

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 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 doesn't 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 than 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 maze array, the computer needs to calculate $\text{maze} + (\text{index} * 8 + \text{index}2)$. 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 GCC's 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. By changing the order of instructions around significantly, reducing the number of jumps, and dealing with registers directly, the optimized code has traded conciseness for speed. However, when it comes to discovering the leprechauns pot of gold at the end of the maze, it is a worthwhile sacrifice.

Appendices

A Contributions

B Code used in this report

maze.c:

```
1  /*
2   * File:   maze.c
3   * Author: Gowtham
4   *
5   * Created on July 17, 2010, 8:01 PM
6   */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11
12 //The purpose of this program is to scan in a file that has
13 //a grid of sorts, X being locations to which you cannot
14 //move to. Locations with an 0 are ok. The sequence of
15 //positions to take to get from (0,1) to (7,7) is
16 //printed out.
17
18 void find_path2(char maze[8][8], int index, int index2);
19 int find_path(char maze[][8], int index, int index2);
20 /*
21 *
22 */
23 int main(int argc, char* argv[]) {
24     FILE *inp;
25     char maze[8][8];
26     int i;
27     if (argc == 2)
28     {
29         inp = fopen(argv[1], "r");
30
31         for (i = 0; i < 8; i++)
32         {
33             fgets(maze[i], 11, inp);
34         } //for
35
36         if (find_path(maze, 7, 7) == 0)
37             printf("No path was found.");
38         else
```

```

39         printf("(7, 7)");
40
41     } //if
42
43
44     return 0;
45
46 } //main()
47
48 int find_path(char maze[][8], int index, int index2)
49 {
50     if (index < 0 || index2 < 0 || index > 7 || index2 > 7)
51     {
52         return 0;
53     } //if
54
55     if (maze[index][index2]== 'X')
56     {
57         return 0;
58     } //if
59
60     if (index == 0 && index2 == 1)
61     {
62         return 1;
63     } //if
64
65     maze[index][index2] = 'X';
66
67
68     if (find_path(maze, index, index2 + 1) == 1)
69     {
70         printf("(%d, %d) \n", index, index2 + 1);
71         return 1;
72     } //if
73     if (find_path(maze, index, index2 - 1) == 1)
74     {
75         printf("(%d, %d) \n", index, index2 - 1);
76         return 1;
77     } //if
78     if (find_path(maze, index - 1, index2) == 1)
79     {
80         printf("(%d, %d) \n", index - 1, index2);
81         return 1;
82     } //if
83     if (find_path(maze, index + 1, index2) == 1)
84     {
85         printf("(%d, %d) \n", index + 1, index2);
86         return 1;
87     } //if
88
89
90
91     return 0;
92 } //find_path()
93

