Decoding the Volatile keyword in C through Assembly Code

Anubhav Chaturvedi
Department of Computer Science
And Engineering,
The M. S. University of Baroda,
Vadodara, Gujarat,India
acanubhav@gmail.com

Saurabh Jain
Department of Computer Science
And Engineering,
The M..S. University of Baroda,
Vadodara,Gujarat,India
saurabhjain9307@gmail.com

Dr. Anjali Jivani
Department of Computer Science
And Engineering,
The M. S. University of Baroda,
Vadodara,Gujarat,India
anjali_jivani@yahoo.com

Abstract— The growing complexity and high efficiency requirements of embedded systems call for new code optimization techniques and architecture exploration, using re target able C and C++ compilers. The first commercial tools are already in industrial use. The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler. In this paper we have tried to decode this volatile keyword mystery by digging into the assembly code generated by implemented C program.

Keywords- *vimdiff, withvolatile, withoutvoatile, polling, delay*

*****

I. INTRODUCTION

The volatile keyword is intended to prevent the compiler from applying any optimisations on objects that can change in ways that cannot be determined by the compiler. So, what are its instructions to the compiler? It tells the compiler that the value of the variable may change at any time during the execution of the code without the knowledge of the compiler. If proper precautions are not taken, the desired output may not be achieved. A variable should be declared volatile whenever its value may change unexpectedly. . Volatile variables are variables that can be changed at any time by other external programs or by the same program.

II. IMPLEMENTATION IN C PROGRAMMING

The syntax for declaring the variable as ‘volatile’ is: volatile dataType variable. Some examples for the volatile keyword are: (Note : All codes are compiled in the gcc compiler version: gcc version 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04)). Now displaying the Polling Implementation.

A. Without Volatile Keyword

```
//filename:withoutvolatile1.cpp
#include<iostream>
Using namespace std;
Intmain()
{inti=0;
Int flag=0;
While(flag!=1)
{
//keep pooling till flag becomes 1
if(i==50)
{flag=1;}
++I;}}
```

B. With Volatile Keyword

```
//filename:withvolatile.cpp
#include<iostream>
Using namespace std;
Intmain()
{inti=0;
Volatile int flag=0;
While(flag!=1)
{
//keep pooling till flag becomes 1
if(i==50)
{flag=1;}
++I;
}}
```

Here, our intention is to keep polling inside the while loop until the flag value is SET to the value 1, which might be done by an I/O device. However, during the compilation phase, the compiler will find that this piece of code is not achieving any valuable results; hence, the code (withoutvolatile.cpp) will be optimized by removing this. If one observes the code that follows below, the condition inside the while loop is replaced by the compiler to while (TRUE). This is primarily done in compilers in the embedded systems environment, where generating optimal machine code is very important. In case of studying about the Linux device drivers and also troubleshooting them this is the primary component.

C. int main()

```
{inti = 0;
int flag = 0;
while(TRUE)
{//Infinite loop;}
//rest of the code
return 0;}
```

III. OPTIMIZATION AND USES IN OPERATING SYSTEM

A. *Optimisation*

How can one confirm that the compiler is really optimizing the code? For that you must see the assembly implementation of

the above program by compiling the C source code with the -save-temps option as shown below:
gcc -o withoutvolatilewithvolatile.c -save-temps

1. When we compile code with the –save-temps option of gcc, it generates three output files:Preprocessed code (with the .i extension).
2. Assembly code (with the .s extension).
3. *Object code (with the .o option).Maintaining the Integrity of the Specifications*

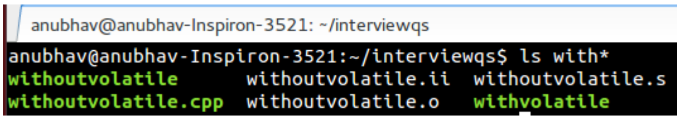

Figure 1. To list all the files with "with" "without" volatile in front of them.

Now, if you observe Figure 2, the size is found to be 1743 bytes in the fifth column. Next, qualify the flag variable to 'volatile' for the code shown in example 1, and generate the assembly code (call this withvolatile.s) before checking the size by issuing the ls command. The size obtained in my system is shown in Figure 3. Now, if you observe Figure 3, the size is found to be 1784 bytes in the fifth column. So, when we compare the sizes of both the codes, with and without the 'volatile' key word, it is obvious that the compiler is not optimizing the 'flag' variable when it is qualified as 'volatile'. We can experiment further to explore where exactly the compiler is optimizing the code. To find this out, apply the vimdiff command to the assembly codes generated with and without the keyword 'volatile'—the difference is shown below.

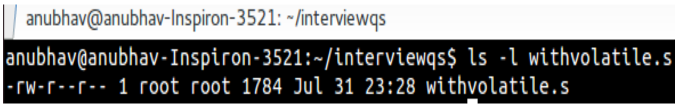

Figure 2. To see the size of withoutvolatile.s we need to run ls -l command on it.

