Fast and Safe Production Monitoring of JVM Applications with BPF Magic

Sasha Goldshtein
CTO, Sela Group

goldshtn
goldshtn
The Plan

• This is a talk on hardcore Linux tracing tools and how they can be used with JVM applications
  • For non-Linux platforms, rough equivalents are sort of available (e.g. dtrace on macOS and FreeBSD)

• You’ll learn:
  ☐ Which production-ready tracing tools can be used with JVM apps
  ☐ How BPF changes the picture of Linux tracing
  ☐ To apply a performance checklist for JVM apps using BPF tools
  ☐ To conduct ad-hoc investigations with one-liners and custom tools
Landscape of Linux Tracing Tools

Ease of use

SysDig

ply/BPF
dtrace for Linux

ktap

LTTng

perf

SystemTap

ftrace

bcc/BPF

C/BPF

custom .ko

new

stable

dead

Level of detail, features
Demo: Observability Points in the JVM

- Objective:
  Understand which tracepoints are available in the JVM
<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem__pool__gc__begin</td>
<td>0x00000078</td>
<td>NT_STAPSDT (SystemTap probe descriptors)</td>
</tr>
<tr>
<td>class__loaded</td>
<td>0x0000004d</td>
<td>NT_STAPSDT (SystemTap probe descriptors)</td>
</tr>
<tr>
<td>object__alloc</td>
<td>0x0000004e</td>
<td>NT_STAPSDT (SystemTap probe descriptors)</td>
</tr>
<tr>
<td>thread__start</td>
<td>0x00000065</td>
<td>NT_STAPSDT (SystemTap probe descriptors)</td>
</tr>
</tbody>
</table>

readelf displays the raw SDT notes from the binary, a quick way to identify which probes are available.
tplist (from BCC) can also display a list of probes from a binary or a running process.
```bash
$ find /usr/lib/jvm -name libjvm.so -exec tplist -vv -l {} + | grep monitor__waited -A10
/usr/lib/jvm/.../server/libjvm.so hotspot:monitor__waited [sema 0x0]
  location #1 0xa0c4dd
    argument #1 8 signed  bytes @ ax
    argument #2 8 unsigned bytes @ r12
    argument #3 8 unsigned bytes @ dx
    argument #4 4 signed  bytes @ cx
  location #2 0xa0e85d
    argument #1 8 signed  bytes @ ax
    argument #2 8 unsigned bytes @ r15
    argument #3 8 unsigned bytes @ dx
    argument #4 4 signed  bytes @ cx

$ find /usr/lib/jvm -name hotspot-*.*.stp -exec grep 'mark("monitor__waited")' -A10 {} +
  process("/usr/lib/jvm/.../server/libjvm.so").mark("monitor__waited")
{
  name = "monitor_waited";
  thread_id = $arg1;
  id = $arg2;
  class = user_string_n($arg3, $arg4);
  probestr = sprintf("%s(thread_id=%d, id=0x%x, class='%s')",
                      name, thread_id, id, class);
}
```
Demo: Methods and Stack Traces

• Objective:
  A Java app is printing undesired stuff to the console, and you want to understand where it’s coming from

• JVM flags we will need:
  -XX:+PreserveFramePointer ~3% overhead, helps get good stacks
$ java ... myapp
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
trace (from BCC) attaches a dynamic trace to an arbitrary location; in this case, the write syscall

<table>
<thead>
<tr>
<th>PID</th>
<th>TID</th>
<th>COMM</th>
<th>FUNC</th>
</tr>
</thead>
<tbody>
<tr>
<td>27982</td>
<td>27983</td>
<td>java</td>
<td>Sys_write</td>
</tr>
</tbody>
</table>

Error fetching data, cleaning up.

write+0x2d [libpthread-2.24.so]
writeBytes+0x1f0 [libjava.so]
Java_java_io_FileOutputStream_writeBytes+0x1a [libjava.so]
[unknown] [perf-15527.map]
[unknown] [perf-15527.map]
[unknown] [perf-15527.map]
[unknown] [perf-15527.map]
[unknown] [perf-15527.map]
[unknown] [perf-15527.map]
[unknown] [perf-15527.map]
[unknown] [perf-15527.map]
[unknown] [perf-15527.map]