```


maze-noopt.s:

```
1      .file      "maze.c"
2      .section   .rodata
3      .LC0:
4          .string  "r"
5      .LC1:
6          .string  "No path was found."
7      .LC2:
8          .string  "(7, 7)"
9      .text
10     .globl main
11     .type       main, @function
12     main:
13         pushl    %ebp
14         movl     %esp, %ebp
15         andl     $-16, %esp
16         subl     $96, %esp
17         cmpl     $2, 8(%ebp)
18         jne      .L2
19         movl     $.LC0, %edx
20         movl     12(%ebp), %eax
21         addl     $4, %eax
22         movl     (%eax), %eax
23         movl     %edx, 4(%esp)
24         movl     %eax, (%esp)
25         call     fopen
26         movl     %eax, 88(%esp)
27         movl     $0, 92(%esp)
28         jmp      .L3
29     .L4:
30         leal     24(%esp), %eax
31         movl     92(%esp), %edx
32         sall     $3, %edx
33         leal     (%eax,%edx), %edx
34         movl     88(%esp), %eax
35         movl     %eax, 8(%esp)
36         movl     $11, 4(%esp)
37         movl     %edx, (%esp)
38         call     fgets
39         addl     $1, 92(%esp)
40     .L3:
41         cmpl     $7, 92(%esp)
42         jle      .L4
43         movl     $7, 8(%esp)
44         movl     $7, 4(%esp)
45         leal     24(%esp), %eax
46         movl     %eax, (%esp)
47         call     find_path
48         testl    %eax, %eax
49         jne      .L5
50         movl     $.LC1, %eax
51         movl     %eax, (%esp)
52         call     printf
53         jmp      .L2
54     .L5:
```

```

55         movl    $.LC2, %eax
56         movl    %eax, (%esp)
57         call    printf
58     .L2:
59         movl    $0, %eax
60         leave
61         ret
62         .size    main, .-main
63         .section .rodata
64     .LC3:
65         .string  "(%d, %d) \n"
66         .text
67     .globl find_path
68         .type    find_path, @function
69     find_path:
70         pushl    %ebp
71         movl    %esp, %ebp
72         subl    $24, %esp
73         cmpl    $0, 12(%ebp)
74         js      .L7
75         cmpl    $0, 16(%ebp)
76         js      .L7
77         cmpl    $7, 12(%ebp)
78         jg      .L7
79         cmpl    $7, 16(%ebp)
80         jle     .L8
81     .L7:
82         movl    $0, %eax
83         jmp     .L9
84     .L8:
85         movl    12(%ebp), %eax
86         sall    $3, %eax
87         movl    %eax, %edx
88         addl    8(%ebp), %edx
89         movl    16(%ebp), %eax
90         leal    (%edx,%eax), %eax
91         movzbl    (%eax), %eax
92         cmpb    $88, %al
93         jne     .L10
94         movl    $0, %eax
95         jmp     .L9
96     .L10:
97         cmpl    $0, 12(%ebp)
98         jne     .L11
99         cmpl    $1, 16(%ebp)
100        jne     .L11
101        movl    $1, %eax
102        jmp     .L9
103     .L11:
104        movl    12(%ebp), %eax
105        sall    $3, %eax
106        movl    %eax, %edx
107        addl    8(%ebp), %edx
108        movl    16(%ebp), %eax
109        leal    (%edx,%eax), %eax
110        movb    $88, (%eax)
111        movl    16(%ebp), %eax

```

```

112      addl    $1, %eax
113      movl    %eax, 8(%esp)
114      movl    12(%ebp), %eax
115      movl    %eax, 4(%esp)
116      movl    8(%ebp), %eax
117      movl    %eax, (%esp)
118      call    find_path
119      cmpl    $1, %eax
120      jne     .L12
121      movl    16(%ebp), %eax
122      leal    1(%eax), %edx
123      movl    $.LC3, %eax
124      movl    %edx, 8(%esp)
125      movl    12(%ebp), %edx
126      movl    %edx, 4(%esp)
127      movl    %eax, (%esp)
128      call    printf
129      movl    $1, %eax
130      jmp     .L9
131  .L12:
132      movl    16(%ebp), %eax
133      subl    $1, %eax
134      movl    %eax, 8(%esp)
135      movl    12(%ebp), %eax
136      movl    %eax, 4(%esp)
137      movl    8(%ebp), %eax
138      movl    %eax, (%esp)
139      call    find_path
140      cmpl    $1, %eax
141      jne     .L13
142      movl    16(%ebp), %eax
143      leal    -1(%eax), %edx
144      movl    $.LC3, %eax
145      movl    %edx, 8(%esp)
146      movl    12(%ebp), %edx
147      movl    %edx, 4(%esp)
148      movl    %eax, (%esp)
149      call    printf
150      movl    $1, %eax
151      jmp     .L9
152  .L13:
153      movl    12(%ebp), %eax
154      leal    -1(%eax), %edx
155      movl    16(%ebp), %eax
156      movl    %eax, 8(%esp)
157      movl    %edx, 4(%esp)
158      movl    8(%ebp), %eax
159      movl    %eax, (%esp)
160      call    find_path
161      cmpl    $1, %eax
162      jne     .L14
163      movl    12(%ebp), %eax
164      leal    -1(%eax), %ecx
165      movl    $.LC3, %eax
166      movl    16(%ebp), %edx
167      movl    %edx, 8(%esp)
168      movl    %ecx, 4(%esp)

