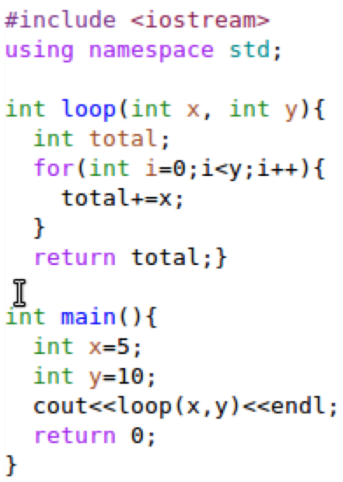
//Robert Alexander, RMA3MZ, inlab9.pdf

In-Lab 9 Report

Optimization Problem



```
#include <iostream>
using namespace std;

int loop(int x, int y){
  int total;
  for(int i=0;i<y;i++){
    total+=x;
  }
  return total;}

int main(){
  int x=5;
  int y=10;
  cout<<loop(x,y)<<endl;
  return 0;
}
```

For the optimization problem I created a small c++ program that runs a loop, adding the parameter x to a total sum y amount of times. I then print out the result in the main method. When looking at the normal code, I can almost completely follow what is happening, however, when looking at the optimized code, I quickly become confused by even the most simple aspects, such as a call. However, I will work through some of this confusion.

Main code optimized on right versus not optimized on the left:

```
main:                       # @main
        push    EBP
        mov     EBP, ESP
.Ltmp14:
        sub     ESP, 24
        mov     DWORD PTR [EBP - 4], 0
        mov     DWORD PTR [EBP - 8], 5
        mov     DWORD PTR [EBP - 12], 10
        mov     EAX, DWORD PTR [EBP - 8]
        mov     ECX, DWORD PTR [EBP - 12]
        mov     DWORD PTR [ESP], EAX
        mov     DWORD PTR [ESP + 4], ECX
        call    _Z4loopii
        lea     ECX, DWORD PTR [_ZSt4cout]
        mov     DWORD PTR [ESP], ECX
        mov     DWORD PTR [ESP + 4], EAX
        call    _ZNSolsEi
        lea     ECX, DWORD PTR
[_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES
6_]
        mov     DWORD PTR [ESP], EAX
        mov     DWORD PTR [ESP + 4], ECX
        call    _ZNSolsEPFRSoS_E
        mov     ECX, 0
        mov     DWORD PTR [EBP - 16], EAX # 4-
byte Spill
        mov     EAX, ECX
        add     ESP, 24
        pop     EBP
        ret
```

```
main:
        push    EBP
        mov     EBP, ESP
        push    EDI
        push    ESI
        sub     ESP, 16
.Ltmp9:
        mov     DWORD PTR [ESP], _ZSt4cout
        call    _ZNSolsEi
        mov     ESI, EAX
        mov     EAX, DWORD PTR [ESI]
        mov     EAX, DWORD PTR [EAX - 12]
        mov     EDI, DWORD PTR [EAX + ESI + 124]
        test    EDI, EDI
        je      .LBB1_5
# BB#1:
        cmp     BYTE PTR [EDI + 28], 0
        je      .LBB1_3
# BB#2:
        mov     AL, BYTE PTR [EDI + 39]
        jmp     .LBB1_4
.LBB1_3:
        mov     DWORD PTR [ESP], EDI
        call    _ZNKSt5ctypeIcE13_M_widen_initEv
        mov     EAX, DWORD PTR [EDI]
        mov     DWORD PTR [ESP], EDI
        mov     DWORD PTR [ESP + 4], 10
        call    DWORD PTR [EAX + 24]
.LBB1_4:
        movsx   EAX, AL
        mov     DWORD PTR [ESP + 4], EAX
        mov     DWORD PTR [ESP], ESI
        call    _ZNSo3putEc
        mov     DWORD PTR [ESP], EAX
        call    _ZNSo5flushEv
        xor     EAX, EAX
        add     ESP, 16
        pop     ESI
        pop     EDI
        pop     EBP
        ret
```

