Real-time Operating Systems and Systems Programming

Optimization

ARVUTITEADUSE INSTITUUT
DEPARTMENT OF COMPUTER SCIENCE

Optimizing compilers

- Same code has different representations
- Some are more efficient (yet less readable)
- Handrwitten assembler code is often optimal

How to optimize: use -O option for gcc

Why not the default option?

Limitations

- Never alter the correct program behaviour
- Their understanding of program behaviour is limited
- Compilation must be fast

Optimization blockers

```
void foo1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;
}
void foo2(int *xp, int *yp)
{
    *xp += 2* *yp;
}
```

- Similar code
- First uses 6 memory references, second 3
- Would be possible to optimize?
- What happens if pointers are equal?

Optimization blockers

```
int f(int);
int func1(x) {
    return f(x) + f(x) \
                + f(x)+ f(x);
}

int func2(x) {
    return 4*f(x)
}
```

- func2() faster
- but only in case f() is without side effects
- Usually not tested

Program performance assessment

- Speed of processors can vary
- Useful measure for examples: *cycles per element.*
- What is the code overhead for any array element

Base example

```
typedef struct {
      int len;
      data_t *data;
} vec_rec, *vec_ptr;

typedef int data_t; //or float for experiments

#define IDENT 0 // or 1
#define OPER + //or *

// actual implementation of vectors less interesting
```

Base implementation

```
void combine1(vec_ptr v, data_t *dest) {
      int i;
      *dest = IDENT;
      for (i = 0; i < vec_length(v); i++) {
            data_t val;
            get_vec_element(v, i, &val);
            *dest = *dest OPER val;
      }
}

//                              int          float
// unoptimized          +42 *41      +41 *160
// optimized -O2              +31 *33      +31 *143
```

Moving calculations from loop

```
void combine2(vec_ptr v, data_t *dest) {
      int i;
      int length = vec_length(v)
      *dest = IDENT;
      for (i = 0; i < length; i++) {
            data_t val;
            get_vec_element(v, i, &val);
            *dest = *dest OPER val;
      }
}

// old optimized -O2          +31 *33     +31 *143
// move vec_len               +22 *21     +21     *135
```

Reducing function calls

```
void combine3(vec_ptr v, data_t *dest) {
      int i;
      int length = vec_length(v)
      data_t *data = get_vec_start(v);
      *dest = IDENT;
      for (i = 0; i < length; i++) {
            *dest = *dest OPER data[i];
      }
}

// move vec_len                +22 *21     +21     *135
// direct data access          +6  *9   +8  *117

// Note: we gained speed by losing in abstraction & modularity
```

Decompilation analysis

Combine 3
dest in edi, data in ecx, i in edx, length in esi

```
.L18                                :loop
movl (%edi), %eax                   Read dest
imull (%ecx, %edx, 4), %eax         Multiply data
movl %eax, (%edi)                   Write *dest
incl %edx                       i++
cmpl %esi, %edx                     Compare
i:length
jl .L18                             if < goto loop
```

Combine 4
data in eax, x in ecx, i in edx, length in esi

```
.L24                                    :loop
imull (%eax, %edx, 4) %ecx              Multiply by data[i]
incl %edx                           i++
cmpl %esi, %edx                         Compare i:length
jl .L24                             If <, goto loop
```

Storage variable

```
void combine4(vec_ptr v, data_t *dest) {
      int i;
      int length = vec_length(v)
      data_t *data = get_vec_start(v);
      data_t x = IDENT;
      *dest = IDENT;
      for (i = 0; i < length; i++) {
            x = x OPER data[i];
      }
      *dest = x;
}

// direct data access          +6 *9   +8 *117
// temporary variable          +2 *4   +3 *5

// Why not automatic?
```

Different functions

- combine3(v, get_vec_start(v) + 2);
- combine4(v, get_vec_start(v) + 2);
- Last element used for destination

c3	c4
2 3 5	2 3 5
2 3 1	2 3 5
2 3 2	2 3 5
2 3 6	2 3 5
2 3 36	2 3 5
2 3 36	2 3 30

Aside: further optimizations

- Modern processors use pipelining, parallelization
- Can be used for advantage

Loop unrolling

```
void combine5(vec_ptr v, data_t *dest) {
      int i;
      int length = vec_length(v)
      int limit = length - 2;
      data_t *data = get_vec_start(v);
      data_t x = IDENT;
      *dest = IDENT;
      for (i = 0; i < limit; i += 3) {
            x = x OPER data[i] OPER data[i+1] OPER data[i+2];
      }
      for(; i < length; i++) {
            x = x OPER data[i];
      }
      *dest = x;
}

// temporary variable        +2  *4  +3  *5
// loop unroll     x3        +1.3 *4+3  *5
```


// How many unrollings optimal?

