Java CPU Sampling Using hprof

09 Jun 2014

hprof is a free profiler shipped with Java, for heap, CPU, and monitor profiling. It is a powerful and well presented tool, however, its CPU sampling mode doesn't work properly, as it can produce misleading or inaccurate results. I'm also not sure its CPU sampling mode has *ever worked*.

In this post I'll describe some of the problems, which should be helpful for anyone trying to use hprof CPU sampling. It's tempting to discuss other profilers in detail as well, but to keep this short I'll stick to hprof.

CPU Sampling Example

Here's hprof CPU sampling of vert.x, an application platform for the JVM. Excerpts from the hprof report:

```
TRACE 301366: (thread=200009)
        sun.nio.ch.EPollArrayWrapper.epollWait(EPollArrayWrapper.java:Unknown line)
        sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:269)
        sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:79)
        sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
        sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
        io.netty.channel.nio.NioEventLoop.select(NioEventLoop.java:605)
        io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:306)
        io.netty.util.concurrent.SingleThreadEventExecutor$2.run(SingleThreadEventExecutor.java:1
        java.lang.Thread.run(Thread.java:744)
[...]
CPU SAMPLES BEGIN (total = 3310) Mon Jun  9 20:33:05 2014
rank   self  accum   count trace method
   1 48.70% 48.70%    1612 301366 sun.nio.ch.EPollArrayWrapper.epollWait
   2 32.08% 80.79%    1062 301340 sun.nio.ch.EPollArrayWrapper.epollWait
   3  3.05% 83.84%     101 301526 sun.nio.ch.NativeThread.current
   4  2.48% 86.31%      82 301457 sun.nio.ch.FileDispatcherImpl.write0
   5  0.88% 87.19%      29 301460 sun.nio.ch.FileDispatcherImpl.read0
   6  0.57% 87.76%      19 301484 sun.misc.Unsafe.copyMemory
   7  0.42% 88.19%      14 301546 io.netty.buffer.AbstractByteBuf.setIndex
[...]
```

The output begins with many stack traces, enumerated, and then a sorted summary showing which stacks were most often sampled. Multiply the "count" column with the sampling interval (in this case, I used 20 ms), to get a rough measure of total time spent in these methods.

Stack trace 301366, in method epollWait, was sampled 48.7% of the time. The second highest stack trace is in the same method, bringing the total to 80.79%. Why is epollWait so hot on-CPU? We should fix it – this app could run 5x faster!

There have been cases of epollWait consuming 100% CPU, due to a bug, in which case hprof would be reporting the truth. But that's not the case here – these threads aren't running on CPU at all.

These hprof results are actually bogus.

Using hprof cpu=samples

Here is how hprof can be invoked to perform "CPU" sampling:

```
java -agentlib:hprof=cpu=samples,depth=100,interval=20,lineno=y,thread=y,file=out.hprof myclass
```

This samples stacks up to 100 frames deep, every 20 ms, and writes a report to out.hprof when the program exits, or when java receives a SIGQUIT.

This was previously available in the JVM as the -Xrunhprof option, which still works today, but may be removed in a future version of the JDK (it's still there as of 1.8.0_05). You can check what options are currently available by using hprof=help. Selected lines below:

```
$ java -agentlib:hprof=help

     HPROF: Heap and CPU Profiling Agent (JVMTI Demonstration Code)

hprof usage: java -agentlib:hprof=[help]|[<option>=<value>, ...]

Option Name and Value  Description                    Default
---------------------  -----------                    -------
heap=dump|sites|all    heap profiling                 all
cpu=samples|times|old  CPU usage                      off
monitor=y|n            monitor contention             n
format=a|b             text(txt) or binary output     a
file=<file>            write data to file             java.hprof[{.txt}]
net=<host>:<port>      send data over a socket        off
depth=<size>           stack trace depth              4
interval=<ms>          sample interval in ms          10
cutoff=<value>         output cutoff point            0.0001
lineno=y|n             line number in traces?         y
thread=y|n             thread in traces?              n
[...]
Examples
--------
  - Get sample cpu information every 20 millisec, with a stack depth of 3:
      java -agentlib:hprof=cpu=samples,interval=20,depth=3 classname
[...]
Warnings
--------
  - This is demonstration code for the JVMTI interface and use of BCI,
    it is not an official product or formal part of the JDK.
[...]
```

Note that the output mentions twice that this is demonstration code for JVMTI. That may help explain why the issues have gone unnoticed and unfixed. Neither the output nor the hprof documentation warns that the results can also be inaccurate. Maybe the developers never discovered this, since for some simple programs it appears to work fine. (And if they did discover it, well, it is only a "demo" and not a product.)

