

# **Introduction to memory order consume**

*2015*

*issue.hsu@gmail.com*

# Outline

- Quick recap of acquire and release semantics
- The purpose of consume semantics
- Today's compiler support

# **Quick Recap of Acquire and Release Semantics**

# Memory Order

- In the C++11 standard atomic library, most functions accept a memory\_order argument

```
enum memory_order {  
    memory_order_relaxed,  
    memory_order_consume,  
    memory_order_acquire,  
    memory_order_release,  
    memory_order_acq_rel,  
    memory_order_seq_cst  
};
```

- Both consume and acquire serve the same purpose
  - To help pass non-atomic information safely between threads
- Like acquire operations, a consume operation must be combined with a **release** operation in another thread

# Example of Acquire and Release

- Declare two shared variables

```
atomic<int> Guard(0);  
int Payload = 0;
```

- The main thread sits in a loop, repeatedly attempting the following sequence of read operations

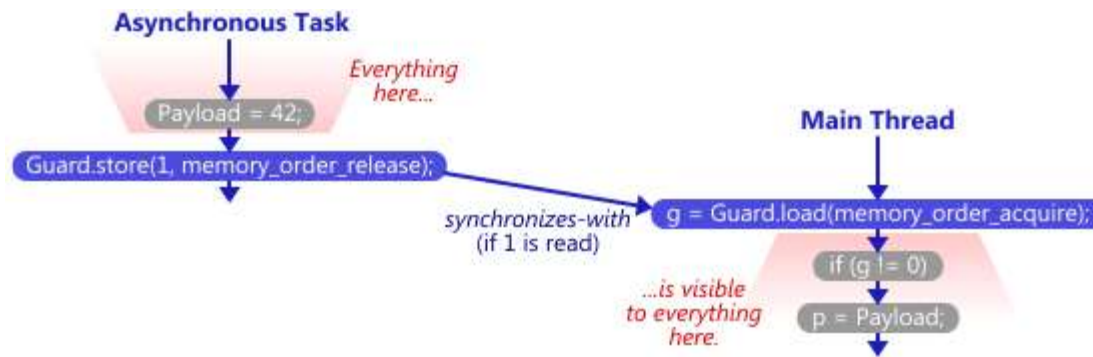
```
for(...)  
{  
    ....  
    g = Guard.load(memory_order_acquire);  
    if (g != 0)  
        p = Payload;  
    ...  
}
```

- Another asynchronous task running in another thread try to do a write operation

```
Payload = 42;  
Guard.store(1, memory_order_release);
```

# Example of Acquire and Release

- Once the asynchronous task writes to Guard, the main thread reads it
  - It means that the **write-release** **synchronized-with** the **read-acquire**
  - We are guaranteed that p will equal 42, no matter what platform we run this example on



- We've used acquire and release semantics to pass a simple non-atomic integer Payload between threads