```

```

169         movl    %eax, (%esp)
170         call    printf
171         movl    $1, %eax
172         jmp     .L9
173     .L14:
174         movl    12(%ebp), %eax
175         leal    1(%eax), %edx
176         movl    16(%ebp), %eax
177         movl    %eax, 8(%esp)
178         movl    %edx, 4(%esp)
179         movl    8(%ebp), %eax
180         movl    %eax, (%esp)
181         call    find_path
182         cmpl    $1, %eax
183         jne     .L15
184         movl    12(%ebp), %eax
185         leal    1(%eax), %ecx
186         movl    $.LC3, %eax
187         movl    16(%ebp), %edx
188         movl    %edx, 8(%esp)
189         movl    %ecx, 4(%esp)
190         movl    %eax, (%esp)
191         call    printf
192         movl    $1, %eax
193         jmp     .L9
194     .L15:
195         movl    $0, %eax
196     .L9:
197         leave
198         ret
199     .size      find_path, .-find_path
200     .ident     "GCC: (Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2"
201     .section   .note.GNU-stack,"",@progbits

```

maze-opt.s:

```
1      .file      "maze.c"
2      .section   .rodata.str1.1,"aMS",@progbits,1
3  .LC0:
4      .string    "(%d, %d) \n"
5      .text
6      .p2align 4,,15
7  .globl find_path
8      .type      find_path, @function
9  find_path:
10     pushl      %ebp
11     movl       %esp, %ebp
12     subl       $56, %esp
13     movl       %esi, -8(%ebp)
14     movl       16(%ebp), %esi
15     movl       %ebx, -12(%ebp)
16     movl       12(%ebp), %ebx
17     movl       %edi, -4(%ebp)
18     movl       8(%ebp), %edi
19     testl      %esi, %esi
20     js         .L8
21     movl       %ebx, %eax
22     shrl       $31, %eax
23     testb      %al, %al
24     jne        .L8
25     cmpl       $7, %esi
26     jg         .L8
27     cmpl       $7, %ebx
28     jg         .L8
29     leal       (%edi,%ebx,8), %edx
30     xorl       %eax, %eax
31     cmpb       $88, (%edx,%esi)
32     je         .L2
33     cmpl       $1, %esi
34     jne        .L12
35     testl      %ebx, %ebx
36     movb       $1, %al
37     je         .L2
38  .L12:
39     movb       $88, (%edx,%esi)
40     leal       1(%esi), %edx
41     movl       %edx, 8(%esp)
42     movl       %edx, -28(%ebp)
43     movl       %ebx, 4(%esp)
44     movl       %edi, (%esp)
45     call       find_path
46     movl       -28(%ebp), %edx
47     cmpl       $1, %eax
48     je         .L14
49     leal       -1(%esi), %edx
50     movl       %edx, 8(%esp)
51     movl       %edx, -28(%ebp)
52     movl       %ebx, 4(%esp)
53     movl       %edi, (%esp)
54     call       find_path
55     movl       -28(%ebp), %edx
```

```

56      cmpl      $1, %eax
57      je        .L14
58      leal      -1(%ebx), %edx
59      movl      %edx, 4(%esp)
60      movl      %edx, -28(%ebp)
61      movl      %esi, 8(%esp)
62      movl      %edi, (%esp)
63      call      find_path
64      movl      -28(%ebp), %edx
65      cmpl      $1, %eax
66      je        .L16
67      addl      $1, %ebx
68      movl      %esi, 8(%esp)
69      movl      %ebx, 4(%esp)
70      movl      %edi, (%esp)
71      call      find_path
72      movl      %eax, %edx
73      xorl      %eax, %eax
74      cmpl      $1, %edx
75      jne       .L2
76      movl      %esi, 12(%esp)
77  .L13:
78      movl      %ebx, 8(%esp)
79      movl      $.LCO, 4(%esp)
80      movl      $1, (%esp)
81      call      __printf_chk
82      movl      $1, %eax
83      jmp       .L2
84      .p2align 4,,7
85      .p2align 3
86  .L8:
87      xorl      %eax, %eax
88  .L2:
89      movl      -12(%ebp), %ebx
90      movl      -8(%ebp), %esi
91      movl      -4(%ebp), %edi
92      movl      %ebp, %esp
93      popl      %ebp
94      ret
95      .p2align 4,,7
96      .p2align 3
97  .L14:
98      movl      %edx, 12(%esp)
99      jmp       .L13
100     .p2align 4,,7
101     .p2align 3
102  .L16:
103     movl      %esi, 12(%esp)
104     movl      %edx, 8(%esp)
105     movl      $.LCO, 4(%esp)
106     movl      $1, (%esp)
107     call      __printf_chk
108     movl      $1, %eax
109     jmp       .L2
110     .size     find_path, .-find_path
111     .section   .rodata.str1.1
112  .LC1:

```

```

113     .string      "r"
114 .LC2:
115     .string      "No path was found."
116 .LC3:
117     .string      "(7, 7)"
118     .text
119     .p2align 4,,15
120 .globl main
121     .type        main, @function
122 main:
123     pushl        %ebp
124     movl         %esp, %ebp
125     andl         $-16, %esp
126     subl         $96, %esp
127     cmpl         $2, 8(%ebp)
128     movl         %ebx, 88(%esp)
129     movl         %esi, 92(%esp)
130     je           .L20
131 .L18:
132     xorl         %eax, %eax
133     movl         88(%esp), %ebx
134     movl         92(%esp), %esi
135     movl         %ebp, %esp
136     popl         %ebp
137     ret
138     .p2align 4,,7
139     .p2align 3
140 .L20:
141     movl         12(%ebp), %eax
142     leal         16(%esp), %esi
143     movl         $.LC1, 4(%esp)
144     movl         4(%eax), %eax
145     movl         %eax, (%esp)
146     call         fopen
147     movl         $11, 4(%esp)
148     movl         %esi, (%esp)
149     movl         %eax, %ebx
150     movl         %eax, 8(%esp)
151     call         fgets
152     leal         24(%esp), %eax
153     movl         %ebx, 8(%esp)
154     movl         $11, 4(%esp)
155     movl         %eax, (%esp)
156     call         fgets
157     leal         32(%esp), %eax
158     movl         %ebx, 8(%esp)
159     movl         $11, 4(%esp)
160     movl         %eax, (%esp)
161     call         fgets
162     leal         40(%esp), %eax
163     movl         %ebx, 8(%esp)
164     movl         $11, 4(%esp)
165     movl         %eax, (%esp)
166     call         fgets
167     leal         48(%esp), %eax
168     movl         %ebx, 8(%esp)
169     movl         $11, 4(%esp)

```

```

170      movl    %eax, (%esp)
171      call    fgets
172      leal    56(%esp), %eax
173      movl    %ebx, 8(%esp)
174      movl    $11, 4(%esp)
175      movl    %eax, (%esp)
176      call    fgets
177      leal    64(%esp), %eax
178      movl    %ebx, 8(%esp)
179      movl    $11, 4(%esp)
180      movl    %eax, (%esp)
181      call    fgets
182      leal    72(%esp), %eax
183      movl    %ebx, 8(%esp)
184      movl    $11, 4(%esp)
185      movl    %eax, (%esp)
186      call    fgets
187      movl    $7, 8(%esp)
188      movl    $7, 4(%esp)
189      movl    %esi, (%esp)
190      call    find_path
191      testl    %eax, %eax
192      je      .L21
193      movl    $.LC3, 4(%esp)
194      movl    $1, (%esp)
195      call    __printf_chk
196      xorl    %eax, %eax
197      movl    88(%esp), %ebx
198      movl    92(%esp), %esi
199      movl    %ebp, %esp
200      popl    %ebp
201      ret
202      .p2align 4,,7
203      .p2align 3
204  .L21:
205      movl    $.LC2, 4(%esp)
206      movl    $1, (%esp)
207      call    __printf_chk
208      jmp     .L18
209      .size    main, .-main
210      .ident   "GCC: (Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2"
211      .section .note.GNU-stack,"",@progbits

```