Mechanical Sympathy

Hardware and software working together in harmony

Thursday, 27 June 2013

Printing Generated Assembly Code From The Hotspot JIT Compiler

Sometimes when profiling a Java application it is necessary to understand the assembly code generated by the Hotspot JIT compiler. This can be useful in determining what optimisation decisions have been made and how our code changes can affect the generated assembly code. It is also useful at times knowing what instructions are emitted when debugging a concurrent algorithm to ensure visibility rules have been applied as expected. I have found quite a few bugs in various JVMs this way.

This blog illustrates how to install a Disassembler Plugin and provides command line options for targeting a particular method.

Installation

Previously it was necessary to obtain a debug build for printing the assembly code generated by the Hotspot JIT for the Oracle/SUN JVM. Since Java 7, it has been possible to print the generated assembly code if a Disassembler Plugin is installed in a standard Oracle Hotspot JVM. To install the plugin for 64-bit Linux follow the steps below:

1. Download the appropriate binary, or build from source, from https://kenai.com/projects/base-hsdis/downloads
2. On Linux rename `linux-hsdis-amd64.so` to `libhsdis-amd64.so`
3. Copy the shared library to `$JAVA_HOME/jre/lib/amd64/server`

You now have the plugin installed!

Test Program

To test the plugin we need some code that is both interesting to a programmer and executes sufficiently hot to be optimised by the JIT. Some details of when the JIT will optimise can be found here. The code below can be used to measure the average latency between two threads by reading and writing `volatile` fields. These `volatile` fields are interesting because they require associated hardware fences to honour the Java Memory Model.

```
import static java.lang.System.out;

public class InterThreadLatency
{
    private static final int REPETITIONS = 100 * 1000 * 1000;

    private static volatile int ping = -1;
    private static volatile int pong = -1;

    public static void main(final String[] args)
        throws Exception
    {
        for (int i = 0; i < 5; i++)
        {
            final long duration = runTest();

            out.printf("%d - %dns avg latency - ping=%d pong=%d\n",
                       i,
                       duration / (REPETITIONS * 2),
                       ping,
                       pong);
        }
    }

    private static long runTest() throws InterruptedException
    {
        final Thread pongThread = new Thread(new PongRunner());
        final Thread pingThread = new Thread(new PingRunner());
        pongThread.start();
        pingThread.start();

        final long start = System.nanoTime();
        pongThread.join();

        return System.nanoTime() - start;
    }

    public static class PingRunner implements Runnable
    {
        public void run()
```

Search This Blog

Search

Discussion Group

https://groups.google.com/forum/#!forum/mechanical-sympathy

About Me

Martin Thompson

London, United Kingdom

Technology geek exploring the capabilities of modern hardware. Available for development, training, performance tuning, and consulting services via Real Logic Limited. Twitter: @mjpt777

View my complete profile

Training & Consulting

www.real-logic.co.uk

Next public course

Conferences & Workshops

- QCon London March
- JavaLand Brühl - March
- Craft Conf Budapest - April
- JOnTheBeach Málaga - May
- Goto Amsterdam - June
- JCrete - July

Popular Posts

Java Garbage Collection Distilled

Serial, Parallel, Concurrent, CMS, G1, Young Gen, New Gen, Old Gen, Perm Gen, Eden, Tenured, Survivor Spaces, Safepoints, and the hundreds ...

CPU Cache Flushing Fallacy

Even from highly experienced technologists I often hear talk about how certain operations cause a CPU cache to "flush". This see...

Compact Off-Heap Structures/Tuples In Java

In my last post I detailed the implications of the access patterns your code takes to main memory. Since then I've had a lot of quest...

Memory Access Patterns Are Important

In high-performance computing it is often said that the cost of a cache-miss is the largest performance penalty for an algorithm. For many...

Native C/C++ Like Performance For Java Object Serialisation

Do you ever wish you could turn a Java object into a stream of bytes as fast as it can be done in

```
        {
            for (int i = 0; i < REPETITIONS; i++)
            {
                ping = i;

                while (i != pong)
                {
                    // busy spin
                }
            }
        }
    }

    public static class PongRunner implements Runnable
    {
        public void run()
        {
            for (int i = 0; i < REPETITIONS; i++)
            {
                while (i != ping)
                {
                    // busy spin
                }

                pong = i;
            }
        }
    }
}
```

Printing Assembly Code

It is possible to print all generated assembly code with the following statement.

```
java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly InterThreadLatency
```