JavaCalls::call_helper(JavaValue*, methodHandle*, ...) +0xf53 [libjvm.so]
jni_invoke_static(JNIEnv_*, JavaValue*, jobject*, ...) +0x357 [libjvm.so]
jni_CallStaticVoidMethod +0x186 [libjvm.so]
JavaMain +0x6d1 [libjli.so]
start_thread +0xca [libpthread-2.24.so]
$ create-java-perf-map.sh `pidof java` "unfoldall,dottedclass"
$ tail /tmp/perf-`pidof java`.map
7f6ed52ea5c0 100 DataFetcher::processIt
7f6ed52eaa20 880 java.lang.ClassLoader::loadClass
7f6ed52ebf40 340 DataFetcher::fetchData
7f6ed52ec660 220 RequestProcessor::processRequest
7f6ed52ecbe0 220 RequestProcessor::processRequest
7f6ed52ed140 120 DataFetcher::<init>
7f6ed52ed500 180 sun.misc.URLClassPath$FileLoader$1::getInputStream
7f6ed52ed9c0 520 java.lang.ClassLoader::loadClass
7f6ed52eff20 100 DataFetcher::fetchData
7f6ed52f1e80 4c0 java.lang.ThreadGroup::add

create-java-perf-map.sh (from perf-map-agent) generates a symbol file describing Java symbols and addresses
# trace 'Sys_write (arg1==1) "%s", arg2' -U -p `pidof java`

<table>
<thead>
<tr>
<th>PID</th>
<th>TID</th>
<th>COMM</th>
<th>FUNC</th>
<th>FUNC</th>
</tr>
</thead>
<tbody>
<tr>
<td>25335</td>
<td>25336</td>
<td>java</td>
<td>SyS_write</td>
<td>Error fetching data, cleaning up.f8</td>
</tr>
</tbody>
</table>

```plaintext
write+0x24  [libpthread-2.24.so]
writeBytes+0x1f0  [libjava.so]
Java_java_io_FileOutputStream_writeBytes+0x1a  [libjava.so]
java.io.FileOutputStream::writeBytes+0xc6  [perf-15527.map]
java.io.FileOutputStream::write+0x74  [perf-15527.map]
java.io.BufferedOutputStream::flushBuffer+0xa5  [perf-15527.map]
java.io.BufferedOutputStream::flush+0x98  [perf-15527.map]
java.io.PrintStream::write+0xf8  [perf-15527.map]
sun.nio.cs.StreamEncoder::writeBytes+0x13c  [perf-15527.map]
sun.nio.cs.StreamEncoder::implFlushBuffer+0xcc  [perf-15527.map]
sun.nio.cs.StreamEncoder::flushBuffer+0xb8  [perf-15527.map]
java.io.OutputStreamWriter::flushBuffer+0x85  [perf-15527.map]
java.io.PrintStream::newLine+0xf4  [perf-15527.map]
java.io.PrintStream::println+0xb0  [perf-15527.map]
DataFetcher::fetchData+0xd4  [perf-15527.map]
RequestProcessor::processRequest+0xc0  [perf-15527.map]
Collecty::main+0x68  [perf-15527.map]
call_stub+0x88  [perf-15527.map]
JavaCalls::call_helper(JavaValue*, methodHandle*, ...)+0xf53  [libjvm.so]
```

...
Demo: Methods and Stack Traces

• Objective:
  A Java app is causing a lot of garbage collections by invoking System.gc() directly, and you want to understand why

