Dynamic Linux Kernel Instrumentation with SystemTap

Eugene Teo, RHCE, RHCX
Linux Enterprise Application Porting (LEAP) Engineer
Red Hat Asia Pacific
Previous Linux Monitoring Tools

- Examples: `ps`, `netstat`, `vmstat`, `iostat`, `sar`, `strace`, `top`, `oprofile`, etc
- Drawbacks:
  - Application-centric tools are narrow in scope
  - Tools with system-wide scope present a static view of system behaviour but does not let you probe further
  - Many different tools and data sources but no easy way to integrate
- Many kinds of problems are not readily exposed by traditional tools:
  - Interactions between applications and the operating system
  - Interactions between processes and kernel subsystems
  - Problems that are obscured by ordinary behaviour and require examination of an activity trace
SystemTap

• A tool to enable a deeper look into a running system:
  - Provides a high-level script language to instrument unmodified running kernels
  - Exposes a live system activity and data
  - Provides performance and safety by careful translation to C
  - Includes growing library of reusable instrumentation scripts
• Started January 2005
• Free/Open Source Software (GPL)
• Active contributions from Red Hat, Intel, IBM, Hitachi, and others
SystemTap Target Audience

- Kernel Developer: I wish I could add a debug statement easily without going through the compile/build cycle.
- Technical Support: How can I get this additional data that is already available in the kernel easily and safely?
- Application Developer: How can I improve the performance of my application on Linux?
- System Administrator: Occasionally jobs take significantly longer than usual to complete, or do not complete. Why?
- Researcher: How would a proposed OS/hardware change affect system performance?
SystemTap Overall Diagram

- **debug-info ELF objects**
- **systemtap translator**
  - **parse**
  - **elaborate**
  - **translate**
- **build**
- **load/run**
- **store output**
- **stop/unload**

- **probe.ko**
- **kernel**
  - **kprobes**
  - **relayfs**
  - **profiling**

- **probe.stp**
- **script library**

- **probe.c**
- **runtime, C tapsets**
- **probe.out**
A tapset defines:

- Probe points/aliases: symbolic names for useful instrumentation points
- Useful data values that are available at each probe point

Written in script and C by developers knowledgeable in the given area

Tested and packaged with SystemTap
Runtime Library

- Implements some utilities:
  - Associative arrays, statistics, counters
  - Stack trace, register dump, symbol lookup
  - Safe copy from userspace
  - Output formatting and transport
- Could also be used by C programmers to simplify writing raw kprobes-based instrumentation
Kprobes

- C API to allow dynamic kernel instrumentation
- Probe Point: An instruction address in the kernel
- Probe Handler: An instrumentation routine, as function pointer
- Replace the instruction at the probe points with a breakpoint instruction
- When the breakpoint is hit, call the probe handler
- Execute the original instruction, then resume
Kprobes Limitations

- C API
- No checking that probe point is at instruction boundary
- Kprobes-based code is hard to maintain and port due to hard coding of addresses
- No library of probes for common tasks
- No convenient access to local variables
- Requires significant kernel knowledge
SystemTap Safety Goals

- For use in production environment – aiming to be crash-proof
- Uses existing compiler tool chain, kernel
- Safe mode: Restricted functionality for production
- Guru mode: Full feature set for development, debugging
- Static analyser:
  - Protection against translator bugs and users errors
  - Detects illegal instructions and external references
SystemTap Safety Features

- No dynamic memory allocation
- Types & types conversions limited
- No assembly or arbitrary C code (unless -g or Guru mode is used)
- Kernel functions known to crash system when probed are blacklisted
  - default_do_nmi, __die, do_int3, do_IRQ, do_page_fault, do_trap, do_sparc64_fault, do_debug, oops_begin, oops_end, etc
  - Discovered with our dejagnu stress test suite
- Limited pointer operations
Probe Scripting Language

- Awk/C-like scripting language
- Limited number of types:
  - 64-bit numbers, strings, associative arrays, statistics
- Full control structures (conditionals, oops, functions)
- Safety features:
  - Full static type checking, automatic type inference
  - No dynamic memory allocation
  - Bounded execution space and time
  - No assembly or arbitrary C code (except in guru mode)
  - Protected access to “$target” values in kernel space
Dynamic Probing

- Several underlying interfaces for inserting probes
  - Probepoints provide a uniform interface for identifying events of interest
- Synchronous probepoints
  - kprobes, jprobes, kretprobes (dynamic)
  - SystemTap Marks (static)
- Asynchronous events
  - Timers, Performance counters
Static Probing

- Probe point: wherever hooks are compiled in
- Fixed probe handler: collect fixed pool of context data, dump it to buffer; off-line post-processing
- Low cost dormant probes
- Dispatch cost low
Static Instrumentation Markers

- Decoupling probe *point* and *handler*

- To create: place it, name it, parametrize it. That's it:
  \[ \text{STAP\_MARK\_NN}(\text{context\_switch}, \text{prev}\to\text{pid}, \text{next}\to\text{pid}); \]

