Observing HotSpot with SystemTap

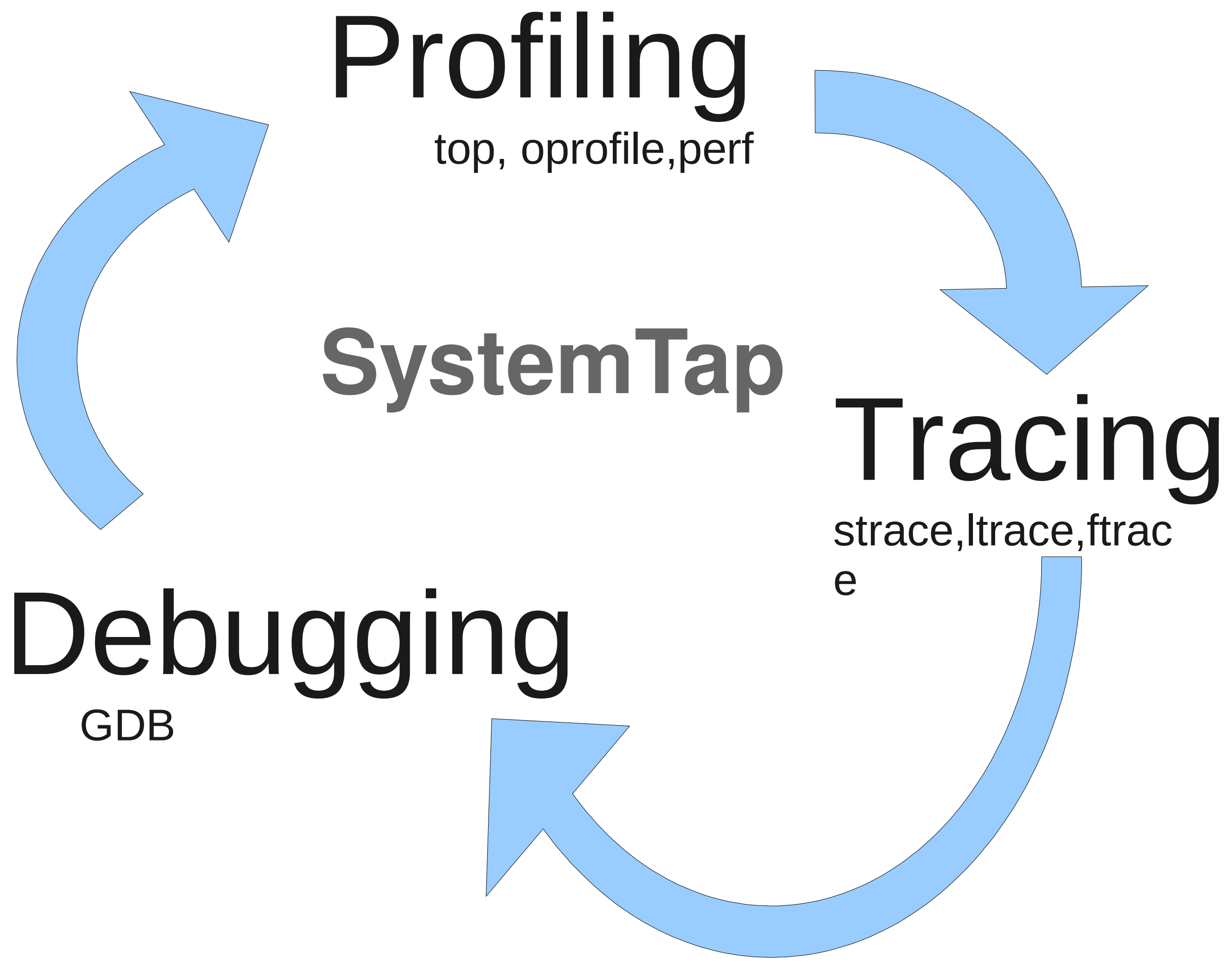
Mark Wielaard

Free Java
Devroom
Fosdem 2011

This Talk

- About SystemTap
  - The circle of observability
  - tracing, profiling, debugging
  - SystemTap basics
- Adding new event sources, specifically
  how they were added/extracted from HotSpot.
- Limitations and future directions