• JVM flags we will need:
  -XX:+PreserveFramePointer ~3% overhead, helps get good stacks
  -XX:+ExtendedDTraceProbes very expensive, only for debugging method calls/object allocations
$ java ... -XX:+PrintGC myapp
[Full GC (System.gc()) 530K->255K(15872K), 0.0021490 secs]
[Full GC (System.gc()) 255K->255K(15936K), 0.0020310 secs]
[Full GC (System.gc()) 255K->255K(15936K), 0.0017840 secs]
[Full GC (System.gc()) 255K->253K(15936K), 0.0019176 secs]
[Full GC (System.gc()) 254K->253K(15936K), 0.0018467 secs]
[Full GC (System.gc()) 254K->253K(15936K), 0.0018358 secs]
```
<table>
<thead>
<tr>
<th>PID</th>
<th>TID</th>
<th>COMM</th>
<th>FUNC</th>
<th>condition</th>
<th>trace message</th>
</tr>
</thead>
<tbody>
<tr>
<td>25413</td>
<td>25414</td>
<td>java</td>
<td>method__entry</td>
<td>(STRCMP(&quot;gc&quot;, arg4))</td>
<td>&quot;induced GC&quot;</td>
</tr>
</tbody>
</table>

```

```bash
U -p `pidof java`
```

SharedRuntime::dtrace_method_entry(...)+0x7b [libjvm.so]
java.lang.Runtime::gc+0x80 [perf-15605.map]
java.lang.System::gc+0x40 [perf-15605.map]
DataFetcher::fetchData+0xdc [perf-15605.map]
RequestProcessor::processRequest+0xc0 [perf-15605.map]
Collecty::main+0x14b [perf-15605.map]
call_stub+0x88 [perf-15605.map]
JavaCalls::call_helper(JavaValue*, methodHandle*, ...)+0xf53 [libjvm.so]
jni_invoke_static(JNIEnv_*, JavaValue*, jobject*, ...)+0x357 [libjvm.so]
jni_CallStaticVoidMethod+0x186 [libjvm.so]
JavaMain+0x6d1 [libjli.so]
start_thread+0x6d1 [libpthread-2.24.so]
```

Here, trace attaches to a relatively expensive method-entry probe that requires `-XX:+ExtendedDTraceProbes`
OK, So You Have Probably Met **perf**

- There is at least one talk at every Java conference about using `perf` for CPU profiling and flame graphs
  - JPoint 2017: Пангин и Цесько, JVM-профайлер с чувством такта
- For example:
  
  ```
  # perf record -g -F 97 -- java ...
  # perf script | ./stackcollapse-perf.pl | ./flamegraph.pl --color java > java.svg
  ```
What’s Wrong With `perf`?

- `perf` relies on pushing a *lot of data* to user space, through *files*, for *analysis*
  - Downloading a file at ~1Gb/s produces ~89K `netif_receive_skb` events/s (19MB/s including stacks)

```
perf | awk | ...
```

average packet size: 189 bytes
BPF: 1990

- **Invented** by McCanne and Jacobson at Berkeley, 1990-1992: instruction set, representation, implementation of packet filters