However this can put you in the situation of not being able to see the forest for the trees. It is generally much more useful to target a particular method. For this test, the `run()` method will be optimised and generated twice by Hotspot. Once for the OSR version, and then again for the standard JIT version. The standard JIT version follows.

```
java -XX:+UnlockDiagnosticVMOptions '-XX:CompileCommand=print,*PongRunner.run'
InterThreadLatency

Compiled method (c2)   10531    5             InterThreadLatency$PongRunner::run (30 bytes)
 total in heap  [0x00007fed81060850,0x00007fed81060b30] = 736
 relocation     [0x00007fed81060970,0x00007fed81060980] = 16
 main code      [0x00007fed81060980,0x00007fed81060a00] = 128
 stub code      [0x00007fed81060a00,0x00007fed81060a18] = 24
 oops           [0x00007fed81060a18,0x00007fed81060a30] = 24
 scopes data    [0x00007fed81060a30,0x00007fed81060a78] = 72
 scopes pcs     [0x00007fed81060a78,0x00007fed81060b28] = 176
 dependencies   [0x00007fed81060b28,0x00007fed81060b30] = 8
Decoding compiled method 0x00007fed81060850:
Code:
[Entry Point]
[Constants]
  # {method} 'run' '()V' in 'InterThreadLatency$PongRunner'
  #           [sp+0x20]  (sp of caller)
  0x00007fed81060980: mov    0x8(%rsi),%r10d
  0x00007fed81060984: shl    $0x3,%r10
  0x00007fed81060988: cmp    %r10,%rax
  0x00007fed8106098b: jne    0x00007fed81037a60  ;   {runtime_call}
  0x00007fed81060991: xchg   %ax,%ax
  0x00007fed81060994: nopl   0x0(%rax,%rax,1)
  0x00007fed8106099c: xchg   %ax,%ax
[Verified Entry Point]
  0x00007fed810609a0: sub    $0x18,%rsp
  0x00007fed810609a7: mov    %rbp,0x10(%rsp)    ;*synchronization entry
                                                ; - InterThreadLatency$PongRunner::run@-1 (line
58)
  0x00007fed810609ac: xor    %r11d,%r11d
  0x00007fed810609af: mov    $0x7ad0fcbf0,%r10  ;   {oop(a 'java/lang/Class' =
'InterThreadLatency')}
  0x00007fed810609b9: jmp    0x00007fed810609d0
  0x00007fed810609bb: nopl   0x0(%rax,%rax,1)   ; OopMap{r10=Oop off=64}
                                                ;*goto
                                                ; - InterThreadLatency$PongRunner::run@15 (line
60)
  0x00007fed810609c0: test   %eax,0xaa1663a(%rip)        # 0x00007fed8ba77000
                                                ;*goto
                                                ; - InterThreadLatency$PongRunner::run@15 (line
60)
                                                ;   {poll}
  0x00007fed810609c6: nopw   0x0(%rax,%rax,1)   ;*iload_1
                                                ; - InterThreadLatency$PongRunner::run@8 (line
60)
```

a native language like C++? If you use s...

Simple Binary Encoding

Financial systems communicate by sending and receiving vast numbers of messages in many different formats. When people use terms like "...

Single Writer Principle

When trying to build a highly scalable system the single biggest limitation on scalability is having multiple writers contend for any item o...

Memory Barriers/Fences

In this article I'll discuss the most fundamental technique in concurrent programming known as memory barriers, or fences, that make th...

Fun with my-Channels Nirvana and Azul Zing

Since leaving LMAX I have been neglecting my blog a bit. This is not because I have not been doing anything interesting. Quite the opposi...

False Sharing

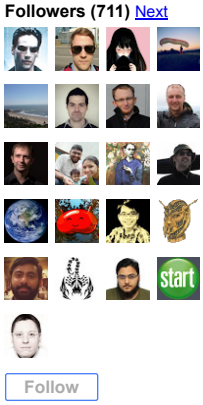
Memory is stored within the cache system in units know as cache lines. Cache lines are a power of 2 of contiguous bytes which are typically...

Blog Archive

- ▶ 2014 (1)
- ▼ 2013 (5)
  - ▶ August (1)
  - ▶ July (1)
  - ▼ June (1)
    - Printing Generated Assembly Code From The Hotspot ...
  - ▶ February (1)
  - ▶ January (1)
- ▶ 2012 (7)
- ▶ 2011 (19)