A. Withvolatile

```
.file  "withvolatile.cpp"
       .local   _ZStL8__ioinit
       .comm    _ZStL8__ioinit,1,1
       .text
       .globl   main
       .type    main, @function
main:
.LFB971:
       .cfi_startproc
       pushq    %rbp
       .cfi_def_cfa_offset 16
       .cfi_offset 6, -16
       movq     %rsp, %rbp
       .cfi_def_cfa_register 6
       movl     $0, -4(%rbp)
       movl     $0, -8(%rbp)
       jmp      .L2
.L4:
       cmpl     $50, -4(%rbp)
       jne      .L3
       movl     $1, -8(%rbp)
.L3:
       addl     $1, -4(%rbp)
.L2:
       movl     -8(%rbp), %eax
       cmpl     $1, %eax
       setne    %al
       testb    %al, %al
       jne      .L4
       movl     $0, %eax
       popq     %rbp
       .cfi_def_cfa 7, 8
       ret
       .cfi_endproc
.LFE971:
       .size    main, .-main
       .type
       _Z41__static_initialization_and_destruction_0ii,
@function
_Z41__static_initialization_and_destruction_0ii:
.LFB972:
       .cfi_startproc
       pushq    %rbp
       .cfi_def_cfa_offset 16
       .cfi_offset 6, -16
       movq     %rsp, %rbp
       .cfi_def_cfa_register 6
       subq     $16, %rsp
       movl     %edi, -4(%rbp)
       movl     %esi, -8(%rbp)
       cmpl     $1, -4(%rbp)
       jne      .L6
       cmpl     $65535, -8(%rbp)
       jne      .L6
       movl     $_ZStL8__ioinit, %edi
       call     _ZNSt8ios_base4InitC1Ev
       movl     $__dso_handle, %edx
       movl     $_ZStL8__ioinit, %esi
       movl     $_ZNSt8ios_base4InitD1Ev, %edi
       call     __cxa_atexit
.L6:
       leave
       .cfi_def_cfa 7, 8
       ret
       .cfi_endproc
.LFE972:
```

```
        .size
        _Z41__static_initialization_and_destruction_0ii,      .-
_Z41__static_initialization_and_destruction_0ii
        .type   _GLOBAL__sub_I_main, @function
_GLOBAL__sub_I_main:
.LFB973:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $65535, %esi
        movl    $1, %edi
        call
        _Z41__static_initialization_and_destruction_0ii
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE973:
        .size   _GLOBAL__sub_I_main,                          .-
_GLOBAL__sub_I_main
        .section  .init_array,"aw"
        .align 8
        .quad   _GLOBAL__sub_I_main
        .hidden __dso_handle
        .ident  "GCC:   (Ubuntu    4.8.4-2ubuntu1~14.04)
4.8.4".section  .note.GNU-stack,"",@progbits
```

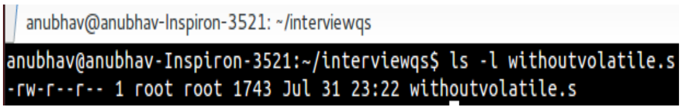

Figure 3. To see the size of withvolatile.s we need to run ls -l command on it.

B. Withoutvolatile

```
        .file   "withoutvolatile.cpp"
        .local  _ZStL8__ioinit
        .comm   _ZStL8__ioinit,1,1
        .text
        .globl  main
        .type   main, @function
main:
.LFB971:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, -8(%rbp)
        movl    $0, -4(%rbp)
        jmp     .L2
.L4:
        cmpl    $50, -8(%rbp)
        jne     .L3
        movl    $1, -4(%rbp)
.L3:
        addl    $1, -8(%rbp)
.L2:
        cmpl    $1, -4(%rbp)
        jne     .L4
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE971:
        .size   main, .-main
        .type
        _Z41__static_initialization_and_destruction_0ii,
@function
_Z41__static_initialization_and_destruction_0ii:
.LFB972:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        cmpl    $1, -4(%rbp)
        jne     .L6
        cmpl    $65535, -8(%rbp)
        jne     .L6
        movl    $_ZStL8__ioinit, %edi
        call    _ZNSt8ios_base4InitC1Ev
        movl    $__dso_handle, %edx
        movl    $_ZStL8__ioinit, %esi
        movl    $_ZNSt8ios_base4InitD1Ev, %edi
        call    __cxa_atexit
.L6:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE972:
        .size
        _Z41__static_initialization_and_destruction_0ii,      .-
_Z41__static_initialization_and_destruction_0ii
        .type   _GLOBAL__sub_I_main, @function
_GLOBAL__sub_I_main:
.LFB973:
```

```
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $65535, %esi
        movl    $1, %edi
        call
        _Z41__static_initialization_and_destruction_0ii
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
    .LFE973:
        .size   _GLOBAL__sub_I_main,                    .-
_GLOBAL__sub_I_main
        .section  .init_array,"aw"
        .align 8
        .quad   _GLOBAL__sub_I_main
        .hidden  __dso_handle
        .ident  "GCC:   (Ubuntu   4.8.4-2ubuntu1~14.04)
4.8.4"
        .section  .note.GNU-stack,"",@progbits
```