# The Cost of Acquire Semantics

```
g = Guard.load(memory_order_acquire);
if (g != 0)
    p = Payload;
```

**Intel x86-64**  
Quad-core MacBook Pro



Intel Core i7 @ 2.3 GHz  
Compiled with Clang 4.6

**PowerPC**  
Quad-core PowerMac G5

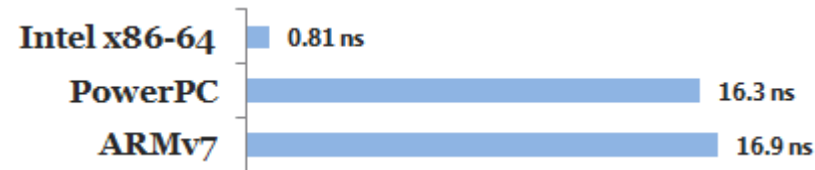


PowerPC 970MP @ 2.5 GHz  
Compiled with GCC 4.8.3

**ARMv7**  
Dual-core iPhone 4S



ARM Cortex-A9 @ 850 MHz  
Compiled with Clang 4.6



strong memory model

weakly-ordered CPU

```
mov    ecx, dword ptr [rip + _Guard]  // load from Guard
test   ecx, ecx
cmovne eax, dword ptr [rip + _Payload] // load from Payload
```

```
lis    r8, Guard@ha
addi   r8, r8, Guard@l
lis    r7, Payload@ha
lwz    r9, 0(r8) // load from Guard
cmpw   cr7, r9, r9 // memory barrier
bne-   cr7, $+4
isync
cmpwi   cr7, r9, 0
beq-    cr7, .L0
lwz    r10, Payload@l(r7) // load from Payload
.L0:
```

```
movw   r9, :lower16:(_Guard-(L0+4))
movt   r9, :upper16:(_Guard-(L0+4))
movw   r3, :lower16:(_Payload-(L1+4))
movt   r3, :upper16:(_Payload-(L1+4))
.L0:
add     r9, pc
.L1:
add     r3, pc // load from Guard
ldr.w   r4, [r9] // memory barrier
dmb     ish
ldr     r5, [r3] // load from Payload
cmp     r4, #0
it      ne
movne   r2, r5
```

# **The Purpose of Consume Semantics**



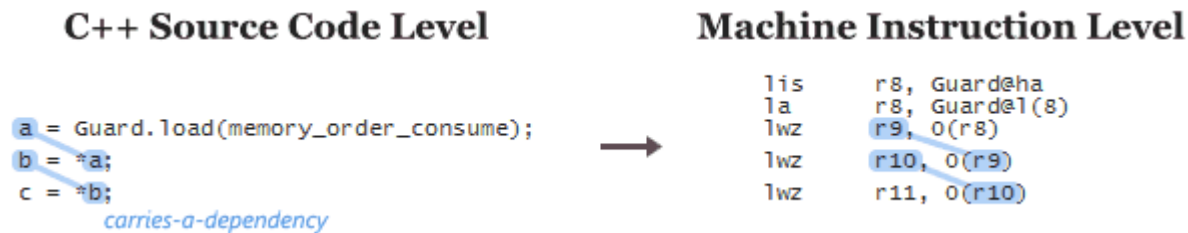
# Data Dependency

- The PowerPC and ARM are weakly-ordered CPUs, but in fact, there are some cases where they do enforce memory ordering at the machine instruction level without the need for explicit memory barrier instructions
  - These processors always preserve memory ordering between **data-dependent** instructions
- When multiple instructions are data-dependent on each other, we call it a **data dependency chain**
  - In the following PowerPC listing, there are two independent data dependency chains

```
lwz      r9, 8(r3)
lwz      r10, 8(r4)
addi     r11, r9, 7
addi     r12, r10, 12
rlwinm   r9, r11, 2, 0, 29
rlwinm   r10, r12, 2, 0, 29
lwzx     r8, r5, r9
lwzx     r9, r5, r10
```

# Data Dependency

- Consume semantics are designed to exploit the data dependency ordering



- At the source code level, a dependency chain is a sequence of expressions whose evaluations all **carry-a-dependency** to each another
  - Carries-a-dependency** is defined in [§ 1.10.9](#) of the C++11 standard
  - It mainly says that one evaluation carries-a-dependency to another if the value of the first is used as an operand of the second

# Example of Consume and Release

- Declare two shared variables

```
atomic<int> Guard(0);  
int Payload = 0;
```



```
atomic<int*> Guard(nullptr);  
int Payload = 0;
```

- The main thread sits in a loop, repeatedly attempting the following sequence of read operations

```
for(...)  
{  
    ...  
    g = Guard.load(memory_order_acquire);  
    if (g != 0)  
        p = Payload;  
    ...  
}
```



```
for(...)  
{  
    ...  
    g = Guard.load(memory_order_consume);  
    if (g != nullptr)  
        p = *g;  
    ...  
}
```

- Another asynchronous task running in another thread try to do a write operation

```
Payload = 42;  
Guard.store(1, memory_order_release);
```



```
Payload = 42;  
Guard.store(&Payload, memory_order_release);
```

# Example of Consume and Release

- This time, we don't have a **synchronizes-with** relationship anywhere. What we have this time is called a **dependency-ordered-before** relationship



- In any dependency-ordered-before relationship, there's a dependency chain starting at the consume operation, and all memory operations performed before the write-release are guaranteed to be visible to that chain.

