider that `matrix[x][y] = 0` by definition is the same as `*(matrix+(x*Y_SIZE)+y) = 0`

```
#define X_SIZE 30
#define Y_SIZE 60

int matrix[X_SIZE][Y_SIZE]

void init_matrix() {
  register int x,y;
  for (x = 0; x < X_SIZE; x++) {
    for (y = 0; y < Y_SIZE; y++) {
      matrix[x][y] = 0;
    }
  }
}
```


Optimisation (3)

- Multiplying by a power of 2 can be done by shifting bits. This may be faster than regular multiplication.

```
#define X_SIZE 30
#define Y_SIZE 64

int matrix[X_SIZE][Y_SIZE]

void init_matrix() {
  register int x,y;
  for (x = 0; x < X_SIZE; x++) {
    for (y = 0; y < Y_SIZE; y++) {
      matrix[x][y] = 0;
    }
  }
}
```

Optimisation (4)

• Strength reduction – replace multiplication by repeated addition

```
#define X_SIZE 30
#define Y_SIZE 64

int matrix[X_SIZE][Y_SIZE]

void init_matrix() {
  register int x,y;
  register int *matrix_ptr;
  matrix_ptr = &matrix[0][0];
  for (x = 0; x < X_SIZE; x++) {
    for (y = 0; y < Y_SIZE; y++) {
      *(matrix_ptr+y) = 0;
    };
    matrix_ptr = matrix_ptr + Y_SIZE;
  }
}
```


Optimisation (5)

• Merge the loops

```
#define X_SIZE 30
#define Y_SIZE 64

int matrix[X_SIZE][Y_SIZE]

void init_matrix() {
  register int y;
  register int *matrix_ptr;

  matrix_ptr = &matrix[0][0];
  for (y = 0; y < X_SIZE * Y_SIZE; y++) {
    *(matrix_ptr+y) = 0;
  }
}
```


Optimisation (6)

• Remove redundant variables

```
#define X_SIZE 30
#define Y_SIZE 64

int matrix[X_SIZE][Y_SIZE]

void init_matrix() {
  register int *matrix_ptr;

  for (matrix_ptr = &matrix[0][0];
       matrix_ptr <= &matrix[X_SIZE-1][Y_SIZE-1];
       matrix_ptr++) {
    *matrix_ptr = 0;
  }
}
```


Optimisation (7)

• Use specialised library routines. They are usually well optimised and may even be written in assembler language and utilise the processor better that the compiler is able to.

```
#include <memory.h>
#define X_SIZE 30
#define Y_SIZE 64

int matrix[X_SIZE][Y_SIZE]

void init_matrix() {
  memset(matrix, 0, sizeof(matrix));
}
```


Optimisation (8)

• Use inlining to remove procedure call overhead

```
include <memory.h>
#define X_SIZE 30
#define Y_SIZE 64

int matrix[X_SIZE][Y_SIZE]

#define init_matrix() \
  memset(matrix, 0, sizeof(matrix));
```