Psychosomatic, Lobotomy, Saw

It's X, you'll need Y, I'll get Z

Wednesday, 29 October 2014

The JVM Write Barrier - Card Marking

In Java, not all value stores are created equal, in particular storing object references is different to storing primitive values. This makes perfect sense when we consider that the JVM is a magical place where object allocation, relocation and deletion are somebody else's problem. So while in theory writing a reference field is the same as writing the same sized primitive (an int on 32bit JVMs or with compressed oops on, or a long otherwise) in practice some accounting takes place to support GC. In this post we'll have a look at one such accounting overhead, the write barrier.

What's an OOP?

An OOP (Ordinary Object Pointer) is the way the JVM views Java object references. They are pointer representations rather than actual pointers (though they may be usable pointers). Since objects are managed memory OOPs reads/writes may require a memory barrier of the memory management kind (as opposed to the JMM ordering barrier kind):

> "A barrier is a block on reading from or writing to certain memory locations by certain threads or processes.
>
> Barriers can be implemented in either software or hardware. Software barriers involve additional instructions around load or store operations, which would typically be added by a cooperative compiler. Hardware barriers don't require compiler support, and may be implemented on common operating systems by using memory protection."
>
> \- Memory Management Reference, Memory Barrier
>
> "Write barriers are used for incremental or concurrent garbage collection. They are also used to maintain remembered sets for generational collectors."
>
> \- Memory Management Reference, Write Barrier

In particular this means card marking.

Card Marking

All modern JVMs support a generational GC process, which works under the assumption that allocated objects mostly live short and careless lives. This assumption leads to GC algorithm where different generations are treated differently, and where cross generation references pose a challenge. Now imagine the time to collect the young generation is upon our JVM, what do we need to do to determine which young objects are still alive (ignoring the Phantom/Weak/Soft reference debate and finalizers)?

- An object is alive if it is referenced by a live object.
- An object is alive if a static reference to it exists (part of the root set).
- An object is alive if a stack reference to it exists (part of the root set).

The GC process therefore:

> "Tracing garbage collectors, such as copying, mark-sweep, and mark-compact, all start scanning from the root set, traversing references between objects, until all live objects have been visited.
>
> A generational tracing collector starts from the root set, but does not traverse references that lead to objects in the older generation, which reduces the size of the object graph to be traced. But this creates a problem -- what if an object in the older generation references a younger object, which is not reachable through any other chain of references from a root?" - Brian Goetz, GC in the HotSpot JVM

It is worth reading the whole article to get more context on the cross generational reference problem, but the solution is card marking:

> "...the heap is divided into a set of cards, each of which is usually smaller than a memory page. The JVM maintains a card map, with one bit (or byte, in some implementations) corresponding to each card in the heap. Each time a pointer field in an object in the heap is modified, the corresponding bit in the card map for that card is set."

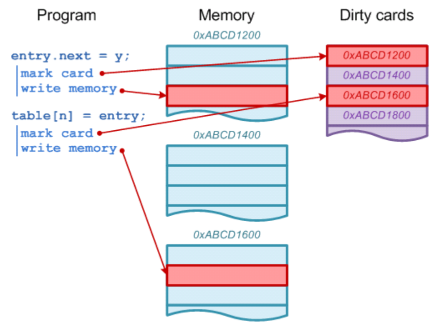

Illustration By Alexey Ragozin

A good explanation of card marking is also given here by Alexey Ragozin. I have taken liberty to include his great illustration of the process.
So there you have it, every time an object reference is updated the compiler has to inject some accounting logic towards card marking. So how does this effect the code generated for your methods?

Default Card Marking

Search This Blog

Search

Pages

- Lock Free Queues Index
- JMH Resources
- Talks
- About

Popular Posts

Atomic*.lazySet is a performance win for single writers

Why (Most) Sampling Java Profilers Are Fucking Terrible

Safepoints: Meaning, Side Effects and Overheads

Linked Array Queues, part 2: SPSC Benchmarks

The JVM Write Barrier - Card Marking

How Inlined Code Makes For Confusing Profiles

The Pros and Cons of AsyncGetCallTrace Profilers

JMH perfasm explained: Looking at False Sharing on Conditional Inlining

135 Million messages a second between processes in pure Java

Java Flame Graphs Introduction: Fire For Everyone!

