The Maze Inside the Machine

Aaron Okano, Jason Wong, Meenal Tambe,
and Gowtham Vijayaragavan

June 3, 2011

The code that we decided to use was a C program titled maze.c. In the program, a two-dimensional array was placed into a file. An "O" indicated an open space while an "X" represented a closed portion in the maze. The goal of the program was to find the best possible path to complete the maze. The method for finding the best path was written into the find_path function, which had recursive properties since it only called either itself or the **printf()** function. When running the program, the coordinates for the best path were printed in row-major order. Because of its frequent dependence on two-dimensional arrays, maze.c was the best example to analyze the differences from its optimized and unoptimized source code files. After compiling maze.c, two .s files were made using the -S option for the unoptimized file and the -O3 option for the optimized files. The attached files were titled maze-opt.s for the optimized file and maze-noopt.s for the unoptimized file. One of the key differences that we noticed in the two files were the order of the functions and the number of jumps in the two files.

One of the most noteworthy changes in the optimized code is the new ordering. Many of the instructions that follow immediately after another instruction are placed right below it, so that the code may run smoothly. The optimized code also combines some functions that were separate in the unoptimized code. Instead of separating the functions, the optimized code reduces the number of jumps. In this regard, the amount of jumps is significantly reduced in the optimized code. The generous use of conditional jumps plays a big part in this. If a jump is not absolutely needed, the code just executes downward, which would not be possible if not for the appropriate ordering. A vast majority of the instructions in the unoptimized code end with the **jmp**, whereas there are only 4 calls to **jmp** in the optimized code.

An instruction that clearly benefits from this is **main()**:

```
main:
   pushl   %ebp
   movl    %esp, %ebp
   andl    $-16, %esp
   subl    $96, %esp
   cmpl    $2, 8(%ebp)
   movl    %ebx, 88(%esp)
   movl    %esi, 92(%esp)
   je      .L20
```

A jump is made if there are only two arguments. Otherwise, the instruction after **main()**, .L18, which performs necessary stack operations, and returns. It is extremely minimalist, and nothing else is performed unless the condition, that the number of arguments is two, is met.

Additionally, when comparing the two files, an observation we made was that the **find_path()** function in the optimized file was slightly longer in length than the **find_path()** function in the unoptimized file. It also appears that it is called quite often, with many checks and calculations being done inside **find_path()** itself. Then, after the jump from the function is made, the instructions do not get tangled in a series of jumps. If a jump is made, it is to terminate the program. Otherwise, the function, **find_path()**, is called again. This is in contrast to the unoptimized version, where jumps are made quite frequently from instruction to instruction.

The -O3 modification also makes some other very subtle changes. One of these subtle changes is how if statements are handled. When we look at the unoptimized version, we see that it it writes the function as if it was building the instructions based on the how the program would be during a straight run while in -O3 we see it more of a function based implementation. For example, take the first if statement checking whether or not there is a correct number of arguments. On the unoptimized version, we see that the jump is to occur if it is not equal. However, on the optimized version, we see that it jumps when it is equal to zero. This implementation can either hurt the run-time or help it. In our case it hurts it during **main()** because it is doing an unnecessary jump. However, if we were searching for a very specific condition such as in our **find_path()**, looking for something that has a probability of less than one third of happening, then the optimized implementation is superior because it prevents excess jumps.

Another subtlety that we found was that the optimized version would push things straight into the stack. We can take our example from the time before the **fopen()** call. The unoptimized version has

```
movl $.LC0, %edx
movl 12(%ebp), %eax
addl $4, %eax
movl (%eax), %eax
movl %edx, 4(%esp)
movl %eax, (%esp)
call fopen
```

while the optimized version has the following:

```
movl $.LC1, 4(%esp)
movl 4(%eax), %eax
movl %eax, (%esp)
call fopen
```

We see that if the last argument (.LC0 and LC1) is put directly into the stack in the optimized version while in the unoptimized version, we see that it is first put into EDX and then into the stack. Right there we can see that the optimized version is better. Why is it better? Because it is shorter. That means there are less instructions for the CPU which mean the run-time is faster and still reach the same desired outcome.

In addition to function calls, loops are also handled very differently in the two versions. In the unoptimized version we see that after each iteration of the loop, it jumps back to the top of the loop. In the optimized file, the .L20 function combined code for the .L3 and .L4 from the unoptimized code. Instead of separating the functions, the optimized code reduces the number of jumps. .L20 plays a crucial role in making the "for" loop concise. In the unoptimized code, .L4 would be called whenever the "for" loops needed to run. The need for multiple jumps necessitates a counter. In the .L20 function of the optimized code, however, the need for a counter is eliminated since the following code for the "for" loops was merely inserted eight times:

```
movl    %ebx, 8(%esp)
movl    $11, 4(%esp)
movl    %eax, (%esp)
call    fgets
movl    $7, 8(%esp)
```

This helps the program run faster in two ways. The first way it helps with runtime is that we do not have to check whether a loop has fulfilled its requirement. We can avoid that comparison thus saving CPU time. Also, if we avoid that comparison, there would be no need to jump. In this case, code compactness is sacrificed for speed.

