Dynamic Tracing and Performance Analysis Using SystemTap

Josh Stone
Software Engineer
Software and Solutions Group

August 4, 2008

Speaker Information

Josh Stone

Software Engineer

joshua.i.stone@intel.com

Intel Corporation

- Software and Solutions Group
  - >Performance, Analysis, and Threading Lab

SystemTap Tutorial

Getting Started

Writing Scripts

Examples and War Stories

Advanced Usage

Future and Conclusion

SystemTap

"painful to use", but more painful not to

Logo credit: Andy Fitzsimon

SystemTap Tutorial

Getting Started

Writing Scripts

Examples and War Stories

Advanced Usage

Future and Conclusion

Tutorial Live CD

- Fedora 9 i686 custom
- SystemTap and all prereqs included
- Please boot into live CD to follow along
  - Insert CD and reboot
  - Make sure BIOS boots from the CD
  - Twiddle thumbs until desktop arrives
- Tutorial slides and scripts are here:

  `/usr/share/doc/systemtap-lw08-1.0/`

Existing Tools

- On Linux
  - Profiling: oprofile, perfmon2, VTune
  - Stats: iostat, lockstat, netstat, vmstat
  - Tracing: strace
  - Printing: printk, printf
  - Debugging: gdb, kgdb
- Elsewhere
  - DTrace, apptrace, mdb, truss

Enter SystemTap

- Dynamic instrumentation tool for Linux systems
  - Complete framework for tracing, collecting data, and reporting
  - Flexible language lets *you* define the collection
  - Safely probe even production systems
  - Little to no overhead when not in use

- Contributions by Hitachi, IBM, Intel, Red Hat, and others

Target Users

- Usage model is flexible
- Applicable to many different user types:

Kernel Developers	**Technical Support**
Application Developers	**Researcher**
System Administrators	**Student**

Use for Developers

- **Kernel Developer:**
  - *How can I add debug statements easily without going through the insert/build/reboot cycle?*

- **Application Developer:**
  - *How can I improve the performance of my application in Linux?*

Use for Administration / Support

- **System Administrator:**
  - *Why do jobs occasionally take significantly longer to complete, or not complete at all?*

- **Technical Support:**
  - *How can I safely and easily collect data out of my customer's production system?*

Use in Academia

- **Researcher:**
  - *How would a proposed OS or hardware change affect system performance?*
- **Student:**
  - *How can I learn more about the inner workings of a kernel subsystem?*

Get it on Fedora

- Do it all with yum:

```
yum install systemtap
yum install kernel-devel gcc make
```

- Prepare kernel debug information:

```
yum install yum-utils
debuginfo-install kernel
```

Get it on Ubuntu

- Do it all with apt-get:

```
apt-get install systemtap
apt-get install linux-headers-generic gcc make
```

- Prepare kernel debug information:

```
apt-get install linux-image-debug-generic
ln -s /boot/vmlinux-debug-$(uname -r) \
      /lib/modules/$(uname -r)/vmlinux
```

User Security

- User **root**
  - Can always do anything
- Group **stapdev**
  - Can build and run any script
- Group **stapusr**
  - Can run blessed pre-built scripts `/lib/modules/$(uname -r)/systemtap/`

Obligatory Start

- Every language has its greeting:

```
# stap -e 'probe begin { println("Hello world!") }'
Hello world!
```

Obligatory Start

- Let's dissect it:

```
# stap -e 'probe begin { println("Hello world!") }'
  (1---)(2)(3----------)(4--------------------------)
```

1. 'stap' is the main executable for SystemTap
2. '-e' tells it to run a script from the next argument
3. 'probe begin' specifies a probe point at the start of execution
4. '{…}' define the probe handler

- See `man stap` for command-line details

SystemTap Tutorial

Getting Started

Writing Scripts

Examples and War Stories

Advanced Usage

Future and Conclusion

Script Language Basics

`global VAR1, VAR2`

- declares variables that are accessible anywhere

`probe PROBE { HANDLER }`

- defines a probe location and its handler

`function FUNC(ARG1, ARG2, ...) { BODY }`

- defines global functions for common code

```
# comment to the end of the line
// comment to the end of the line
/* enclosed comment */
```