The circle of observability



Tracing

- Nice to have
  - Provides info while running
  - Did I pass GO?
  - Quick overview of code flow
- Limitations
  - Often specialized tools (strace, ltrace, ftrace)
  - Limited filtering
  - Overwhelming amount of information
    - Trying to “grep it out” can alter system under observation

Profiling

- Nice to have
  - Monitors while running (sampling)
  - Can (often) see systemwide
- Limitations
  - Often one kind of event (time)
  - After the fact analysis

Debugging

- Nice to have
  - Full context (variables, parameters, memory, registers, backtrace)
  - Conditional breakpoints
- Limitations
  - Stops the program under inspection
  - One program at a time (not system wide)

One tool to rule them all

- Unobtrusive, non-stop
- System wide
- Monitoring multiple event types

  Synchronous and Asynchronous
- Scriptable (in-place) filtering and statistics collection
- Safe (enforces limits, stops dumb things)

How does it look

- probe <event> { handler }
  - Where event is kernel.function, process.statement, timer.ms, begin, end, (tapset) aliases
- handler can have:
  - filtering/conditionals (if ... next)
  - control structures (foreach, while)
- Variables are primitive (number, string), associate arrays or statistical aggregate
- Helper functions (log, printf, gettimeofday, pid)

Tracing all functions in a program:

```
$ stap -e 'probe process("/bin/ls").function("*")
{ log(pp()) }' -c /bin/ls

process("/bin/ls").function("main@/usr/src/debug/
coreutils-7.6/src/ls.c:1225")
process("/bin/ls").function("set_program_name@/us
r/src/debug/coreutils-7.6/lib/progname.c:35")
process("/bin/ls").function("initialize_exit_fail
ure@/usr/src/debug/coreutils-
7.6/src/system.h:118")
process("/bin/ls").function("decode_switches@/usr
/src/debug/coreutils-7.6/src/ls.c:1497")
[...]
```

Showing all parameters of probed functions

```
$ stap -e 'probe process("/bin/ls").function("*")
{ log(probefunc() . ": " . $$parms) }' -c /bin/ls

main: argc=0x1 argv=0xbfd4c504
set_program_name: argv0=0xbfd4d568
initialize_exit_failure: status=0x2
[...]
```

System wide syscall probing

```
$ stap -e 'probe syscall.open
  {log(execname() . ": " . filename)}'

gnome-settings-: /proc/mounts
hald-addon-stor: /dev/sr0
devkit-disks-da: /dev/sr0
gpm: /dev/tty0
sendmail: /proc/loadavg
gnome-settings-: /proc/mounts
hald-addon-stor: /dev/sr0
devkit-disks-da: /dev/sr0
[...]
```

Keeping statistics for vfs.read (unread global aggregate variables are printed at and)

```
$ cat iotop2.stp
global reads
probe vfs.read {reads[execname()] <<< $count}

$ stap iotop2.stp
reads["simpress.bin"] @count=0x28e @min=0x4f
@max=0x2f060 @sum=0x4a3296 @avg=0x1d0b
reads["firefox"] @count=0x10c @min=0x1
@max=0x1000 @sum=0xab061 @avg=0xa35
reads["Xorg"] @count=0x2d @min=0x100 @max=0x1018
@sum=0x20c58 @avg=0xba6
reads["dbus-daemon"] @count=0x20 @min=0x800
@max=0x800 @sum=0x10000 @avg=0x800
```

Not just SystemTap

- Observability is not just fancy scripting
- The events and context need to be there
- Low Level & High Level

Low Level (events & contexts)

- Nicely covered
- Timers, processes, kprobes, uprobes (perf-counters, data watches)
- GCC debuginfo (dwarf) keeps improving
- Trick is to get more tools using them

High level (events & contexts)