As we traverse the program, we notice another subtlety. Every time we want to make a register 0, the unoptimized code moves 0 into the corresponding register (ie. `movl $0, %eax`). The optimized version however, uses a different syntax. The function, .L8, uses the **xorl** command in the optimized code:

```
xorl    %eax, %eax
```

This command provides a more concise way of using the **movl** command since it doesnt need to initialize the EAX register to 0 first. This saves one line of instruction that would have used up space on the stack. The purpose of **xorl** in this program is to initialize the for loop that was written in the maze.c code. **xorl** shared similar properties with .L7 in the unoptimized code, where the EAX register was set to zero so that the recursive section could loop again without carrying values from the previous run. Moving a 0 into a register is considered an integer operation while **xorl** is considered a bitwise operation and since bitwise operations are always faster than integer operations, using **xorl** is more efficient. Overall, the reduction in the number of jumps and the increase in immediate code allows the program to run faster since this process does not need to move throughout the stack as frequently as the unoptimized file.

The reduction of jumps is not the only thing that affects the performance of the program. The optimized version also takes advantage of the speed of registers to improve the performance of the program. In the function **find_path()**, the unoptimized version, the index and the index2 variables are compared from the stack while the optimized version copies the values from the stack into the registers. Although it is an extra instruction, it pays off as the program progresses. If it fails the first if statement (the compound if statement), it progresses to the second one. Here, index and index2 are accessed again. In the optimized version, it simply uses the copied value in the registers instead of looking at the stack again. Since registers are within the CPU, the access speed to registers are many magnitudes faster the from memory. With the reduction of accesses to memory, the running time of the program is reduced thus improving the performance of the program further.

Because the optimized version places arguments to **find_path()** in registers, it is able to access the 2d array much more efficiently than the unoptimized code. To access a particular portion of the array, the computer needs to calculate maze+(index*8+index2). In the unoptimized version, the code appears as such:

```
movl    12(%ebp), %eax
sall    $3, %eax
movl    %eax, %edx
addl    8(%ebp), %edx
movl    16(%ebp), %eax
leal    (%edx,%eax), %eax
movzbl  (%eax), %eax
cmpb    $88, %al
```

The values for the address of the maze pointer and the two index values need to be copied from the stack into registers to perform the necessary arithmetic operations. On top of that, there are still unnecessary operations, such as the MOV from EAX to EDX. The optimized code can take shortcuts because the arguments are already in registers:

```
leal    (%edi,%ebx,8), %edx
cmpb    $88, (%edx,%esi)
```

Because of the easier access to the arguments, not only are the calculations easier to perform, the code can also take advantage of more advanced addressing modes.

One feature of GCC's optimizations that appears in maze-opt.s is the emphasis that is placed on safety. Encompassed within GCC's -O3 flag is the optimization `-fcaller-saves`, which tells GCC to place the current register values at the front of the stack frame of the calling function. For example, surrounding the recursive calls to **find_path()** appears the code:

```
movl %edx, -24(%ebp)
movl %esi, 8(%esp)
movl %edi, (%esp)
call find_path
movl -28(%ebp), %edx
```

Here, the compiler put c(EDX) 24 bytes into the stack frame and then put it back into EDX after the find_path call. The reason for this is to prevent situations such as the subroutine changing the EDX register or malicious leprechaun invasions.

We also observed GCCs security emphasis in its use of the function **__printf_chk()** in maze-opt.s in place of the usual **printf()** that the unoptimized version uses. The optimized version uses the stack far more than the unoptimized one, so the need to check for stack overflows is more necessary. In particular, the **printf()** function tends to use a tremendous amount of space on the stack, and added on top of the increased stack usage from optimizations such as `-fcaller-saves`, the possibility of a stack overflow is increased. **__printf_chk()** partially solves this problem by checking the size of the stack prior to doing any stack-heavy calculations. Naturally, this extra action of checking the stack translates to slower performance, which illustrates how GCC does not focus entirely on the speed of the program.

Overall, the optimized version of maze.c was efficient, faster, and provided more safety than the unoptimized code. Although the code was longer and initially gave the illusion of a less organized and therefore less efficient way of decreasing the run-time of the program, the optimized file superseded this notion by providing a different approach to run the program and save memory.