Data Types and Operators

- Numeric type '`long`'
  - 64-bit signed integer
  - Supports normal arithmetic operators:
    `* / % + - >> << & ^ | && || = *= /= %= += -= >>= <<= &= ^= |= < > <= >= == !=`
- String type '`string`'
  - Zero-terminated string in a fixed-length buffer
  - Supports concatenation and comparison:
    `. .= < > <= >= == !=`
- Associative arrays (global only)
  - Mapping between long/string indexes to long/string/stat value
- Statistics (global only)
  - Accumulates longs with the `<<<` operator

Script statements

- Semicolon separator is optional
- Group compound statements with `{}`
- Branching:
  - `if (COND) STMT [else STMT]`
- Looping:
  - `while (COND) STMT`
  - `for (INIT; COND; ITER) STMT`
  - `foreach (VAR in ARRAY [limit NUM]) STMT`
  - `foreach ([VAR1, VAR2] in ARRAY [limit NUM]) STMT`
  - `break; continue;`
- Other:
  - `return [VAL]; next; delete VAR;`

Script Arguments

- Arguments can be pulled in from the script command line
  - Use `$1` .. `$N` to access numeric arguments
  - Use `@1` .. `@N` to access string arguments
  - Use `$#` for the number of arguments (or `@#` as a string)
  - Missing arguments trigger a compile-time error
- The argument tapset provides strings in C style
  - Use `argc` for the number of arguments
  - Index `argv[]` for strings of each argument, max 32

Writing Probes

- The syntax is simple:

```
probe PROBE {
      /* code to run when the PROBE hits */
}
```

- Use `next` to return from a probe handler
- What are the available probe points?

  `stap -l PROBE`

  - List all the points that would be probed if you had typed:
    `stap -e 'probe PROBE {}'`
  - Wildcards are generally allowed
  - See also `man stapprobes`

Probes: Script Lifetime

begin

– Runs when the script is starting, before any other probes

end

– Runs when the script is ending, after all other probes have finished

error

– Runs when the script is terminating due to errors, instead of **end**

- Each can also take a sequence number to define order

  begin(-1), begin, begin(1)

Probes: Timers

- Interval timer probes fire periodically

  `timer.s(10) # runs every 10 seconds`

  - Available units: `jiffies, s/sec, ms/msec, us/usec, ns/nsec`
  - Add variation with `.randomize(N)`
- The profile timer runs on every system tick

  `timer.profile`

  - Runs on all CPUs
  - Includes context of interrupted process

Probes: Kernel Functions

- Use kprobes and kretprobes, without the headache.

  `kernel.function("FUNC@FILE:LINE")`

  `module("NAME").function("FUNC@FILE:LINE")`

  - Probes any call to the named function
  - The `@FILE` and `:LINE` are optional
  - Wildcards are supported
- Suffixes
  - With `.inline`, only probe inlined functions
  - With `.call`, only probe non-inlined functions
  - With `.return`, probe the return of non-inlined functions
- Try running `stap -l 'kernel.function("*").call'`

Probes: Kernel Function Variables

- Function probes handlers can read parameters
  - Use **$NAME** to read the value as a long
  - For a **char***, convert it to a string
    - **kernel_string($NAME)** for pointers in kernel memory
    - **user_string($NAME)** for user memory (e.g. from system calls)
  - Use **$NAME->MEMBER** to read struct members

Probes: Tapsets

- Tapsets provide abstractions of common probe points

`syscall.*`

- –Probes each system call, with **`name`** and **`argstr`**

`process.*`

- –Probes process lifetime events

`socket.*`

- –Probes socket-related events

- And many more…

Writing Functions

- For common functionality in your script, use functions

```
function FNAME:type(ARG1:type, ARG2:type) {
    /* code to run when FNAME is called */
    return SOMETHING
}
```

  - The `:type`'s are optional, and may be `:string` or `:long`
    - Return type may be `:unknown` for no return value