Blog Archive

- ► 2018 (2)
- ► 2017 (1)
- ► 2016 (9)
- ► 2015 (11)
- ▼ 2014 (18)
  - ► December (1)
  - ► November (1)
  - ▼ October (2)
    - The JVM Write Barrier - Card Marking
    - Celebrating 2 years of blogging!
  - ► August (3)
  - ► July (2)
  - ► June (2)
  - ► April (1)
  - ► March (2)
  - ► February (2)
  - ► January (2)
- ► 2013 (23)
- ► 2012 (6)

About Me

OpenJDK/Oracle 1.6/1.7/1.8 JVMs default to the following card marking logic (assembly for a setter such as setFoo(Object bar)):

```
; rsi is 'this' address
; rdx is setter param, reference to bar
; JDK6:
mov    QWORD PTR [rsi+0x20],rdx  ; this.foo = bar
mov    r10,rsi                   ; r10 = rsi = this
shr    r10,0x9                   ; r10 = r10 >> 9;
mov    r11,0x7ebdfcff7f00        ; r11 is base of card table, imagine byte[] CARD_TABLE
mov    BYTE PTR [r11+r10*1],0x0  ; Mark 'this' card as dirty, CARD_TABLE[this address >> 9] = 0

; JDK7(same):
mov    QWORD PTR [rsi+0x20],rdx
mov    r10,rsi
shr    r10,0x9
mov    r11,0x7f6d852d7000
mov    BYTE PTR [r11+r10*1],0x0

; JDK8:
mov    QWORD PTR [rsi+0x20],rdx
shr    rsi,0x9                   ; clever JIT noticed RSI is not in use later, so shifting in plac
mov    rdi,0x7f2d42817000
mov    BYTE PTR [rsi+rdi*1],0x0

```

gistfile1.asm hosted with ❤ by **GitHub** **view raw**

So setting a reference throws in the overhead of a few instructions, which boil down to:

CARD_TABLE [this address >> 9] = 0;

This is significant overhead when compared to primitive fields, but is considered necessary tax for memory management. The tradeoff here is between the benefit of card marking (limiting the scope of required old generation scanning on young generation collection) vs. the fixed operation overhead for all reference writes. The associated write to memory for card marking can sometimes cause performance issues for highly concurrent code. This is why in OpenJDK7 we have a new option called UseCondCardMark.
[UPDATE: as JP points out in the comments, the (>> 9) is converting the address to the relevant card index. Cards are 512 bytes in size so the shift is in fact address/512 to find the card index.]

Conditional Card Marking

This is the same code run with -XX:+UseCondCardMark:

```
; rsi is 'this' address
; rdx is setter param, reference to bar
; JDK7:
0x00007fc4a1071d5c: mov    r10,rsi                   ; r10 = this
0x00007fc4a1071d5f: shr    r10,0x9                   ; r10 = r10 >> 9
0x00007fc4a1071d63: mov    r11,0x7f7cb98f7000        ; r11 = CARD_TABLE
0x00007fc4a1071d6d: add    r11,r10                   ; r11 = CARD_TABLE + (this >> 9)
0x00007fc4a1071d70: movsx  r8d,BYTE PTR [r11]        ; r8d = CARD_TABLE[this >> 9]
0x00007fc4a1071d74: test   r8d,r8d
0x00007fc4a1071d77: je     0x00007fc4a1071d7d        ; if(CARD_TABLE[this >> 9] == 0) goto 0x00007
0x00007fc4a1071d79: mov    BYTE PTR [r11],0x0        ; CARD_TABLE[this >> 9] = 0
0x00007fc4a1071d7d: mov    QWORD PTR [rsi+0x20],rdx  ; this.foo = bar

```

gistfile1.asm hosted with ❤ by **GitHub** **view raw**

Which boils down to:

if (CARD_TABLE [this address >> 9] != 0) CARD_TABLE [this address >> 9] = 0;

This is a bit more work, but avoids the potentially concurrent writes to the card table, thus side stepping some potential false sharing through minimising recurring writes. I have been unable to make JDK8 generate similar code with the same flag regardless of which GC algorithm I run with (I can see the code in the OJDK codebase... not sure what's the issue, feedback/suggestions/corrections welcome).

Card Marking G1GC style?