```
$ tcpdump -d 'ip and dst 186.173.190.239'
(000) ldh [12]
(001) jeq #0x800    jt 2    jf 5
(002) ld [30]
(003) jeq #0xbaadbeef    jt 4    jf 5
(004) ret #262144
(005) ret #0
```
BPF: Today

• Supports a wide spectrum of usages
• Has a JIT for maximum efficiency
BPF Scenarios

**XDP**
- kernel
  - TCP
  - IP
  - BPF program
  - e10000
  - rx
  - fwd
  - drop
to app

**seccomp**
- user
  - application
  - syscall
  - BPF program
  - syscall interface
  - allow
  - errno
  - kill
BPF Tracing

① installs BPF program and attaches to events
② events invoke the BPF program
③ BPF program updates a map or pushes a new event to a buffer shared with user-space
④ user-space program is invoked with data from the shared buffer
⑤ user-space program reads statistics from the map and clears it if necessary
## BPF Tracing Features in The Linux Kernel

<table>
<thead>
<tr>
<th>Version</th>
<th>Feature</th>
<th>Scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>kprobes/uprobes attach</td>
<td>Dynamic tracing with BPF becomes possible</td>
</tr>
<tr>
<td>4.1</td>
<td>bpf_trace_printk</td>
<td>BPF programs can print output to ftrace pipe</td>
</tr>
<tr>
<td>4.3</td>
<td>perf_events output</td>
<td>Efficient tracing of large amounts of data for analysis in user-space</td>
</tr>
<tr>
<td>4.6</td>
<td>Stack traces</td>
<td>Efficient aggregation of call stacks for profiling or tracing</td>
</tr>
<tr>
<td>4.7</td>
<td>Tracepoints support</td>
<td>API stability for tracing programs</td>
</tr>
<tr>
<td>4.9</td>
<td>perf_events attach</td>
<td>Low-overhead profiling and PMU sampling</td>
</tr>
</tbody>
</table>
The Old Way And The New Way

The diagram illustrates the comparison between the old and new ways of monitoring kernel operations, specifically focusing on the VFS (Virtual File System) probe mechanism. The old way involves using `perf` and `awk` to analyze performance data, while the new way utilizes BPF (Binary Instrumentation Framework) tools for more efficient monitoring.

### The Old Way

- **Kernel**: `vfs_read` probe
- **User**: `perf | awk` to analyze the `perf_events`
- **Monitor**: LATμs distribution

### The New Way

- **Kernel**: `vfs_read` probe
- **User**: BPF program to analyze `perf.data`
- **Monitor**: LATμs distribution

Both methods show a distribution of latency times, with the new method indicating a more efficient monitoring approach, as evidenced by the chart showing a smaller and more efficient distribution compared to the old method.
The BCC BPF Front-End

- [https://github.com/iovisor/bcc](https://github.com/iovisor/bcc)

- BPF Compiler Collection (BCC) is a BPF frontend library and a massive collection of performance tools
  - Contributors from Facebook, PLUMgrid, Netflix, Sela

- Helps build BPF-based tools in high-level languages
  - Python, Lua, C++
BCC JVM on Linux Performance Checklist

1. ustat
2. ugc
3. execsnoop
4. opensnoop
5. ext4slower
   (or btrfs*, xfs*, zfs*)
6. biolatency
7. biosnoop
8. cachestat
9. tcpconnect
10. tcpaccept
11. tcptop
12. gethostlatency
13. uthreads
14. cpudist
15. runqlat
16. profile
Demo: CPU Investigation

• Objective:
  Identify on-CPU hot methods in a running application
$ top
PID USER   PR  NI  VIRT  RES  SHR  S %CPU %MEM    TIME+  COMMAND
25491 vagrant  20   0  2260396 29372 15876 S   99.3  2.9   0:15.93 java

# uthreads -l java `pidof java`
Tracing thread events in process 25582 (language: java)... Ctrl-C to quit.

TIME   ID      TYPE       DESCRIPTION
1.061  25594   pthread    [unknown]
1.062  R=8/N=0  start      Thread-0
1.068  25595   pthread    [unknown]
1.069  R=9/N=0  start      Thread-1

# profile -U -p `pidof java` -F 97 5

Primes::isPrime
Primes::primesThread
Primes::access$000
Primes$1::run
java.lang.Thread::run
call_stub

start_thread
- java (25582)
  43
profile -f outputs folded stacks in a format suitable for flame graph generation
$ grep -A3 -B1 synchronized Computey.java
    if (isPrime(i)) {
        synchronized (primesLock) {
            primes.add(i);
        }
    }

# argdist -p `pidof java` -i 5 -C 'u:libjvm.so:monitor__contended__enter()'
[11:22:06]
u:libjvm.so:monitor__contended__enter()
    COUNT     EVENT
u:libjvm.so:monitor__contended__enter()
    COUNT     EVENT
    3     total calls
u:libjvm.so:monitor__contended__enter()
    COUNT     EVENT
u:libjvm.so:monitor__contended__enter()
    COUNT     EVENT
u:libjvm.so:monitor__contended__enter()
    COUNT     EVENT
    9     total calls

argdist (from BCC) generates frequency counts or histograms of interesting events
Demo: Slow MySQL Queries

• Objective:
  Determine why a Java + MySQL application occasionally produces slow results, and where the slow queries are coming from
<table>
<thead>
<tr>
<th>query latency (ms)</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 1</td>
<td>3027</td>
</tr>
<tr>
<td>2 - 3</td>
<td>5</td>
</tr>
<tr>
<td>4 - 7</td>
<td>1</td>
</tr>
<tr>
<td>8 - 15</td>
<td>0</td>
</tr>
<tr>
<td>16 - 31</td>
<td>0</td>
</tr>
<tr>
<td>32 - 63</td>
<td>0</td>
</tr>
<tr>
<td>64 - 127</td>
<td>0</td>
</tr>
<tr>
<td>128 - 255</td>
<td>0</td>
</tr>
<tr>
<td>256 - 511</td>
<td>0</td>
</tr>
<tr>
<td>512 - 1023</td>
<td>0</td>
</tr>
<tr>
<td>1024 - 2047</td>
<td>8</td>
</tr>
</tbody>
</table>

# dbstat mysql
Tracing database queries slower than 5ms for PID 25776... Ctrl+C to quit.

<table>
<thead>
<tr>
<th>query latency (ms)</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 - 31</td>
<td>0</td>
</tr>
<tr>
<td>32 - 63</td>
<td>0</td>
</tr>
<tr>
<td>64 - 127</td>
<td>0</td>
</tr>
<tr>
<td>128 - 255</td>
<td>0</td>
</tr>
<tr>
<td>256 - 511</td>
<td>0</td>
</tr>
<tr>
<td>512 - 1023</td>
<td>0</td>
</tr>
<tr>
<td>1024 - 2047</td>
<td>6</td>
</tr>
</tbody>
</table>
$ top

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+ COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>25776</td>
<td>mysql</td>
<td>20</td>
<td>0</td>
<td>1201036</td>
<td>101120</td>
<td>15144</td>
<td>S</td>
<td>2.0</td>
<td>10.0</td>
<td>0:03.80</td>
</tr>
<tr>
<td>26036</td>
<td>vagrant</td>
<td>20</td>
<td>0</td>
<td>2261580</td>
<td>59580</td>
<td>16344</td>
<td>S</td>
<td>1.3</td>
<td>5.9</td>
<td>0:07.80</td>
</tr>
</tbody>
</table>

# dbslower mysql -m 500
Tracing database queries for PID 25776 slower than 500 ms...

<table>
<thead>
<tr>