Pointer code

- Use pointer code for speedups

```
void combine4p(vec_ptr v, data_t *dest) {
      int length = vec_length(v)
      data_t *data = get_vec_start(v);
      data_t *dend = data +length;
      data_t x = IDENT;
      for (; data < dend; data++) {
            x = x OPER *data;
      }
      *dest = x;
}

// temporary variable          +2  *4   +3  *5
// pointer code              +3  *4   +3  *5
// Mostly useless here, but really depends on compiler/platform
// Readability often priority
```

Parallelism

- It's often good to parallelize code

```
void combine6(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    int limit = length - 1;
    data_t = *data = get_vec_start(v);
    data_t x0 = IDENT; data_t x1 = IDENT;
    int i;
    for(i = 0; i < limit; i+=2) {
        x0 = x0 OPER data[i];
        x1 = x1 IOPER data[i+1];
    }
    for(;i < length; i++) {
        x0 = x0 OPER data[i];
    }
    *dest = x0 OPER x1;
}
// loop unroll     x3          +1.3 *4+3  *5
// parallelize by 2            +1.5 *2+2  *2.5
```

End results

- For most things unroll x8, parallel x4 is best
- For integer addition, best unroll x16
- **on Pentium III**

- Your results would be different

Less predictable features

- Cached data
- Load/store latency
- Branch prediction (predictive execution)

Cache

```
struct a {
    int a;
    int b;
    int c;
    int d;
};
```

```
struct a {
    int a;
    int b;
};
```

This is faster!

Row vs Column based access

- For two-dimensional arrays, visit by rows, not by columns!

What to do in real-life?

- High level design: choose appropriate algorithms and data structures.
- Basic coding principles
  - Eliminate exessive function calls, move computations out from loops, compromise on modularity if priority
  - Eliminate unnecessary memory references. Use temporary variables to hold intermediate results. Store results only when final value calculated

Real-Life 2

- Low level optimizations
  - Try different pointer-array code
  - Reduce loop overhead by unrolling them
  - Pipelined architecture: Find ways to split iterations when needed
- Avoid introducing errors by unittesting. Benchmark to find anomalies

GCC Options for Optimization

- -O0 – "don't optimize, keep code for debug"
- -O1 – "optimize, but don't spend time on it"
- -O2 – "optimize, but don't trade speed for size"
  - compilation will take more time
- -O3 – "optimize well"
- -Os – "optimize for (code) size"
- -Ofast – "O3 + break standards"
- -Og – "optimize, but let me debug"

Profiling

- Gprof
  - calculates cpu time for programs
  - counts function calling
  - thus gives data on what to optimize for greater gain
- Use -pg option for compiling on gcc
  - runs slower due to data collection, gmon.out file
- gprof prog
  to analyze (gives a table of functions, times, etc)

Code Efficiency

- To avoid macro definitions, C99 has keyword: inline
- When checking for alternatives, use switch carefully:
  - Put more popular cases first
  - Use function pointer arrays
- Inline assembly
- Gloal variables (but maintenance nightmare)

Code Efficiency

- Fixed-vs-floating point: former is faster
  - Small amount of decimal places: val << 2
- Use native word size (bus + registers are faster)

Code Size

- Standard library routines refer to other functions. Write your own printf()
- Goto is bad, yet more efficient for jumping out of nested loops than having to check a variable.

```
// NOTE: this example shows an error handling pattern, not
optimization
Int fun(void) {
      /* working */
      goto CLEANING; /* in case of error */
      /* more work */
      return SUCCESS;
CLEANING:
      /* cleanup here */
      return FAILURE;
}
```

Memory Usage

- Reduce dependence on stack & heap by using ROM for constant values (declare them as const)
- Some constants change: use flash memory & technicians
- Stack space estimation: fill memory with some pattern; check changes after running

Power-saving

- Necessary for battery-powered devices
- Processor modes (PXA255 example)
  - Turbo – minimize memory access due to waiting
  - Run – default mode
  - Idle – processor not clocked, peripherals operate
  - Sleep – lowest power state
- Clock frequency – tricky, needs HW knowledge
- Reduce external memory access – cache, processor memory

Optimization problems

- Dead code elimination
  - Declare variables as volatile
- Debugging more difficult: breakpoints missing, functions split and code different