Idle Test

To better understand the problem, I profiled vert.x when it was completely idle, and checked CPU usage:

```
$ top -p 10515
[...]
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
30515 root      20   0 4273m 108m  12m S    0  1.5   0:14.33 java

$ mpstat 1
[...]
08:50:53 PM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
08:50:54 PM  all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
08:50:54 PM    0    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
08:50:54 PM    1    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
```

vert.x is completely idle, yet hprof still reports "CPU SAMPLES" in epollWait, at a rate of around 2000ms every second – indicating two CPU-bound threads running in parallel.

jstack(1) can help explain what is happening. Showing the thread that is supposed to be hot on-CPU:

```
"vert.x-eventloop-thread-0" #10 daemon prio=5 os_prio=0 tid=0x00007f5398256000 nid=0x774a runnabl
   java.lang.Thread.State: RUNNABLE
    at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
    at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:269)
    at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:79)
    at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:86)
    - locked <0x000000008b97d450> (a io.netty.channel.nio.SelectedSelectionKeySet)
    - locked <0x000000008b97e4c0> (a java.util.Collections$UnmodifiableSet)
    - locked <0x000000008b97d3b8> (a sun.nio.ch.EPollSelectorImpl)
    at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:97)
    at io.netty.channel.nio.NioEventLoop.select(NioEventLoop.java:605)
    at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:306)
    at io.netty.util.concurrent.SingleThreadEventExecutor$2.run(SingleThreadEventExecutor.java:11
    at java.lang.Thread.run(Thread.java:745)
```

epollWait is RUNNABLE according to the JVM – not the kernel scheduler (it's really asleep!).

Internals

hprof is sampling the Java RUNNABLE state. From jvmti/hprof/hprof_trace.c:

```
/* Get traces for all threads in list (traces[i]==0 if thread not running) */
void
trace_get_all_current(jint thread_count, jthread *threads,
[...]
        /* If thread has frames, is runnable, and isn't suspended, we care */
        if ( always_care ||
             ( stack_info[i].frame_count > 0
               && (stack_info[i].state & JVMTI_THREAD_STATE_RUNNABLE)!=0
               && (stack_info[i].state & JVMTI_THREAD_STATE_SUSPENDED)==0
               && (stack_info[i].state & JVMTI_THREAD_STATE_INTERRUPTED)==0 )
           ) {
[...do sampling...]
```

JVMTI_THREAD_STATE_RUNNABLE maps to the Java RUNNABLE Thread.State, which is documented as:

> A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as processor.

Runnable does not mean *running*. It is also not the same as what the kernel scheduler calls "runnable". Sampling Java runnable threads is not the same thing as CPU sampling, but hprof documents it as "CPU SAMPLES". It would be better described as "JVM RUNNABLE SAMPLES".

This fact doesn't seem to be well known. The best reference I found said (from 2005):

> When you have Java threads that are somehow not using the CPU, but managing to stay active, then it will appear as if those stack traces are consuming large amounts of CPU time when they aren't.

This was written by Kelly O'Hair, who wrote hprof. He references the source, and speculates on how to fix this based on examining more Java thread states. I think it needs to go further than that...

Inaccurate Timing

There's another issue with hprof worth mentioning. The paper "Evaluating the Accuracy of Java Profilers" [Mytkowicz 10] shows that different Java profilers, including hprof, produce conflicting profiles. The reason the authors identified was that hprof only samples on *yield points*, which may not align with the interval specified. They included a sample program which hprof profiles incorrectly, to prove their point.

I find this issue really hard to believe, but I profiled their sample program and saw results that also didn't make sense. I'd need to study some more to confirm that this really is yield, and not another profiling issue that the authors have stumbled upon.

UPDATE: The JVM term for these are *safepoints* (see comments).

Why hprof Isn't Used

If hprof was in common usage for CPU sampling, these problems should have been fixed long ago. But it isn't. I've heard various reasons as to why, including:

1. Other (commercial) profilers have more features. Eg: YourKit, JProfiler, JRocket Flight Recorder, etc.
2. IDE profilers (incl. NetBeans Profiler, etc.) are easier for developers than the CLI hprof.
3. hprof can't be turned on and off when needed, so the output contains everything since startup.
4. The hprof overhead for CPU profiling is too high.

(3) is true, but there seem to be fixes for that. hprof uses the JVMTI interface, and while hprof doesn't currently provide on/off capabilities, the JVMTI interface does have them: the NetBeans profiler uses JVMTI for on-demand profiling, as does the VisualVM profiler (which uses the NetBeans profiler).

A simple workaround (ignoring overheads for a moment) could be fashioned using SIGQUIT. With hprof running, a SIGQUIT signal causes hprof to write out its report, and reset profiler data. And so, a second SIGQUIT will only show profile data since the first. This isn't on-demand, since the profiler is always running, but it does let us collect data for an interval of interest.

(4) is definitely true for the "cpu=times" mode, which instruments every method call, and can slow the target application by 1000x (I saw 480x, last time I tried). As an aside, I've been *stunned* to see products describe this as providing "highly accurate" times, while at the *same time* warning that it may massively slow the target. Heard of **observer effect**?

