Code optimization

Concept

An optimized program should run faster than the original one without giving out anything different under any circumstance.

Cycles per element, abbreviated CPE, is widely used to measure the performance of a program. In the experiments within this note, system function clock() is used to measure CPE.

Any optimization is bind to data structure and CPU design.

Experiment

I have picked some interesting optimization cases and carried out experiments on them.

The platform is Ubuntu 16.04 LTS on a MBP 2015, Intel Core i7-4870HQ CPU @2.5GHz, VMware Fusion.

1 Call reduction

The first one is quite simple and I basically use it to see whether the evaluating system works or not.

```c
#include <stdio.h>
#include <time.h>
#define BOUND1 1000000
void call(long i, long sum)
{
        for(i = 0;i <= BOUND1;i++)
        {
                sum++;
        }

}

long compCall()
{
        clock_t begin,end;
        long i, sum, time1, time2, t_result;

wser    begin = clock();
        for(i = 0;i <= BOUND1;i++)
        {
                sum++;
        }
        end = clock();
        time1 = end - begin;

        i = 0, sum = 0;

        begin = clock();
        call(i, sum);
        end = clock();
        time2 = end - begin;

        t_result = time2 - time1;
        printf("Clock cycles difference in compCall: %ld", t_result);
        return t_result;
}
```

clock() will return how many clock cycles the cpu has processed since the main() is obtained.

The result is about 10 clocks to 200 clocks.

It is not difficult to understand the consumption caused by function calls. Callq is one jump in disassemble code, which consumes more resource.

2 Loop unrolling

The basic idea of loop unrolling is to merge several loops into one (increasing the step at the same time, of course) to save the time that machine would take to judge whether the loop should be ended.

Merging k loops is called ‘k-1 unrolling’ .

Here is the code of combine1() and combine2(), you can see how combine3() and combine4() would look like.

```
#define BOUND1 12000
long combine1()
{
        int i;
        long result;
        for(i = 0;i <= BOUND1;i += 1)
        {
                result++;
        }
        return result;
}

long combine2()
{
        int i;
        long result;
        for(i = 0;i <= BOUND1;i += 2)
        {
                result++;
                result++;
        }
        return result;
}
```

compCall() run each of them for 100,000 times and average the clock cycles respectively.

```
long compCall()
{
        int i;
        clock_t begin,end;
        long time1, time2, time3, time4;
        for(i = 0; i <= 100000; i++)
        {
                begin = clock();
                combine1();
                end = clock();
                time1 += end - begin;

                begin = clock();
                combine2();
                end = clock();
                time2 += end - begin;

                begin = clock();
                combine3();
                end = clock();
                time3 += end - begin;

                begin = clock();
                combine4();
                end = clock();
                time4 += end - begin;
        }
        printf("Clock cycles in compCall: %lf, %lf, %lf, %lf",
                time1/100000.0, time2/100000.0, time3/100000.0, time4/100000.0);

        return 1;
}
```

The result is surprising.

Clock cycles in compCall: 28.102010, 24.636370, 23.869680, 66.110050

The difference among 1-1, 2-1, 3-1 unrolling indicates that unrolling strategy makes sense, while k = 4 case begs to differ.

Their disassemble codes are quite similar.

```
(gdb) disassemble combine3
Dump of assembler code for function combine3:
   0x00000000004005b5 <+0>: push   %rbp
   0x00000000004005b6 <+1>: mov    %rsp,%rbp
   0x00000000004005b9 <+4>: movl   $0x0,-0xc(%rbp)
   0x00000000004005c0 <+11>:    jmp    0x4005d5 <combine3+32>
   0x00000000004005c2 <+13>:    addq   $0x1,-0x8(%rbp)
   0x00000000004005c7 <+18>:    addq   $0x1,-0x8(%rbp)
   0x00000000004005cc <+23>:    addq   $0x1,-0x8(%rbp)
   0x00000000004005d1 <+28>:    addl   $0x3,-0xc(%rbp)
   0x00000000004005d5 <+32>:    cmpl   $0x2ee0,-0xc(%rbp)
   0x00000000004005dc <+39>:    jle    0x4005c2 <combine3+13>
   0x00000000004005de <+41>:    mov    -0x8(%rbp),%rax
   0x00000000004005e2 <+45>:    pop    %rbp
   0x00000000004005e3 <+46>:    retq
End of assembler dump.
(gdb) disassemble combine4
Dump of assembler code for function combine4:
   0x00000000004005e4 <+0>: push   %rbp
   0x00000000004005e5 <+1>: mov    %rsp,%rbp
   0x00000000004005e8 <+4>: movl   $0x0,-0xc(%rbp)
   0x00000000004005ef <+11>:jmp    0x400609 <combine4+37>
   0x00000000004005f1 <+13>:addq   $0x1,-0x8(%rbp)
   0x00000000004005f6 <+18>:addq   $0x1,-0x8(%rbp)
   0x00000000004005fb <+23>:addq   $0x1,-0x8(%rbp)
   0x0000000000400600 <+28>:    addq   $0x1,-0x8(%rbp)
   0x0000000000400605 <+33>:    addl   $0x4,-0xc(%rbp)
   0x0000000000400609 <+37>:    cmpl   $0x2ee0,-0xc(%rbp)
   0x0000000000400610 <+44>:    jle    0x4005f1 <combine4+13>
   0x0000000000400612 <+46>:    mov    -0x8(%rbp),%rax
   0x0000000000400616 <+50>:    pop    %rbp
   0x0000000000400617 <+51>:    retq
End of assembler dump.
```

The reason require more experiments.