- Through TapSets (aliases)
  - e.g. vfs, nfs, tcp tapsets (mapping to one or more alternative low-level function/statement probes)
- Through markers (in source)
  - Developer guided: “this might be interesting” points

A long lost friend

- dtrace – similar in spirit to SystemTap
- Not suitable for GNU/Linux

  (long sad licensing story)
- But they had a great idea:

  User Static Dynamic probes

#include <sys/sdt.h>

- Deprecate/Discourage STAP_PROBE
- Made DTRACE_PROBE macros source compatible
- Provide dtrace script that emulates build setup

Differences

- Only source level compatible
- DTRACE_PROBE_N macros expands differently
  - dtrace puts in fake function call, nops it out, knows where to find arguments on stack
  - Systemtap uses fancy GCC inline assembler to “capture” arguments.
  - Both store names, address, argument number in executable/library elf file section/note (but differently)

Mostly source level compatible

- Often just –enable-dtrace done
- But hotspot is written in c++ and gcc had some limitations that we had to work around for using the fancy inline assemble macros that create elf sections (so you need fairly new gcc 4.5+).
- HotSpot used some old dtrace tricks, like defining their own macros to create the “fake” dtrace functions. Rewrote to use more “standard” dtrace probe macros.
  - Found some small bugs. Thanks Keith McGuigan for helping to get them fixed.

What do you get?

- vm_init_begin, vm_init_end, vm_shutdown
- thread_start, thread_stop
- class_loaded, class_unloaded
- [mem_pool]_gc_begin, [mem_pool]_gc_end
- method_compile_begin, method_compile_end
- compiled_method_load, compiled_method_unload
- JNI calls

What do you get? (*)

- monitor_contended_enter/entered/exit
- monitor_wait, monitor_waited, monitor_notify[All]
- method_entry/exit
- object_alloc

What is that (*)?

- Some probe points only work in “special” degraded mode.
- -XX:+ExtendedDtraceProbes
- Makes all method/monitor/allocation calls go through common shared runtime functions.
  - Probe point gets set in this “catch all” function.

Example hotspot system and java user threads

```
$ stap -e 'probe hotspot.thread_*
  { printf("%s: %s%s\n",
           ctime(gettimeofday_s()),
           name, thread_name) }' \
     -c 'java Hello'
```

```
Sun Feb 06 10:48:39 2011: thread_start Reference Handler
Sun Feb 06 10:48:39 2011: thread_start Finalizer
Sun Feb 06 10:48:39 2011: thread_start Signal Dispatcher
Sun Feb 06 10:48:39 2011: thread_start CompilerThread0
Sun Feb 06 10:48:39 2011: thread_start CompilerThread1
Sun Feb 06 10:48:39 2011: thread_start Low Memory Detector
Sun Feb 06 10:48:40 2011: thread_start Thread-0
Sun Feb 06 10:48:40 2011: thread_stop Thread-0
[...]
```