//Robert Alexander, RMA3MZ, inlab9.pdf

Comparison between main methods:

- The first thing I notice when adding the optimization is that the optimized code makes use of more registers as opposed to offsets of EBP. I am assuming that using more registers allows for faster access.
- The next thing I notice when comparing the two mains is that the call for the loop appears to be missing. After some general research online, I discovered that the loop call is being unwound, which causes the program to still be fairly long (150 lines for non-optimized vs 114 lines for optimized). Wikipedia states, “The goal of loop unwinding is to increase a program's speed by reducing (or eliminating) instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration.” This clearly explains the multiple small snippets of x86 code throughout the optimized .s file.

Unoptimized loop left, optimized right:

```
_Z4loopii:                  # @_Z4loopii
# BB#0:
        sub     ESP, 16
        mov     EAX, DWORD PTR [ESP + 24]
        mov     ECX, DWORD PTR [ESP + 20]
        mov     DWORD PTR [ESP + 12], ECX
        mov     DWORD PTR [ESP + 8], EAX
        mov     DWORD PTR [ESP], 0
.LBB1_1:                    # =>This Inner Loop Header: Depth=1
        mov     EAX, DWORD PTR [ESP]
        cmp     EAX, DWORD PTR [ESP + 8]
        jge     .LBB1_4
# BB#2:                     #   in Loop: Header=BB1_1 Depth=1
        mov     EAX, DWORD PTR [ESP + 12]
        mov     ECX, DWORD PTR [ESP + 4]
        add     ECX, EAX
        mov     DWORD PTR [ESP + 4], ECX
# BB#3:                     #   in Loop: Header=BB1_1 Depth=1
        mov     EAX, DWORD PTR [ESP]
        add     EAX, 1
        mov     DWORD PTR [ESP], EAX
        jmp     .LBB1_1
.LBB1_4:
        mov     EAX, DWORD PTR [ESP + 4]
        add     ESP, 16
        ret
```

```
_Z4loopii:                  # @_Z4loopii
# BB#0:                     # %.lr.ph
        ret

From Global:
# BB#0:
        push    EBP
.Ltmp15:
        .cfi_def_cfa_offset 8
.Ltmp16:
        .cfi_offset ebp, -8
        mov     EBP, ESP
.Ltmp17:
        .cfi_def_cfa_register ebp
        sub     ESP, 24
        mov     DWORD PTR [ESP], _ZStL8__ioinit
        call    _ZNSt8ios_base4InitC1Ev
        mov     DWORD PTR [ESP + 8], __dso_handle
        mov     DWORD PTR [ESP + 4], _ZStL8__ioinit
        mov     DWORD PTR [ESP],
_ZNSt8ios_base4InitD1Ev
        call    __cxa_atexit
        add     ESP, 24
        pop     EBP
        ret
```

Things start to get confusing quickly when comparing the unoptimized loop on the left to the optimized on the right. The first thing I notice is the unoptimzed code has four returns and the optimized has only three, probably due to the addition of the loop unwinding. Also, I notice that the loop function (_Z4Loopii) is never actually called in the optimized code. Given that it is unnecessary to call it because all it does is run the loop and return. If we are unwinding the loop, then the function no longer does anything of value, besides returning the total. I also notice that the loop in the unoptimzed code creates 16 bytes for local variables. I do not see this sort of allocation for local variables anywhere in the optimized code.

In conclusion, the optimized code, while faster, is extremely difficult to comprehend. Luckily, I was able to find more information about loop unwinding on Wikipedia, which ended up helping to explain some of the strange structure of the assembly code.

Sources:

- http://en.wikipedia.org/wiki/Loop_unwinding
- http://en.wikibooks.org/wiki/X86_Disassembly/Code_Optimization