In the disassembly of the non-volatile version (withoutvolatile.s) of the while loop shown in the statements in Italics, load the value of the flag into memory locations [-8(%rbp) & -4(%rbp)] outside the loop labeled .L2. This is because, since the flag variable is not declared volatile, the compiler assumes that its value cannot be modified outside the program. Having already read the value of the flag into memory locations [-8(%rbp) & -4(%rbp)], the compiler omits reloading the value of the flag variable when optimization is enabled, because its value cannot change. The result is ultimately the control getting into the infinite loop labeled .L2. In contrast, in the disassembly of the volatile version (withvolatile.s) of the while loop shown in above code, the compiler assumes that the value of the flag variable can change outside the program and performs no optimisations. Consequently, the value of the flag is loaded into the register %eax every time from the memory [-8(%rbp)] inside the loop labeled .L2. As a result, the value of the flag is checked every time, and further decisions are taken depending upon the value of the flag variable. To avoid optimization problems caused by changes to the program state external to the implementation, it is always safer to declare the variable as ‘volatile’. This helps to avoid unexpected results. From this, we can conclude that the ‘volatile’ key word prevents optimization of the code by the compiler.

B. Uses In Operating System

1. Delay generations using loops Let us consider another example, where ‘for’ loops are used commonly in the Embedded C code to generate small delays in LED's used in PC's and Laptops as shown in the following code:

```
int main()
{ inti;
//Loop for delay generation
for( i = 0; i < 100; i++)
{ ; }
//Again the remaining code goes here
return 0;  }
```

In fact, a compiler might optimize the code shown above into nothing. A local variable ‘i’ is the counter for a loop that does nothing but increment value ‘i’ until it’s equal to 100. Thus, the optimizer can replace the loop with a single assignment that just sets ‘i’ to its final value. When that happens, the delay code doesn’t achieve what the programmer had intended. So, it is always better to declare the local variable ‘i’ as ‘volatile’ even though the code might be less efficient, since we will get the desired results, as shown in the code below:

```
int main()
{ volatileinti;
//Loop for delay generation
for( i = 0; i < 100; i++)
{ ;  }
//Again the remaining code goes here
return 0;  }
```

Similar results are acquired when we run both the codes, leading to a difference in sizes of the codes sizes and also addition of code in the assembly code of “volatile” containing program to tell the compiler not to optimize the code.

2. Global variables accessed by multiple tasks within a multi-threaded application Let us consider one more example to show how the global variable will be affected by the compiler optimization in a multi threaded application. The example code snippet is shown below:

```
#define FALSE 0
#define TRUE 1
volatile unsigned intglobal_item_count;
//Other functions
voidthread_one(void)
{
global_item_count = FALSE;
while(global_item_count == FALSE)
{  sleep(1);}
//Some code goes here
}
voidthread_two(void)
{
//some code goes here
global_item_count++;
```

```
sleep(5);
//some code goes here
}
```

In the above demo program, the compiler doesn't have any knowledge of the context switching between two threads. If the compiler optimizations are turned 'ON', then the compiler will assume that the global_item_count variable is always 'ZERO' and no other part of the thread is attempting to modify it. So, the compiler may replace the while loop in the code above, as shown in the code below:

```
. . . while(TRUE)
{ sleep(1);}
```

which is nothing but the infinite loop; so in order to avoid such optimizations by the compiler, it is safe to declare the variable global_item_count as 'volatile'. Similarly, one can realize the effect of the producer consumer problem accessing the global variable without declaring it as 'volatile'.

3. Interrupt service routines Let us consider another example given in in the code snippet below, where 'volatile' plays a very important role in the ISR (Interrupt Service Routines):

```
int flag = 0;
voidrx_isr(void)
{ flag = 1; }
int main()
    {while(!flag) { //Some code goes here } ... }
```

In the above example, if the flag is not declared as 'volatile', the compiler may optimize the code (assuming always that the flag is ZERO) and replace the while(!flag) to while(TRUE), which is nothing but the infinite loop. But the flag value might change when the interrupt occurs.

Note : Whether to declare the variable as 'volatile' or not is cross-compiler dependent. Anyhow it is a good practice to declare the variable as 'volatile' to achieve the portability of the code.

A variable should be declared volatile whenever its value can change asynchronously. In real time, three types of variables can change:

- *Memory-mapped peripheral registers (e.g., polling and waiting).*
- *Global variables modified by an Interrupt Service Routine.*
- *Global variables accessed by multiple tasks within a multi-threaded.*

IV. CONCLUSION

The main use of the 'volatile' key word is to prevent the compiler from optimizing the code in terms of time complexity, by generating a code that uses CPU registers as faster ways to represent variables. Declaring the variable as 'volatile' forces compiled code to access the exact memory location in RAM on every access to the variable to get its latest value, thereby avoiding any runtime surprises for the programmer.

REFERENCES

[1] http://www.geeksforgeeks.org/
[2] The 8051 Microcontroller (Merrill's international series in engineering technology)
[3] http://www.tldp.org/LDP/abs/abs-guide.pdf-Bash Scripting Guide
[4] http://www.tldp.org/LDP/abs/html/
[5] askubuntu.com/