<th>TIME(s)</th>
<th>PID</th>
<th>MS</th>
<th>QUERY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000000</td>
<td>25776</td>
<td>2001</td>
<td>call getproduct(97)</td>
</tr>
<tr>
<td>2.123951</td>
<td>25776</td>
<td>2002</td>
<td>call getproduct(97)</td>
</tr>
<tr>
<td>4.259418</td>
<td>25776</td>
<td>2002</td>
<td>call getproduct(97)</td>
</tr>
<tr>
<td>6.387346</td>
<td>25776</td>
<td>2002</td>
<td>call getproduct(97)</td>
</tr>
</tbody>
</table>

dbslower (from BCC) displays MySQL/PostgreSQL queries slower than the specified threshold
# trace -p `pidof mysqld` 'u:/usr/local/mysql/bin/mysqld:query__exec__start "query=%s", arg1' 

<table>
<thead>
<tr>
<th>PID</th>
<th>TID</th>
<th>COMM</th>
<th>FUNC</th>
<th>Query Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>25776</td>
<td>26047</td>
<td>mysqld</td>
<td>query__exec__start</td>
<td>call getproduct(95)</td>
</tr>
<tr>
<td>25776</td>
<td>26047</td>
<td>mysqld</td>
<td>query__exec__start</td>
<td>select * from products where id = 95</td>
</tr>
<tr>
<td>25776</td>
<td>26047</td>
<td>mysqld</td>
<td>query__exec__start</td>
<td>select * from users where id = 48</td>
</tr>
<tr>
<td>25776</td>
<td>26047</td>
<td>mysqld</td>
<td>query__exec__start</td>
<td>select id from products where userid = 48</td>
</tr>
<tr>
<td>25776</td>
<td>26047</td>
<td>mysqld</td>
<td>query__exec__start</td>
<td>call getproduct(96)</td>
</tr>
<tr>
<td>25776</td>
<td>26047</td>
<td>mysqld</td>
<td>query__exec__start</td>
<td>select * from products where id = 96</td>
</tr>
<tr>
<td>25776</td>
<td>26047</td>
<td>mysqld</td>
<td>query__exec__start</td>
<td>call getproduct(97)</td>
</tr>
<tr>
<td>25776</td>
<td>26047</td>
<td>mysqld</td>
<td>query__exec__start</td>
<td>do sleep(2)</td>
</tr>
</tbody>
</table>

...
# ./mysqlsniff.py -p `pidof java` -f "call getproduct(97)" -S
Sniffing process 26036, Ctrl+C to quit.
call getproduct(97)

__libc_send+0x0
Java_java_net_SocketOutputStream_socketWrite0+0x102
java.net.SocketOutputStream::socketWrite0+0xda
java.net.SocketOutputStream::socketWrite+0x84
java.net.SocketOutputStream::write+0x34
java.io.BufferedOutputStream::flushBuffer+0x5c
java.io.BufferedOutputStream::flush+0x78
com.mysql.jdbc.MysqlIO::send+0x2d0
com.mysql.jdbc.MysqlIO::sendCommand+0x188
com.mysql.jdbc.MysqlIO::sqlQueryDirect+0x8f8
com.mysql.jdbc.ConnectionImpl::execSQL+0x324
com.mysql.jdbc.ConnectionImpl::execSQL+0x74
com.mysql.jdbc.StatementImpl::executeQuery+0x4ec
Product::load+0x288
User::loadProducts+0x33c
Database::main+0x16b
call_stub+0x88
...

start_thread+0xca

mysqlsniff (demo tool) analyzes client network traffic to identify MySQL queries and display call stack.
Demo Summary: Slow MySQL Queries

• Objective:
  Determine why a Java + MySQL application occasionally produces slow results, and where the slow queries are coming from
Demo: Lots of GC

• Objective:
  Understand the load generated by a Java application, identify that it has to do with GC, and figure out where the garbage is coming from
```bash
$ top
PID USER   PR NI VIRT  RES   SHR S %CPU %MEM    TIME+ COMMAND
26176 vagrant 20   0 2256284 45276 15884 S 99.9  4.5 0:07.10 java