Example what is the garbage collector doing

```
$ stap -e 'probe hotspot.*gc_*
  { printf("%s, %d, %s\n", ctime(gettimeofday_s()), pid(), probestr); }'

Tue Feb  8 13:40:35 2011, 8841, gc_begin(is_full=0)
Tue Feb  8 13:40:35 2011, 8841, mem_pool_gc_begin(manager='PS Scavenge',pool='Code
Cache',initial=2555904,used=715776,committed=2555904,max=50331648)
Tue Feb  8 13:40:35 2011, 8841, mem_pool_gc_begin(manager='PS Scavenge',pool='PS Eden
Space',initial=24772608,used=24772608,committed=24772608,max=518914048)
Tue Feb  8 13:40:35 2011, 8841, mem_pool_gc_begin(manager='PS Scavenge',pool='PS Survivor
Space',initial=4063232,used=0,committed=4063232,max=4063232)
Tue Feb  8 13:40:35 2011, 8841, mem_pool_gc_begin(manager='PS Scavenge',pool='PS Old
Gen',initial=65929216,used=0,committed=65929216,max=1054212096)
Tue Feb  8 13:40:35 2011, 8841, mem_pool_gc_begin(manager='PS Scavenge',pool='PS Perm
Gen',initial=21757952,used=17788456,committed=21757952,max=174063616)
Tue Feb  8 13:40:35 2011, 8841, mem_pool_gc_end(manager='PS Scavenge',pool='Code
Cache',initial=2555904,used=715776,committed=2555904,max=50331648)
Tue Feb  8 13:40:35 2011, 8841, mem_pool_gc_end(manager='PS Scavenge',pool='PS Eden
Space',initial=24772608,used=0,committed=24772608,max=518914048)
Tue Feb  8 13:40:35 2011, 8841, mem_pool_gc_end(manager='PS Scavenge',pool='PS Survivor
Space',initial=4063232,used=3791472,committed=4063232,max=4063232)
Tue Feb  8 13:40:35 2011, 8841, mem_pool_gc_end(manager='PS Scavenge',pool='PS Old
Gen',initial=65929216,used=0,committed=65929216,max=1054212096)
Tue Feb  8 13:40:35 2011, 8841, mem_pool_gc_end(manager='PS Scavenge',pool='PS Perm
Gen',initial=21757952,used=17788456,committed=21757952,max=174063616)
Tue Feb  8 13:40:35 2011, 8841, gc_end
```

Notice objects moving between memory pool generations.

What about context?

- Not that much on “java level” yet.
- We can get java level backtrace.
  - Pretty cool, “safe”, but somewhat resource intensive
  - Fragile...

Java – full backtrace on jni function call

```
$ stap -e 'probe hotspot.jni.GetArrayLength {print_jstack_full()}'
-c 'java Hello' | c++filt
jni_GetArrayLength
<Interpreter@0x14c14aa>
java/io/FileOutputStream.writeBytes([BII)V<Interpreter@0x14b9e61>
java/io/FileOutputStream.write([BII)V<Interpreter@0x14b9e61>
java/io/BufferedOutputStream.flushBuffer()V<Interpreter@0x14b9e61>
java/io/BufferedOutputStream.flush()V<Interpreter@0x14b9e61>
java/io/PrintStream.write([BII)V<Interpreter@0x14b9e61>
sun/nio/cs/StreamEncoder.writeBytes()V<Interpreter@0x14b9e61>
sun/nio/cs/StreamEncoder.implFlushBuffer()V<Interpreter@0x14b9e61>
sun/nio/cs/StreamEncoder.flushBuffer()V<Interpreter@0x14b9e61>
java/io/OutputStreamWriter.flushBuffer()V<Interpreter@0x14b9e61>
java/io/PrintStream.newLine()V<Interpreter@0x14b9e61>
java/io/PrintStream.println(Ljava/lang/String;)
V<Interpreter@0x14b9e61>
Hello.main([Ljava/lang/String;)V<StubRoutines_(1)@0x14b734c>
.L176
os::os_exception_wrapper(void (*)(JavaValue*, methodHandle*,
JavaCallArguments*, Thread*), JavaValue*, methodHandle*,
JavaCallArguments*, Thread*)
[...]
```

Future work/ideas – better context

- Get help from SystemTap unwinder

  for jstack/ustack (fixframe script function) and address/symbol lookup (vframe). Proposal from Keith McGuigan for dtrace. Adopt similar helpers handlers for systemtap. But nobody working on it right now.

  `http://article.gmane.org/gmane.comp.sysutils.dtrace.user/1097`

Future work/ideas – better java level probes

- Hook into jvmti callbacks. This is what oprofile does. You can get addresses of methods when compiled. Needs SystemTap to become more dynamic (currently calculates all probe point addresses ahead of time).

SystemTap – Q/A

- http://sourceware.org/systemtap/
  - Tutorial, Beginners Guide, Language reference, Examples, man pages, and more...
- Examples using HotSpot
  - http://icedtea.classpath.org/~vanaltj/stapexamples/