Is complicated... have a look:

```
  movsx  edi,BYTE PTR [r15+0x2d0] ; read some GC flag
  cmp    edi,0x0; if (flag != 0)
  jne    0x00000001066fc601; GOTO OldValBarrier
Label WRITE:
  mov    QWORD PTR [rsi+0x20],rdx; this.foo = bar
  mov    rdi,rsi; rdi = this
```

Nitsan

View my complete profile

Subscribe To

Posts

Comments

```
  xor    rdi,rdx; rdi = this XOR bar
  shr    rdi,0x14; rdi = (this XOR bar) >> 20
  cmp    rdi,0x0; If this and bar are not same gen
  jne    0x00000001066fc616; GOTO NewValBarrier
Label EXIT:
;…

Label OldValBarrier:
  mov    rdi,QWORD PTR [rsi+0x20]
  cmp    rdi,0x0; if(this.foo == null)
  je     0x00000001066fc5dd; GOTO WRITE
  mov    QWORD PTR [rsp],rdi ; setup rdi as parameter
  call   0x000000010664bca0  ;   {runtime_call}
  jmp    0x00000001066fc5dd; GOTO WRITE
Label NewValBarrier:
  cmp    rdx,0x0; bar == null
  je     0x00000001066fc5f5 goto Exit
  mov    QWORD PTR [rsp],rsi
  call   0x000000010664bda0  ;   {runtime_call}
  jmp    0x00000001066fc5f5  ; GOTO exit;
```

gistfile1.asm hosted with ❤ by **GitHub** **view raw**

To figure out exactly what this was about I had to have a read in the Hotspot codebase. A rough translation would be:

```
oop oldFooVal = this.foo;
if (GC.isMarking != 0 && oldFooVal != null){
  g1_wb_pre(oldFooVal);
}
this.foo = bar;
if ((this ^ bar) >> 20) != 0 && bar != null) {
  g1_wb_post(this);
}
```

The runtime calls are an extra overhead whenever we are unlucky enough to either:

1. Write a reference while card marking is in process (and old value was not null)
2. Target object is 'older' than new value (and new value is not null) **[UPDATE: (SRC^TGT>>20 != 0) is a cross region rather than a cross generational check. Thanks Gleb!]**

The interesting point to me is that the generated assembly ends up being somewhat fatter (nothing like your mamma) and has a significantly worse 'cold' case (cold as in less likely to happen), so in theory mixing up the generations will be painful.

Summary

Writing references incurs some overhead not present for primitive values. The overhead is in the order of a few instructions which is significant when compared to primitive types, but minor when we assume most applications read more than they write and have a healthy data/object ratio. Estimating the card marking impact is non-trivial and I will be looking to benchmark it in a later post. For now I hope the above helps you recognise card marking logic in your print assembly output and sheds some light on what the write barrier and card marking is about.

at 09:31:00

Labels: Assembly, GC, Java, JVM Internals, JVM Options

8 comments:

Jean-Philippe Bempel 29 Oct 2014, 10:26:00

Hi Nitsan,
Good stuff as always. And even better when using Intel syntax for assembly ! :)

However just a quick remark about the 9 bit shift. I would explain why there is this shift, and relation with size of the card (512 bytes 2^9). But maybe you assume the reader should have read all referenced articles before... :)

Reply

Replies

Nitsan 29 Oct 2014, 11:15:00

Thanks :)
Added note at relevant paragraph, you are now earmarked to review next post ;-)

Reply

maaartinus 31 Dec 2015, 17:53:00

Good stuff, indeed. Could you explain, why g1_wb_pre is needed? It handles the case when a reference disappears, which doesn't seem important enough as treating a single object which recently became unreachable as reachable is just a small overhead when compared to such a test done on many reference stores.

Reply

Replies

Nitsan 6 Jan 2016, 08:46:00

I am probably not the best person to ask... I'll see if I can get a G1GC person to comment here.

gvsmirnov 6 Jan 2016, 11:21:00

That's because the concurrent marking in G1 is snapshot-at-the-beginning. It marks the objects which were live *at the beginning* of the marking cycle. So, if mutators overwrite a field, the previous value is stored for concurrent marking. No need to store nulls, though.

See section 2.5.3 of the original G1 paper by Detlefs, Flood, Heller, Printezis for details.

Nitsan 6 Jan 2016, 12:31:00

For a tall person you sure know allot about GC...

Monica Beckwith 6 Jan 2016, 15:51:00

+1. Objects that are a part of the snapshot (and reachable) could get overwritten before they get traced. Hence SATB guarantee requires the mutator to log the previous value of the pointer in a log buffer. This needs to be done before the write hence it's called a pre-write barrier. The log buffers are processed at regular intervals.

maaartinus 8 Jan 2016, 21:52:00

I see I was thinking in exactly the opposite direction. Thank you, now it's pretty clear.

Reply

Enter your comment...

Comment as: Google Accoun

Publish **Preview**

Note: only a member of this blog may post a comment.

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Simple theme. Powered by Blogger.