# ustat -C
15:02:09 loadavg: 0.24 0.05 0.03 3/202 28698
PID CMDLINE METHOD/s  GC/s OBJNEW/s CLOAD/s EXC/s THR/s
28689 java -XX:+PreserveF 0 888 0 0 0 0

15:02:10 loadavg: 0.30 0.07 0.04 4/202 28698
PID CMDLINE METHOD/s  GC/s OBJNEW/s CLOAD/s EXC/s THR/s
28689 java -XX:+PreserveF 0 898 0 0 0 0

# profile -p `pidof java` 5
...

__memset_erms
[unknown]
_new_array_Java
ResponseBuilder::addLine
Allocy::main
call_stub
...

start_thread
- java (26176)

31
```

ustat (from BCC) is a top-like extension for Java events (and other languages)
# uobjnew java `pidof java` 5
Tracing allocations in process 26259 (language: java)... Ctrl-C to quit.

<table>
<thead>
<tr>
<th>TYPE</th>
<th># ALLOCS</th>
<th># BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>java/lang/String</td>
<td>12588</td>
<td>0</td>
</tr>
<tr>
<td>[C]</td>
<td>12588</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TYPE</th>
<th># ALLOCS</th>
<th># BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>java/lang/String</td>
<td>11680</td>
<td>0</td>
</tr>
<tr>
<td>[C]</td>
<td>11680</td>
<td>0</td>
</tr>
</tbody>
</table>

# stackcount -i 5 -p `pidof java` "u:.../libjvm.so:object__alloc"

... SharedRuntime::dtrace_object_alloc(oopDesc*, int)
TypeArrayKlass::allocate_common(int, bool, Thread*)
OptoRuntime::new_array_C(Klass*, int, JavaThread*)
_new_array_Java
ResponseBuilder::addLine
Allocy::main
call_stub
...

JavaMain
start_thread 870

---

uobjnew (from BCC) traces object allocations by using an expensive probe that requires -XX:+ExtendedDTraceProbes

---

stackcount (from BCC) attaches to that probe and summarizes Java call stacks leading up to it (could also do a flame graph)
Demo Summary : Lots of GC

• Objective:
  Understand the load generated by a Java application, identify that it has to do with GC, and figure out where the garbage is coming from

• Commands used:
  $ top
  # ustat
  # profile -U -p `pidof java` 5
  # uobjnew java `pidof java` 5
  # stackcount -i 5 -p `pidof java`
  "u:../libjvm.so:object__alloc"
Demo: Failed Initialization

• Objective:
  Figure out why an application fails to initialize and keeps printing weird messages
$ java ... Servery
[*] Server started, initializing.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
# Opensnoop

<table>
<thead>
<tr>
<th>PID</th>
<th>COMM</th>
<th>FD</th>
<th>ERR</th>
<th>PATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>26456</td>
<td>java</td>
<td>-1</td>
<td>2</td>
<td>/etc/acme-svr.config</td>
</tr>
<tr>
<td>26456</td>
<td>java</td>
<td>-1</td>
<td>2</td>
<td>/etc/acme-svr.config</td>
</tr>
<tr>
<td>26456</td>
<td>java</td>
<td>-1</td>
<td>2</td>
<td>/etc/acme-svr.config</td>
</tr>
</tbody>
</table>

# Trace

```
# trace -p `pidof java` -U 'r::SyS_open (retval==-2) "failed open"'
```

```
PID | TID | COMM | FUNC                     |
26456 | 26466 | java | SyS_open                
```

__open64+0x2d [libpthread-2.24.so]
fileOpen+0x7a [libjava.so]
java.io.FileInputStream::open0+0xbc [perf-16028.map]
Initializer::openConfigFile->
java.util.Scanner::<init>->
java.io.FileInputStream::<init>->
java.io.FileInputStream::open+0x0 [perf-16028.map]
Initializer$1::run->Initializer::access$100+0x0 [perf-16028.map]
java.lang.Thread::run+0x13d [perf-16028.map]
call_stub+0x88 [perf-16028.map]

...
Demo: Slow HTTP Requests

• Objective:
  Figure out why an HTTP client application occasionally makes very slow requests
$ java -cp bin Clienty good
Crawl complete, elapsed: 2241 milliseconds.
Crawl complete, elapsed: 1614 milliseconds.
Crawl complete, elapsed: 1442 milliseconds.
Crawl complete, elapsed: 1467 milliseconds.
Crawl complete, elapsed: 1601 milliseconds.
Crawl complete, elapsed: 1599 milliseconds.
...

$ java -cp bin Clienty bad
Crawl complete, elapsed: 6442 milliseconds.
Crawl complete, elapsed: 6060 milliseconds.
Crawl complete, elapsed: 6044 milliseconds.
Crawl complete, elapsed: 6092 milliseconds.
Crawl complete, elapsed: 6031 milliseconds.
Crawl complete, elapsed: 6023 milliseconds.
...

$ grep -A4 bad Clienty.java
    if (args[0].equals("bad")) {
        urls.add("https://i-dont-exist-at-all-20170126.com");
    } else {
        urls.add("https://facebook.com");
    }
```plaintext
# gethostlatency

<table>
<thead>
<tr>
<th>TIME</th>
<th>PID</th>
<th>COMM</th>
<th>LATms HOST</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:18:11</td>
<td>29003</td>
<td>java</td>
<td>6021.00 i-dont-exist-at-all-20170126.com</td>
</tr>
<tr>
<td>15:18:17</td>
<td>29003</td>
<td>java</td>
<td>6030.00 i-dont-exist-at-all-20170126.com</td>
</tr>
<tr>
<td>15:18:23</td>
<td>29003</td>
<td>java</td>
<td>6029.00 i-dont-exist-at-all-20170126.com</td>
</tr>
</tbody>
</table>