# Example of Consume and Release

**Intel x86-64**  
Quad-core MacBook Pro



Intel Core i7 @ 2.3 GHz  
Compiled with Clang 4.6

**PowerPC**  
Quad-core PowerMac G5



PowerPC 970MP @ 2.5 GHz  
Compiled with GCC 4.8.3

**ARMv7**  
Dual-core iPhone 4S



ARM Cortex-A9 @ 850 MHz  
Compiled with Clang 4.6

```

mov rcx, qword ptr [rip + _Guard]
test rcx, rcx
je LO
mov eax, dword ptr [rcx]
    
```

Annotations:  
 - `rcx, qword ptr [rip + _Guard]`: load from Guard  
 - `[rcx]`: load from \*g

```

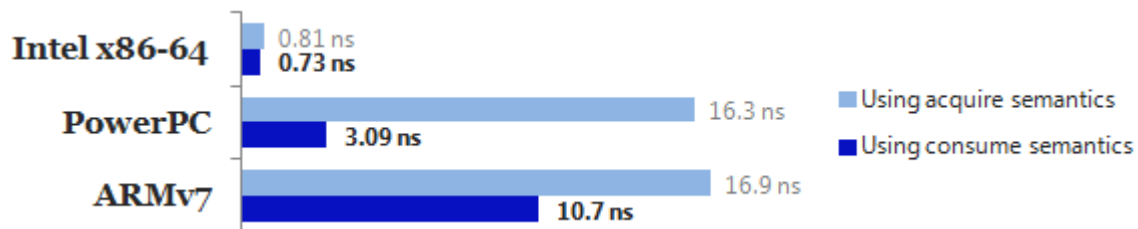
.LO:
lis r8, Guard@ha
la r8, Guard@l(8)
lwz r9, 0(r8)
cmpw cr7, r9, 0
beq- cr7, .LO
lwz r10, 0(r9)
    
```

Annotations:  
 - `r8, Guard@ha` and `la r8, Guard@l(8)`: load from Guard  
 - `r9, 0(r8)`: load from \*g  
 - `r10, 0(r9)`: load from \*g

```

movw r3, :lower16:(_Guard-(LO+4))
movt r3, :upper16:(_Guard-(LO+4))
.LO:
add r3, pc
ldr r4, [r3]
cmp r4, #0
it ne
ldrne r2, [r4]
    
```

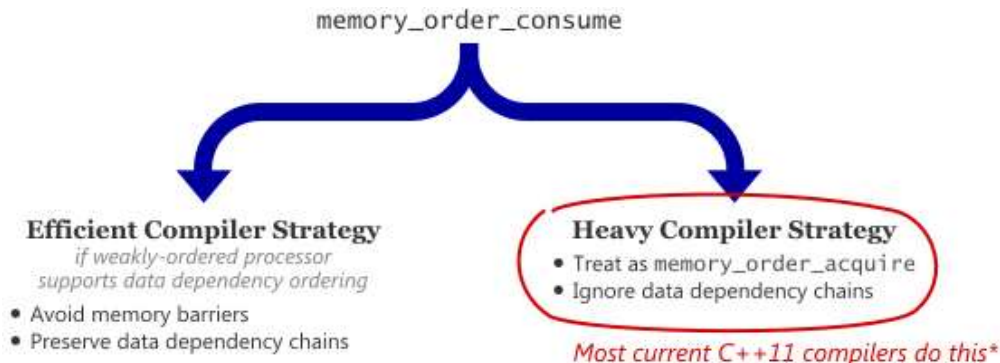
Annotations:  
 - `[r3]`: load from Guard  
 - `[r4]`: load from \*g