Built-in Functions

- SystemTap includes many functions
  - Printing
    `print(), println(), printd(), printf(), sprint(), sprintln(), sprintd(), sprintf()`
  - Strings
    `strlen(), substr(), isinstr(), strtol()`
  - Timestamps
    `get_cycles(), gettimeofday_s(), gettimeofday_ns()`
  - Context
    `cpu(), execname(), tid(), pid(), uid(), backtrace(), print_stack(), print_backtrace(), pp(), probefunc(), probemod()`
  - See `man stapfuncs` for details and many more

Fibonacci Example

```
# cat fib1.stp
 probe begin { println(fib($1)); exit() }
 function fib(n) {
     if (n < 0) return 0
     if (n == 1) return 1
     return fib(n - 1) + fib(n - 2)
 }
# stap fib1.stp 4
 3
```

- Now run it again to compute fib(6)
- How about fib(10)?

Safety in Recursion

- Kernel stack space is limited, and fatal to overflow
- SystemTap variables are allocated off-stack
  - Still, fixed resources have limits
- The runtime explicitly checks the call depth (nesting) on each call, and errors out if too deep

Fibonacci Example 2

```
# cat fib2.stp
 probe begin { println(fib($1)); exit() }
 global fibdata
 function fib(n) {
     fibdata[0] = 0
     fibdata[1] = 1;
     for (i=2; i<=n; ++i)
         fibdata[i] = fibdata[i-1] + fibdata[i-2]
     return fibdata[n]
 }
# stap fib2.stp 10
 55
```

- How about fib(3000)?

Safety in Array Size

- Probes never allocate memory in the handler
  - Allocation takes too much time
  - Allocating too much kernel memory starves the system
- Thus, all arrays are pre-allocated
  - When the array gets too big, an error is thrown

Fibonacci Example 2, continued

- Add a "delete" statement to the for-loop to limit the array size
  - Individual entries can be removed with "`delete ARRAY[INDEX]`"
  - Don't forget to add `{}` to allow multiple statements in the for-loop

- Now can you compute fib(3000)?
- How about fib(10000)?

Safety in Probe Execution Time

- Lengthy probe handlers could degrade the system
- Infinite probe handlers would be fatal
- SystemTap counts statements and enforces an upper bound
  - Running too long throws an error

- Both of these will be detected and killed

```
while (1) { ... } /* deliberate infinite loop */
for (i=0; i<10; +i) { ... } /* innocent typo */
```

Safety Summary

- To ensure safe execution, the script compiler guarantees:
  - Limited recursion depth
  - Limited array size
  - Limited execution time
- Data checks are also performed
  - Divide by zero
  - String overflow
  - Pointer access

- **If you find a vulnerability, we want to know!**

A Tracing Example

- Simple command line to trace all "open" system calls:

```
# stap -e 'probe syscall.open {
    printf("%s[%d] open(%s)\n",
        execname(), pid(), argstr)
}'
irqbalance[2129] open("/proc/net/dev", O_RDONLY)
sendmail[2486] open("/proc/loadavg", O_RDONLY)
```

(etc.)

A Bigger Tracing Example

- Simple command line to trace all system calls:

```
# stap -e 'probe syscall.* {
    if (pid() == stp_pid()) next
    printf("%s[%d] %s(%s)\n",
        execname(), pid(), name, argstr)
}'
```

(lots of output...)

- Q: Why would I want to filter out a certain PID?

Associative Arrays

- Indexed by one or more long and/or string indexes
- Value can be a long, a string, or a statistical accumulation
- Arrays have a fixed size, which can be declared

```
global foo # foo gets the default size
global bar[100] # bar will have 100 slots
```

- Test membership with the "in" operator

```
if (42 in foo) println("foo has 42")
if (["bar", 42] in foo) println("foo has [\"bar\", 42]")
```

- Iterate all entries with "foreach"

```
foreach (x in foo) printf("foo[%d] = %d\n", x, foo[x])
foreach ([s, x] in foo)
     printf("foo[%s, %d] = %d\n", s, x, foo[x])
```

Example Array

```
# cat syscount.stp
 global syscalls
 probe syscall.* { syscalls[name] += 1 }
 probe timer.s(10) {
     printf("\n%8s  %s\n", "count", "name")
     foreach(name in syscalls- limit 10)
         printf("%8d  %s\n", syscalls[name], name)
     delete syscalls
 }

# stap syscount.stp
       98 automount
       22 stapio
 …
```