The "cpu=samples" mode has much less overhead, and the interval is tunable. For an app I'm testing (vert.x), profiling at 20 ms reduced its request rate by 2%, and increased JVM CPU consumption by 4%. I imagine if this server was out of CPU headroom, the total reduction in request rate would be around 6%. If I had a lot more threads, I may need to increase the sample rate to 50ms or higher to keep the overhead this low.

The most compelling reason not to use hprof for CPU sampling is that the output can be inaccurate. I asked a coworker, a Java performance expert, whether he used hprof, and his answer was to the point: "No, it includes blocking time so the results are misleading".

Fixing hprof

The runnable state issue and yield-based sampling could both be fixed at the same time, by profiling *from* the system. The best example was developed by Sun: the jstack() action for DTrace, which can sample Java stack traces based on the kernel's understanding of running CPUs, and using reliable system timers. This approach can sample not just Java methods, but also JVM internals and the kernel, providing the most complete picture possible of CPU consumption. Since it was launched around the same time as hprof, it may further explain why hprof wasn't fixed earlier: Sun's attention was on DTrace, not hprof. (I'd use DTrace myself, although I've hit an issue with incomplete jstacks.)

Aside from system profiling, the runnable issue may be fixable by reading kernel state (eg, via /proc on Linux), to associate actual CPU state with Java thread state. I haven't looked inside commercial profilers, but this may be how they work. One would need to keep the overheads for reading OS state in check.

Assuming the yield issue hasn't distorted our results too much, a dumb fix may be to create an exclude list of methods that are known to block yet stay in the Java RUNNABLE state, and to filter these from the hprof results. I saw an implementation of this by jnorris:

```
    ignore_set = set([
        'java.net.SocketInputStream.socketRead0',
        'java.net.SocketOutputStream.socketWrite0',
        'java.net.PlainSocketImpl.socketAvailable',
        'java.net.PlainSocketImpl.socketAccept',
        'sun.nio.ch.EPollArrayWrapper.epollWait',
    ])
```

I feel dirty just looking at it. While this is an enormous kludge, it may also be a pragmatic workaround.

Of course, I could (and should) also ask Oracle to fix hprof in the JVM. Although I'm not sure how much of a priority that would be, given that Java Flight Recorder is also a revenue stream.

Conclusion

Don't trust any performance observability tools you use, without double (or triple) checking their results. Including tools that are bundled and seem standard.

In this case, the bundled Java hprof profiler may not sample CPU usage correctly, depending on your program. It samples based on Java's notion of a thread being runnable, rather than the kernel's notion of a thread *actually running on a CPU*. These can differ, such as in epollWait, leading to profiles that blame the wrong methods for CPU consumption. Seasoned hprof users may have learned to look past such blocking methods, knowing that the profiler is sampling them incorrectly.

Another reported hprof issue involves CPU sampling based on yield points, rather than intervals, which can also severely skew results.

If hprof, and its JVMTI infrastructure, could be fixed, we'd have a free bundled profiler for Java. However, I'm not sure if anyone wants to do that work. The status quo for Java is to pay for commercial profilers. I'm tempted to take this on myself, because I don't believe I should need to pay for *basic* CPU profiling.

Update

I didn't want to get into other profilers too much, but I just found this, and it's worth mentioning since it was written to address the same issues, and is also free and opensource: Google's lightweight java profiler. It is described on slides 26-30 of a 2013 talk by Jeremy Manson, which has the awesome slide "Why Profiling Doesn't work", with the reasons:

- Profiling happens at safe points
- Doesn't say what's actually running
- Doesn't account for GC time

The first two are what I discussed. Not accounting GC time would be a third problem.

This led Jeremy to create a profiler that operated asynchronously (like one would expect), using the Hotspot JVM's AsyncGetCallTrace interface. You can read about it more in his blog post. I just tried it on a few programs, and so far it appears to be working correctly.

As mentioned in the comments, Richard Warburton's honest-profiler also uses the AsyncGetCallTrace interface, building upon Jeremy Manson's approach. This version is on github, and described recently in his An open source JVM Sampling Profiler blog post. Like the google profiler and hprof, these may not be production ready as they are more demos than products. Some assembly may be needed.

Resources

- hprof Documentation on docs.oracle.com.
- Java thread states on docs.oracle.com.
- Mytkowicz 10: T Mytkowicz, A Diwan, M Hauswirth, PF Sweeney. "Evaluating the Accuracy of Java Profilers," *ACM Sigplan Notices* 45 (6), 187-197
- Kelly O'Hair on the hprof runnable issue.
- Jeremy Manson on his Lightweight Asynchronous Sampling Profiler, and Why Many Profilers have Serious Problems.
- Richard Warburton's honest-profiler, which may be the most up to date, working, free, and open source Java CPU profiler.

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).