# **Today's Compiler Support**

# Current Compiler Status

- Those assembly code listings just showed you for PowerPC and ARMv7 were fabricated
  - Sorry, but GCC 4.8.3 and Clang 4.6 don't actually generate that machine code for consume operations



*Most current C++11 compilers do this\**

- Current versions of GCC and Clang/LLVM use the heavy strategy, all the time
  - As a result, if you compile `memory_order_consume` for PowerPC or ARMv7 using today's compilers, you'll end up with unnecessary memory barrier instructions

# Efficient Compiler Strategy in GCC

- GCC 4.9.2 actually has an efficient compiler strategy in its implementation of `memory_order_consume`, as described in this [GCC bug report](#)
  - Only available in GCC 4.9.2 AARCH64 target



# Example That Illustrates the Compiler Bug

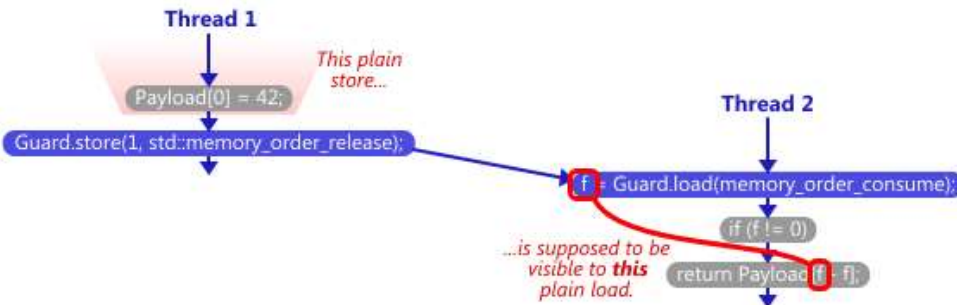
```
#include <atomic>

std::atomic<int> Guard(0);
int Payload[1] = { 0xbadf00d };

int read()
{
    int f = Guard.load(std::memory_order_consume); // load-consume
    if (f != 0)
        return Payload[f - f]; // plain load from Payload[f - f]
    return 0;
}

int write()
{
    Payload[0] = 42; // plain store to Payload[0]
    Guard.store(1, std::memory_order_release); // store-release
}
```

\$ aarch64-linux-g++ -std=c++11 -O2 -S consumetest.cpp



```
_Z4readv:
.LF8327:
.cfi_startproc
adrp    x0, .LANCHOR1
add     x0, x0, :lo12:.LANCHOR1
ldr     w1, [x0]
mov     w0, 0
cbz     w1, .L3
adrp    x0, .LANCHOR0
ldr     w0, [x0, #:lo12:.LANCHOR0]
ret
.cfi_endproc
```

*no memory barrier!*

*load from Guard*

*load from Payload[f - f]*

*separate dependency chains!*

- In this example, we are admittedly abusing C++11's definition of carry-a-dependency by using  $f$  in an expression that cancels it out ( $f - f$ ). Nonetheless, we are still technically playing by the standard's current rules, and thus, its ordering guarantees should still apply

# A Patch for This Bug


- Andrew Macleod [posted a patch](#) for this issue in the bug report. His patch adds the following lines near the end of the `get_memmodel` function in [gcc/builtins.c](#)

```
/* Workaround for Bugzilla 59448. GCC doesn't track consume properly, so  
be conservative and promote consume to acquire. */  
if (val == MEMMODEL_CONSUME)  
    val = MEMMODEL_ACQUIRE;
```