# trace -T -p `pidof java` 'c:getaddrinfo "resolving: %s", arg1' \  
  'r:c:getaddrinfo "done resolving: %d", retval'

<table>
<thead>
<tr>
<th>TIME</th>
<th>PID</th>
<th>TID</th>
<th>COMM</th>
<th>FUNC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16:21:55</td>
<td>15611</td>
<td>15612</td>
<td>java</td>
<td>getaddrinfo</td>
<td>resolving: i-dont-exist-...com</td>
</tr>
<tr>
<td>16:22:01</td>
<td>15611</td>
<td>15612</td>
<td>java</td>
<td>getaddrinfo</td>
<td>done resolving: -2</td>
</tr>
<tr>
<td>16:22:01</td>
<td>15611</td>
<td>15612</td>
<td>java</td>
<td>getaddrinfo</td>
<td>resolving: i-dont-exist-...com</td>
</tr>
<tr>
<td>16:22:07</td>
<td>15611</td>
<td>15612</td>
<td>java</td>
<td>getaddrinfo</td>
<td>done resolving: -2</td>
</tr>
<tr>
<td>16:22:07</td>
<td>15611</td>
<td>15612</td>
<td>java</td>
<td>getaddrinfo</td>
<td>resolving: i-dont-exist-...com</td>
</tr>
<tr>
<td>16:22:13</td>
<td>15611</td>
<td>15612</td>
<td>java</td>
<td>getaddrinfo</td>
<td>done resolving: -2</td>
</tr>
</tbody>
</table>

```

```c
#define ENOENT 2
```

gethostlatency (from BCC) traces DNS resolution latency
$ dig +multiline +answer any i-dont-exist-at-all-20170126.com

... 

;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 20042

... 

;; AUTHORITY SECTION:
com. 0 IN SOA a.gtld-servers.net. nstld.verisign-grs.com. (1486992927 ; serial
1800 ; refresh (30 minutes)
900 ; retry (15 minutes)
604800 ; expire (1 week)
86400 ; minimum (1 day)
)

;;; Query time: 3020 msec

...

$ dig +multiline +answer any i-dont-exist-at-all-20170126.com

... 

;; AUTHORITY SECTION:
com. 0 IN SOA a.gtld-servers.net. nstld.verisign-grs.com. (
$ dig +multiline +answer any i-dont-exist-at-all-20170126.com

... ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 20042

;; AUTHORITY SECTION:
com. 828 IN SOA a.gtld-servers.net. nstld.verisign-grs.com. (1486992927 ; serial
1800 ; refresh (30 minutes)
900 ; retry (15 minutes)
604800 ; expire (1 week)
86400 ; minimum (1 day)
)

;; Query time: 3020 msec
...

$ less Clienty.java
...
static {
    Security.setProperty(
        "networkaddress.cache.negative.ttl", "0");
}
...
Summary

• We have seen:
  ✓ Which production-ready tracing tools can be used with JVM apps
  ✓ How BPF changes the picture of Linux tracing
  ✓ To apply a performance checklist for JVM apps using BPF tools
  ✓ To conduct ad-hoc investigations with one-liners and custom tools
References

• BPF
  • https://github.com/torvalds/linux/tree/master/samples/bpf
  • https://www.kernel.org/doc/Documentation/networking/filter.txt
  • https://github.com/iovisor/bpf-docs

• BCC tutorials (by Brendan Gregg)
  • https://github.com/iovisor/bcc/blob/master/docs/tutorial.md
  • https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md
  • https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md

• JVM USDT probes
  • http://blog.sashag.net/2016/12/23/usdtbpf-tracing-tools-java-python-ruby-node-mysql-postgresql/
  • https://docs.oracle.com/javase/8/docs/technotes/guides/vm/dtrace.html
  • https://sourceware.org/systemtap/wiki/AddingUserSpaceProbingToApps
Thank You!

Slides:  https://s.sashag.net/jpoint17
Demos & labs:  https://github.com/goldshtn/linux-tracing-workshop

Sasha Goldshtein  
CTO, Sela Group  
goldshtn  
goldshtn