DCL17-C. Beware of miscompiled volatile-qualified variables

As described in depth in rule DCL22-C. Use volatile for data that cannot be cached, a volatile-qualified variable "shall be evaluated strictly according to the rules of the abstract machine" [ISO/IEC 9899:2011]. In other words, the volatile qualifier is used to instruct the compiler to not make caching optimizations about a variable.

However, as demonstrated in "Volatiles Are Miscompiled, and What to Do about It" [Eide and Regehr], all tested compilers generated some percentage of incorrect compiled code with regard to volatile accesses. Therefore, it is necessary to know how your compiler behaves when the standard volatile behavior is required. The authors also provide a workaround that eliminates some or all of these errors.

Noncompliant Code Example

As demonstrated in Eide and Regehr's work, the following code example compiles incorrectly using GCC 4.3.0 for IA32 and the -O0 optimization flag:

```c
const volatile int x;
volatile int y;
void foo(void) {
    for(y = 0; y < 10; ++y) {
        int z = x;
    }
}
```

Because the variable `x` is volatile-qualified, it should be accessed 10 times in this program. However, as shown in the compiled object code, it is accessed only once due to a loop-hoisting optimization [Eide and Regehr]:

```assembly
foo:
    movl $0, y
    movl x, %eax
    jmp .L2
.L3:
    movl y, %eax
    incl %eax
    movl %eax, y
.L2:
    movl y, %eax
    cmpl $10, %eax
    jg .L3
    ret
```

Should `x` represent a hardware register or some other memory-mapped device that has side effects when accessed, the previous miscompiled code example may produce unexpected behavior.

Compliant Solution

Eide and Regehr tested a workaround by wrapping volatile accesses with function calls. They describe it with the intuition that "we can replace an action that compilers empirically get wrong by a different action—a function call—that compilers can get right" [Eide and Regehr]. For example, the workaround for the noncompliant code example would be

```c
int vol_read_int(volatile int *vp) {
    return *vp;
}
volatile int *vol_id_int(volatile int *vp) {
    return vp;
}

const volatile int x;
volatile int y;
void foo(void) {
    for(*vol_id_int(&y) = 0; vol_read_int(&y) < 10; *vol_id_int(&y) = vol_read_int(&y) + 1) {
        int z = vol_read_int(&x);
    }
}```
The workarounds proposed by Eide and Regehr fix many of the volatile-access bugs in the tested compilers. However, compilers are always changing, so critical sections of code should be compiled as if for deployment, and the compiled object code should be inspected for the correct behavior.

**Risk Assessment**

The volatile qualifier should be used with caution in mission-critical situations. Always make sure code that assumes certain behavior when using the volatile qualifier is inspected at the object code level for compiler bugs.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL17-C</td>
<td>Medium</td>
<td>Probable</td>
<td>High</td>
<td>P4</td>
<td>L3</td>
</tr>
</tbody>
</table>

**Automated Detection**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>134 S</td>
<td>Partially implemented</td>
</tr>
</tbody>
</table>

**Bibliography**

[Eide and Regehr] "Volatiles Are Miscompiled, and What to Do about It"