- After patching
  - `$ aarch64-linux-g++ -std=c++11 -O2 -S consumetest.cpp`

```
    _Z4readv:  
    .LFB327:  
        .cfi_startproc  
        adrp    x0, .LANCHOR1  
        add     x0, x0, :1012:.LANCHOR1  
        ldar    w1, [x0]          ← load from Guard  
        mov     w0, 0  
        cbz     w1, .L3  
        adrp    x0, .LANCHOR0  
        ldr     w0, [x0, #:1012:.LANCHOR0] ← load from Payload[f - f]  
    .L3:  
        ret  
        .cfi_endproc
```

*"load-acquire" instruction acts as a memory barrier*



# This Bug Doesn't Happen on PowerPC

- Interestingly, if you compile the same example for PowerPC, there is no bug. This is using the same GCC version 4.9.2 without Andrew's patch applied
  - \$ powerpc-linux-g++ -std=c++11 -O2 -S consumetest.cpp

```

_Z4readv:
.LFB327:
    .cfi_startproc
    lis 9,Guard@ha
    lwz 9,Guard@l(9)
    cmpw 7,9,9
    bne- 7,$+4
    isync
    li 3,0
    cmpwi 7,9,0
    beqlr- 7
    lis 9,Payload@ha
    lwz 3,Payload@l(9)
    b1r
    .cfi_endproc

```

Annotations:

- load from Guard (points to `lis 9,Guard@ha`)
- memory barrier (points to `isync`)
- load from Payload[f - f] (points to `lwz 3,Payload@l(9)`)

gcc-4.9.2/gcc/config/aarch64/atomics.md

```

if (model == MEMMODEL_RELAXED
    || model == MEMMODEL_CONSUME
    || model == MEMMODEL_RELEASE)
    return "ldr<atomic_sfx>\t%<w>0, %1";
else
    return "ldar<atomic_sfx>\t%<w>0, %1";

```

gcc-4.9.2/gcc/config/rs6000/sync.md

```

switch (model)
{
case MEMMODEL_RELAXED:
    break;
case MEMMODEL_CONSUME:
case MEMMODEL_ACQUIRE:
case MEMMODEL_SEQ_CST:
    emit_insn (gen_loadsync_<mode> (operands[0]));
    break;
}

```

# The Uncertain Future of memory\_order\_consume

- The C++ standard committee is wondering what to do with memory\_order\_consume in future revisions of C++
- The author's opinion is that [the definition of carries-a-dependency should be narrowed to require that different return values from a load-consume result in different behavior for any dependent statements that are executed](#)
  - Using  $f - f$  as a dependency is nonsense, and narrowing the definition would free the compiler from having to support such nonsense “dependencies” if it chooses to implement the efficient strategy
  - This idea was first proposed by Torvald Riegel [in the Linux Kernel Mailing List](#) and is captured among various alternatives described in Paul McKenney's [proposal N4036](#)

```
int my_array[MY_ARRAY_SIZE];  
  
i = atomic_load_explicit(gi, memory_order_consume);  
r1 = my_array[i];
```

# References

# References

- The Purpose of `memory_order_consume` in C++11
  - [http://preshing.com/20140709/the-purpose-of-memory\\_order\\_consume-in-cpp11/](http://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/)
- Fixing GCC's Implementation of `memory_order_consume`
  - [http://preshing.com/20141124/fixing-gccs-implementation-of-memory\\_order\\_consume/](http://preshing.com/20141124/fixing-gccs-implementation-of-memory_order_consume/)
- [http://en.cppreference.com/w/cpp/atomic/memory\\_order](http://en.cppreference.com/w/cpp/atomic/memory_order)
- Bug 59448 - Code generation doesn't respect C11 address-dependency
  - [https://gcc.gnu.org/bugzilla/show\\_bug.cgi?id=59448](https://gcc.gnu.org/bugzilla/show_bug.cgi?id=59448)
- N4036: Towards Implementation and Use of `memory_order_consume`
  - <https://isocpp.org/files/papers/n4036.pdf>
- Demo program
  - <https://github.com/preshing/ConsumeDemo>