Statistics

- Scalable accumulation with **<<<**
- Aggregates values only when read
- Extract numeric values with function-like operators

  `@count(v) @sum(v) @min(v) @max(v) @avg(v)`
- Extract histograms that may be printed, indexed, or iterated

  `@hist_linear(v, start, stop, interval)`

  `@hist_log(v)`

Example Statistic

```
# cat syscount2.stp
 global syscalls
 probe syscall.* { syscalls[name] <<< 1 }
 probe timer.s(10) {
     printf("\n%8s  %s\n", "count", "name")
     foreach(name in syscalls- limit 10)
         printf("%8d  %s\n",
             @count(syscalls[name]), name)
     delete syscalls
 }
# stap syscount2.stp
       98 automount
       22 stapio
 …
```

SystemTap Tutorial

Getting Started

Writing Scripts

Examples and War Stories

Advanced Usage

Future and Conclusion

Track Scheduling Time

- Is it bad to have a lot of processes/threads?
  - Just resident -- not necessarily bad
  - Active & frequently switching -- may be oversubscribed

- Use schedtime.stp
  - Peek into the kernel scheduler
  - Visualize current activity
  - Evaluate system load

Track Scheduling Time (script)

```
# cat schedtime.stp
  global timestamp, stat

  probe scheduler.cpu_on {
      if (!idle)
          timestamp[cpu()] = gettimeofday_ns()
  }

  probe scheduler.cpu_off {
      if (!idle && timestamp[cpu()])
          stat[cpu()] <<< gettimeofday_ns() - timestamp[cpu()]
  }

  probe timer.s(2) {
      printf("\n=========================================\n")
      foreach (cpu+ in stat) {
          printf("\nCPU%d  count %d  min %d  max %d  avg %d\n", cpu,
                  @count(stat[cpu]), @min(stat[cpu]),
                  @max(stat[cpu]), @avg(stat[cpu]))
          print(@hist_log(stat[cpu]))
      }
      delete stat
  }
```

Track Scheduling Time (output)

```
# stap schedtime.stp

 ==================================================

 CPU0  count 365  min 3299  max 175394  avg 9042
  value |-------------------------------------------------- count
    512 |                                                     0
   1024 |                                                     0
   2048 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         210
   4096 |@@@@@@@@@@@@@@@@@@@@@@                             112
   8192 |@                                                    6
  16384 |@@@                                                 18
  32768 |@@                                                  13
  65536 |                                                     3
 131072 |                                                     3
 262144 |                                                     0
 524288 |                                                     0

 CPU1  count 117  min 3930  max 242673  avg 10635
 ...
```

Track Scheduling Time (follow-up)

- Possible modifications
  - Aggregate by process names
  - Report the most frequent switchers

Top Process I/O

- The disk is constantly busy -- why?
  - Kill the rouge process dragging down the system
  - Find ways to tweak I/O usage

- Use pid-iotop.stp
  - Quickly see which processes are responsible for the most I/O

Top Process I/O (script)

```
# cat pid-iotop.stp
 # Based on a script by Mike Grundy and Mike Mason from IBM
 global reads, writes
 probe vfs.read { reads[pid()] += bytes_to_read }
 probe vfs.write { writes[pid()] += bytes_to_write }

 # print top 5 IO users by pid every 5 seconds
 probe timer.s(5) {
     printf ("\n%-10s\t%10s\t%15s\n", "Process ID",
         "KB Read", "KB Written")
     foreach (id in reads- limit 5)
         printf("%-10d\t%10d\t%15d\n", id,
             reads[id]/1024, writes[id]/1024)
     delete reads
     delete writes
 }
```

Top Process I/O (output)

```
# stap pid-iotop.stp

 Process ID          KB Read          KB Written
 25553                  3216                   0
 4272                     24                   0
 4253                     16                   0
 4048                     16                   0
 4230                     16                   0

 Process ID          KB Read          KB Written
 25553                  3328                   0
 19033                    16                   0
 4253                     16                   0
 4246                     12                   0
 2132                      8                   0
```

Top Process I/O (follow-up)

- Possible modifications
  - Sort by writes or reads+writes
  - Report the top I/O users instead

Kernel Profiling

- What's my kernel up to?
  - Perhaps top is reporting high %sys -- why?