Followers

Followers (711) Next



Follow

```
  0x00007fed810609d0: mov    0x74(%r10),%r9d    ;*getstatic ping
                                                ; - InterThreadLatency::access$000@0 (line 3)
                                                ; - InterThreadLatency$PongRunner::run@9 (line
60)
  0x00007fed810609d4: cmp    %r9d,%r11d
  0x00007fed810609d7: jne    0x00007fed810609c0
  0x00007fed810609d9: mov    %r11d,0x78(%r10)
  0x00007fed810609dd: lock addl $0x0,(%rsp)     ;*putstatic pong
                                                ; - InterThreadLatency::access$102@2 (line 3)
                                                ; - InterThreadLatency$PongRunner::run@19 (line
65)
  0x00007fed810609e2: inc    %r11d              ;*iinc
                                                ; - InterThreadLatency$PongRunner::run@23 (line
58)
  0x00007fed810609e5: cmp    $0x5f5e100,%r11d
  0x00007fed810609ec: jl     0x00007fed810609d0  ;*if_icmpeq
                                                ; - InterThreadLatency$PongRunner::run@12 (line
60)
  0x00007fed810609ee: add    $0x10,%rsp
  0x00007fed810609f2: pop    %rbp
  0x00007fed810609f3: test   %eax,0xaa16607(%rip)        # 0x00007fed8ba77000
                                                ;   {poll_return}
  0x00007fed810609f9: retq                      ;*iload_1
                                                ; - InterThreadLatency$PongRunner::run@8 (line
60)
  0x00007fed810609fa: hlt
  0x00007fed810609fb: hlt
  0x00007fed810609fc: hlt
  0x00007fed810609fd: hlt
  0x00007fed810609fe: hlt
  0x00007fed810609ff: hlt
[Exception Handler]
[Stub Code]
  0x00007fed81060a00: jmpq   0x00007fed8105eaa0  ;   {no_reloc}
[Deopt Handler Code]
  0x00007fed81060a05: callq  0x00007fed81060a0a
  0x00007fed81060a0a: subq   $0x5,(%rsp)
  0x00007fed81060a0f: jmpq   0x00007fed81038c00  ;   {runtime_call}
  0x00007fed81060a14: hlt
  0x00007fed81060a15: hlt
  0x00007fed81060a16: hlt
  0x00007fed81060a17: hlt
OopMapSet contains 1 OopMaps

#0
OopMap{r10=Oop off=64}
```

An Interesting Observation

The red highlighted lines of assembly code above are very interesting. When a `volatile` field is written, under the Java Memory Model the write must be sequentially consistent, i.e. not appear to be reordered due to optimisations normally applied such as staging the write to the store buffer. This can be achieved by inserting the appropriate memory barriers. In the case above, Hotspot has chosen to enforce the ordering by issuing a MOV instruction (register to memory address - i.e. the write) followed by a LOCK ADD instruction (no op to the stack pointer as a fence idiom) that has ordering semantics. This could be less than ideal on an x86 processor. The same action could have been performed more efficiently and correctly with a single LOCK XCHG instruction for the write. This makes me wonder if there are some significant compromises in the JVM to make it portable across many architectures, rather than be the best it can on x86.

Posted by Martin Thompson at 20:24

Labels: ASM, Debugging, Hotspot, Java, JIT
Location: London, UK

5 comments:

Dmitry Zaslavsky 28 June 2013 at 00:52

I am not sure you would win anything. I am guessing top of the stack is already in the local cache in M state, so lock add won't cause any additional bus traffic. Previous move as a separate instruction can probably write combine with another store near by. I am guessing it won't write combine on xchng.

testing would show...

Reply

Michael Nitschinger 28 June 2013 at 08:01

For everyone on Mac, here is a quick guide to get ASM code spit out there as well: http://nitschinger.at/Printing-JVM-generated-Assembler-on-Mac-OS-X

Reply

Unknown 28 June 2013 at 19:53

Is LOCK XCHG really faster than MOV followed by LOCK ADDL? I could see it being slower as a result of x86 O-O-O dispatch, but maybe I'm missing something.

Poking around, there seems to at least be some question about that. Doug Lea's "JSR-133 Cookbook for Compiler Writers" has this: "Alternatively, if available, you can implement volatile store as an atomic instruction (for example XCHG on x86) and omit the barrier. This may be more efficient if atomic instructions are cheaper than StoreLoad barriers."

Reply

George Spofford 29 June 2013 at 00:35

Now I'm inspired to translate from the descriptions in
http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf
to assembly to see if there are clues.

Reply

Unknown 2 July 2013 at 16:56

I mention the use of XCHG in https://blogs.oracle.com/dave/resource/NHM-Pipeline-Blog-V2.txt. Benchmarking showed no appreciable difference so we stuck with LOCK:ADD to the top-of-stack as that kills the ICCs but doesn't kill a register as does XCHG. (Note that XCHG to memory is, for historical reasons, always LOCK-ed) . As Dmitry mentioned, and is noted in comments in the hotspot source, the line underlying top-of-stack is almost always in M-state.

In x86_64-bit mode we dedicate a register to hold the thread pointer. I've done some experiments that suggest the following idiom _might be slightly better than lock:add of 0 to top-of-stack : xchg rThread, rThread->Self where the Self field simply refers to the thread structure itself. This doesn't kill any registers.

Reply

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)