- To use from systemtap:
  \[ \text{probe kernel\_mark(“context\_switch”) \{ print($arg1) \}} \]
  
  #define STAP\_MARK\_NN(n,a1,a2) do {
    static void (*__stap\_mark\_##n##\_NN)(int64\_t,int64\_t);
    if (unlikely (__stap\_mark\_##n##\_NN)) {
      (void) (__stap\_mark\_##n##\_NN((a1),(a2)));
    }
  } while (0)
Static Instrumentation Markers

- Marker-based top-process listing; placing a marker in a sensitive spot (context switching)

```c
1796 /*
1797 * context_switch - switch to the new MM and the new
1798 * thread's register state.
1799 */
1800 static inline struct task_struct *
1801 context_switch(struct rq *rq, struct task_struct *prev,
1802                     struct task_struct *next)
1803 {
1804     struct mm_struct *mm = next->mm;
1805     struct mm_struct *oldmm = prev->active_mm;
1806 ...
1829     /* Here we just switch the register state and the stack. */
1830     STAP_MARK_NN(context_switch, prev->pid, next->pid);
1831     switch_to(prev, next, prev);
1832
1833     return prev;
1834 }
```
Static Instrumentation Markers

- probe kernel.mark("context_switch") {
  switches ++  # count number of context switches
  now = get_cycles()
  times[\$arg1] += now-lasttime  # accumulate cycles spent in process
  execnames[\$arg1] = execname()  # remember name of pid
  lasttime = now
}

probe timer.ms(3000) {  # every 3000 ms
  printf ("\n%5s %20s %10s ( %d switches)\n",
    "pid", "execname", "cycles", switches);
  foreach ([pid] in times-)  # sort in decreasing order of cycle-count
    printf ("%5d %20s %10d\n", pid, execnames[pid], times[pid]);
  # clear data for next report
  delete times
  switches = 0
}
...

- # stap mark-top.stp
  pid   execname        cycles   (1813 switches)
  0     swapper         764411819
  4473   X                51465833
  4538   gnome-terminal   33217978
  4745   firefox-bin     24762308
  ...

Live Demos

- Which process in the running system uses `open(2)`?
  ```c
  int open(const char *pathname, int flags);
  int open(const char *pathname, int flags, mode_t mode);
  ```
- Which system calls are triggered when executing `bash`?
- What programs/scripts are executed when you run a command?
- Which are the top 10 applications that use `sys_ioctl`?
- Use `plimits.stp` to check the rlimits of any arbitrary process
- Use `pfiles.stp` to check the currently opened file descriptors of any arbitrary process
- Use `udpstat.stp` to analyse the UDP traffic in the system
- Hook the `kbd_event` handler to perform something
Things that you can write

- Block I/O submissions & completions
Things that you can write

- Is CPU busy now?
SystemTap Demo Scripts

- Scripts demonstrating various SystemTap features can be found at http://sourceware.org/systemtap/documentation.html
  - top.stp - print the top twenty system calls.
  - prof.stp - simple profiling.
  - keyhack.stp - modifying variables in the kernel.
  - kmalloc.stp - statistics example.
  - kmalloc2.stp - example using arrays of statistics.
  - ansi_colors.stp – example using \0?? to display ansi colours

- For example:
  - $ stap top.stp
SystemTap Availability

- SystemTap is still evolving rapidly
  - Latest sources available at http://sourceware.org/systemtap
  - Anonymous CVS access
- Distribution & architecture support
  - Red Hat Enterprise Linux 4 from U2 (technology preview)
    - x86, EM64T/AMD64, Itanium2
  - Fedora Core 4, 5 & 6
    - x86, EM64T/AMD64, Itanium2, PPC
SystemTap Packages

- Main RPM is **systemtap**
  - **stap**, **stapd**
  - SystemTap Runtime
  - **Tapsets**
- **Man pages**: **stap(1)**, **stapfuncs(5)**, **stapprobes(5)**, **stapex(5)**

- **SystemTap requires**
  - **gcc**
  - **kernel-devel**
  - **kernel-debuginfo**
SystemTap Kernel Packages

- SystemTap requires support packages for the kernels in use
- **kernel-devel RPMs**
  - Provide headers, Makefiles and configuration information to allow modules to be built against a packaged kernel
- **kernel-debuginfo RPMs**
  - Provide source and debug symbols for packaged kernels
  - Debug information in DWARF format
    - Allows location of inlines, local variables, macros, line numbers
    - Due to the volume of data kernel-debuginfo RPMs are large
  - But FC6 and RHEL5 will use modular debuginfo packages
War Stories

- We are compiling a list of SystemTap stories, and interesting demos
- If you have a SystemTap success story, do share with us at http://sourceware.org/systemtap/wiki/WarStories
Further Information

- Website: http://sources.redhat.com/systemtap
- Wiki: http://sources.redhat.com/systemtap/wiki
- Mailing list: systemtap@sources.redhat.com
- IRC channel: #systemtap on irc.freenode.net
Thank you!

Eugene Teo, eteo@redhat.com