- Use pf2.stp
  - Sample kernel function location
  - Report hotspots

Kernel Profiling (script)

From http://sourceware.org/systemtap/wiki/WSKernelProfile

```
# cat pf2.stp
 global profile, pcount
 probe timer.profile {
   pcount <<< 1
   fn = probefunc ()
   if (fn != "") profile[fn] <<< 1
 }
 probe timer.ms(4000) {
   printf ("\n--- %d samples recorded:\n", @count(pcount))
   foreach (f in profile- limit 10) {
     printf ("%s\t%d\n", f, @count(profile[f]))
   }
   delete profile
   delete pcount
 }
```

Kernel Profiling (output)

```
# stap pf2.stp
 --- 109 samples recorded:
 mwait_idle 71
 check_poison_obj 4
 _spin_unlock_irqrestore 2
 dbg_redzone1 1
 kfree 1
 kmem_cache_free 1
 _spin_unlock_irq 1
 end_buffer_write_sync 1
 lock_acquire 1

 --- 108 samples recorded:
 mwait_idle 91
 check_poison_obj 3
 _spin_unlock_irq 2
 delay_tsc 1
```

Kernel Profiling (follow-up)

- Possible modifications
  - Record stack traces
  - Aggregate by PID or CPU

Function Call Counts

- Which functions are frequently called?

- Use callcount.stp
  - Report call counts for all probed functions
  - Outliers should be very obvious

Function Call Counts (script)

From http://sourceware.org/systemtap/wiki/WSFunctionCallCount

```
# cat callcount.stp
 # probe every function in any C file under mm/
 probe kernel.function("*@mm/*.c") {
   called[probefunc()] <<< 1  # add a count efficiently
 }
 global called
 probe end, timer.ms(30000) {
   foreach (fn+ in called)  # Sort by function name
   #        (fn in called-)  # Sort by call count (in
                             # decreasing order)
     printf("%s %d\n", fn, @count(called[fn]))
   exit()
 }
```

Function Call Counts (output)

```
# stap callcount.stp
 ClearSlabFrozen 50
 INIT_LIST_HEAD 3641
 PageUptodate 2
 SetSlabFrozen 50
 SlabFrozen 1438
 __ClearPageTail 7
 __SetPageTail 8
 __alloc_pages 204
 ...
 zap_pte_range 52
 zone_statistics 204
 zone_to_nid 21
 zone_watermark_ok 204
 zonelist_policy 204
```

Function Call Counts (follow-up)

- Possible modifications
  - Parameterize the probe point and explore
  - Filter only the top N results
  - Probe returns too, and aggregate call times

Call Graph Tracing

- When I call a function, what happens next?

- Use para-callgraph.stp
  - Start tracing when a trigger is called
  - Visualize all functions that are invoked
  - See how long each step takes

Call Graph Tracing (script)

From http://sourceware.org/systemtap/wiki/WSCallGraph

```
# cat para-callgraph.stp
  function trace(entry_p) {
    if(tid() in trace)
        printf("%s%s%s\n",thread_indent(entry_p),
                          (entry_p>0?"->":"<-"),
                          probefunc())
 }

 global trace
 probe kernel.function(@1).call {
    if (pid() == stp_pid()) next # skip our own helper process
    trace[tid()] = 1
    trace(1)
 }

 probe kernel.function(@1).return {
    trace(-1)
    delete trace[tid()]
 }
 probe kernel.function(@2).call { trace(1) }
 probe kernel.function(@2).return { trace(-1) }
```

Call Graph Tracing (output)

```
# stap para-callgraph.stp sys_read '*@fs/*.c'
      0 clock-applet(4325):->sys_read
      9 clock-applet(4325): ->fget_light
     13 clock-applet(4325): <-fget_light
     18 clock-applet(4325): ->vfs_read
     24 clock-applet(4325):  ->rw_verify_area
     29 clock-applet(4325):  <-rw_verify_area
     36 clock-applet(4325):  ->do_sync_read
     42 clock-applet(4325):  <-do_sync_read
     46 clock-applet(4325): <-vfs_read
     50 clock-applet(4325):<-sys_read
```

Call Graph Tracing (follow-up)

- Possible modifications
  - Explore different kernel subsystems
  - Add filters to narrow down results

SystemTap Tutorial

Getting Started

Writing Scripts

Examples and War Stories

Advanced Usage

Future and Conclusion

Advanced SystemTap

- Guru mode
- Live kernel patching
- Prototyping new modules
- Building test suites for kernel code
- Define and refine tapsets

SystemTap Guru

"Stop protecting me -- I know what I'm doing!"

