On Optimizing Without the Little Man

Brian Earle and Allyson Cauble-Chantrenne

The code we decided to analyze is the matrix manipulation code in C that is attached. We chose this code because of the matrices and function calls utilized. The code prompts the user to input a matrix with a maximum size of fifty rows and fifty columns. It first asks for the size in terms of number of rows and columns, and then it asks for the elements. We store these elements in a two-demensional array by utilizing a nested for loop, storing in row-major order. Then, it gives the user a set of options that it can perform: addition, subtraction, and scalar multiplication. Using a switch case statement, the code performs the proper operations before calling the appropriate function. Finally the function called calls a print function to print out the result.

After compiling with and without the -O3 option, the most obvious difference between the two versions was the fact that the optimized version placed all of the functions before main, while the non-optimized version placed them after the main function. Because the functions were before main in the optimized .s file, the code could use them in a fashion similar to macros. While the non-optimized version called the add, subtract, mult, and print functions, the optimized one did not, and instead seemed to place slightly altered versions of said functions in the code for main. This makes sense because it does not require the stack manipulation necessary to make a function call, making the program run faster. One example of this is illustrated in the print function. This function is called no matter what option the user chooses when prompted, but instead of calling the function, the optimized version chose to paste the code for this function three different times within main. As a result, there is significantly more code in the optimized .s file than in the non-optimized one. However, the extra code does not translate to more runtime, but rather to less runtime due to the quick access to the print function available in the optimized version.

At the beginning of the main function, the non-optimized code, we found that very few registers were saved. The optimized code, however, pushed three extra registers onto the stack, implying that it would utilize these registers to avoid accessing memory as often. We also noticed that global variable constants were stored in different ways. The non-optimized code chose to save it to a register, and then pushing the register a few lines later, while the optimized code skipped the extra register and stored the constant directly onto the stack. This occured every time a call to printf was made. For example, for the non-optimized version, the code before the first printf looked like:

```
movl $.LC0, %eax
movl %eax, (%esp)
call printf
```

This shows the EAX register as being a sort of "middle man," allowing quick access to the constant later in the code, without the access of memory. On the other hand, a call to printf in the optimized code looked like:

```
movl $.LC1, (%esp)
call printf
```

In this version, the global variable is pushed straight onto the stack as the argument to printf, ignoring the "middle man", which is just an extra move that slows the program down. Two instructions that were included in the optimized version, but not the other, were testl and .p2align. The former is a means of replacing cmpl when checking to see if a register is zero. Testl is equivalent to performing an andl instruction, and in this situation, the optimized code uses it with one register as both operands. This allows it to check for zero since the zero flag will be set if the result of an andl is zero, which would occur only if the operand is zero. Yet testl is faster than andl because it only sets EFLAGS and does not change the operands themselves. The lines .p2align 4,,7 and .p2align 3 appeared after certain sections of code because the optimized version makes sure that the address it uses are multiples of eight. This is to help improve cache hits, in turn making the program run faster.

Another way the optimized version attempts to speed up the program is to make the for loops more efficient. Right before the for loops in the non-optimized version, there is a mandatory jump that skips over the actual code for the for loop. Afterwards, there is a second conditional jump that, if successful, jumps back to the start of the loop. These two jumps could potentially slow the code down. On the other hand, the optimized code assumes that the for loop will be run, and has a conditional jump to skip it if necessary. Since most of the time, the program will go through the for loop, having just one jump is advantageous, as it will rarely be taken. Inside the loop, the non-optimized code always uses a movl instruction to set counters to zero, even if that counter could be reset multiple times, like in the inner loop. Alternately, the optimized version resets counters by using the xorl instruction as follows:

```
xorl %ebx ,%ebx
```

In this context, the EBX register is being used as the counter for the inner loop. This instruction sets EBX to zero because the operands are the same and can never have a 1 in different bit positions. Using the xorl instruction is faster than a movl intruction because xorl deals directly with the bits, eliminating the need to store a new zero value. Even though the zero constant would be accessed in immadiate mode, it will require more code space since the constant itself has to be stored in the instruction. Another possible explanation for this is that the assembler might be able to recognize this instruction as simply setting the register to zero, which it would do without worrying about the contents of the register.

The two versions of code also differ in the way they access the matrices, causing a change in the structure of their nested for loops. In a general sense, the non-optimized version computes the postition in the matrix it needs within the inner loop. In contrast, the optimized version computes the correct row in the outer loop, and then moves through said row in the inner loop. This is beneficial because it breaks up a complex computation. Every time the inner loop of the non-optimized code runs it has to go through this set of instructions:

```
imull $50, %ecx, %ecx
leal (%ecx, %edx), %edx
movl %eax, 10040(%esp, %edx, 4)
```

Recall that we set the maximum size of the matrix to fifty rows and fifty columns. Also note that the EDX register is acting as the inner loop counter, the column number, and the ECX register is acting as the outer loop counter, the row number. These instructions are using the counters to compute how many bytes passed the beginning of the matrix the piece of data stored in EAX should be placed. Since this computation is performed in the inner loop, it occurs for every element in the matrix. On the other hand, the optimized version of the code first performs these instructions in the outer loop:

```
imull $200, 28(%esp), %esi
leal 10044(%esp), %eax
leal (%eax, %esi), %esi
```

And then uses this instruction in the inner loop:

```
addl $4, %esi
```

Note that in the first section of code 28(%esp) holds the outer loop counter, which is the row number. Also note that 10044(%esp) holds that address of the beginning of the matrix. The result of the first section of code is that the ESI register now holds the address of the first element of the current row of the matrix. This now allows the inner loop to step through the current row, element by element, using the addl instruction shown. All of this is faster than the non-optimized version's method because the imull instruction is only used as many times as the number of rows, as compared to the non-optimized code, which uses imull as many times as the number of rows times the number of columns. The advantage here is that being able to use an addl instruction repeatedly is faster than having to use an imull instruction repeatedly, since imull deals with changing multiple registers. Also, addl has the advantage of taking less codespace than imull.

In conclusion, the optimizer made the program run faster through a series structural and computational changes. These included deleting some function calls, removing extra register moves, rearranging the for loops, and exchanging slower instructions for faster ones. Overall, the changes were less concerned with conserving memory or shortening compile time, but more with program speed. The most amazing part about the optimizer is that it acts as though it knows exactly what the code is doing, just like a little man inside the machine would, but there is no little man.

A File Names

1. matrix.c holds our C code
2. optimized.s holds the optimized version of the assembly code
3. non-optimized.s holds the non-optimized version of the assembly code