- Ok, you're the boss. Just use the `-g` option.

- All `$target` variables become writable
- You can write embedded-C in your scripts!

- Send all complaints to $USER@localhost

Using Embedded-C

- At the top level, bracket C code in **%{ ... %}**
  - Add new functions, #include files, etc.
- Create a function that's callable from the script language
  - Arguments are passed in **THIS->argname**
  - Return value is stored in **THIS->__retvalue**
  - Example:

```
function divide:long(num:long, den:long) %{
    THIS->__retvalue = THIS->num / THIS->den;
%}
```

  - Q: What are two problems with this divide function?

Live Kernel Patching

- February 2008: local-root exploit in sys_vmsplice!
  - Fix requires a new kernel
  - Planning downtime is non-trivial for many admins

- Solution: patch it with a simple script!

```
# stap -g -e 'probe syscall.vmsplice {
    printf("blocking vmsplice (%s) uid %d pid %d exec %s\n",
        argstr, uid(), pid(), execname())
    $nr_segs = 0
}'
```

(posted by Frank Ch. Eigler, http://tinyurl.com/4edbsm)

Easy Kernel Modules

- With guru-mode C code, write *anything* you want

```
%{
/* write #include's, functions, etc. */
%}

function init() %{
    /* call various init routines... */
    if (error)
        CONTEXT->last_error = "some error string";
%}

function shutdown() %{
    /* call various shutdown routines... */
%}

probe begin { init() }
probe end { shutdown() }
```

Kernel testing

- Place probes on key points of your own code
- Check assertions about correct behavior
- Modify variables to inject faults

- Example: SCSI Fault Injection
  http://sourceforge.net/projects/scsifaultinjtst

Writing Tapsets

- Write a probe alias to abstract implementation details
- Extract useful arguments

```
probe syscall.open =
        kernel.function("sys_open") ?,
        kernel.function("compat_sys_open") ?,
        kernel.function("sys32_open") ?
{
    name = "open"
    filename = user_string($filename)
    flags = $flags
    mode = $mode
    if (flags & 64)
        argstr = sprintf("%s, %s, %#o", user_string_quoted($filename),
            _sys_open_flag_str($flags), $mode)
    else
        argstr = sprintf("%s, %s", user_string_quoted($filename),
            _sys_open_flag_str($flags))
}
```

SystemTap Tutorial

Getting Started

Writing Scripts

Examples and War Stories

Advanced Usage

Future and Conclusion

SystemTap Future Features

- Kernel Markers
  - Static, low-latency probe points in the kernel
  - Works now, but few markers are available
    `stap -l 'kernel.mark("*")'`
- User-space Probes
  - Some prototype availability today with CONFIG_UTRACE=y

  `process(PID), process("PATH")`

    - Base for probes on a certain PID or executable PATH
    - Probe raw system calls with `.syscall`, parameter `$syscall`
    - Probe lifetime events for the process or its threads:
      `.begin, .end, .thread.begin, .thread.end`

  `process(PID).statement(ADDRESS).absolute`

    - Place a probe at a raw address in a process

Conclusion

- SystemTap is:
  - Dynamic
  - Safe
  - Flexible

  - Not so painful
  - Indispensable

Disclaimer

- This work represents the view of the author and does not necessarily represent the view of Intel.
- Intel is a registered trademark of Intel Corporation.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.

Project Contact Information

- Web: http://sourceware.org/systemtap/
  - Wiki: http://sourceware.org/systemtap/wiki/
  - FAQ: http://sourceware.org/systemtap/wiki/SystemTapFAQ
- Mailing list: systemtap@sourceware.org
- IRC: join #systemtap on irc.freenode.net