redhat.

Dynamic Linux Kernel Instrumentation with SystemTap

張漢輝
紅帽亞太區

台北開放原始碼軟體使用者社群（TOSSUG）
10 月 03 日 （周二） Mix Coffee & Tea

Previous Linux Monitoring Tools

- **Examples: ps, netstat, vmstat, iostat, sar, strace, 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 them
- **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

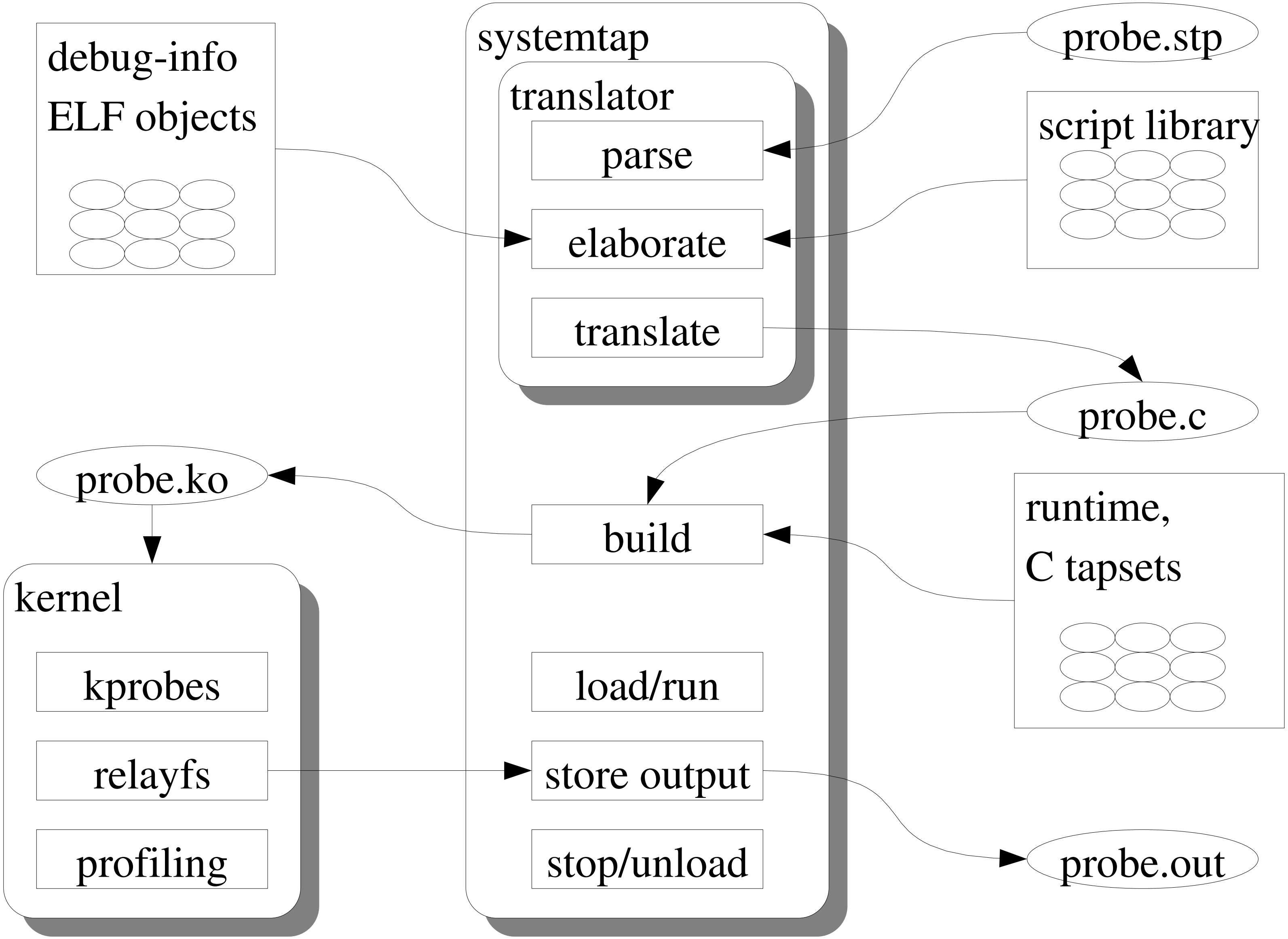
SystemTap

- **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



Tapsets

- **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

  ```
  kp.addr = (kprobe_opcode_t *)
  kallsyms_lookup_name("<kernel function name>");
  ```

- Probe Handler: An instrumentation routine, as function pointer

  ```
  kp.pre_handler=handler_pre;
  kp.post_handler=handler_post;
  kp.fault_handler=handler_fault;
  ```

- 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

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**

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:**

  ```
  STAP_MARK_NN(context_switch,prev->pid,next->pid);
  ```

- **To use from SystemTap:**

  ```
  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)**

```
/*
 * context_switch - switch to the new MM and the new
 * thread's register state.
 */
static inline struct task_struct *
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next)
{
        struct mm_struct *mm = next->mm;
        struct mm_struct *oldmm = prev->active_mm;

...
        /* Here we just switch the register state and the stack. */
        STAP_MARK_NN(context_switch, prev->pid, next->pid);
        switch_to(prev, next, prev);

        return prev;
}
```

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
   ...
  ```

Demonstrations

- **Let's trace and analyze open(2)**

  ```
  int open(const char *pathname, int flags);
  int open(const char *pathname, int flags, mode_t
  mode);
  ```

- **Which system calls were executed when you run bash?**

- **What happens when you run a command?**

- **Which are the top 10 processes that use `sys_ioctl`?**

- **Use `plimits.stp` to find out resource limits of processes**

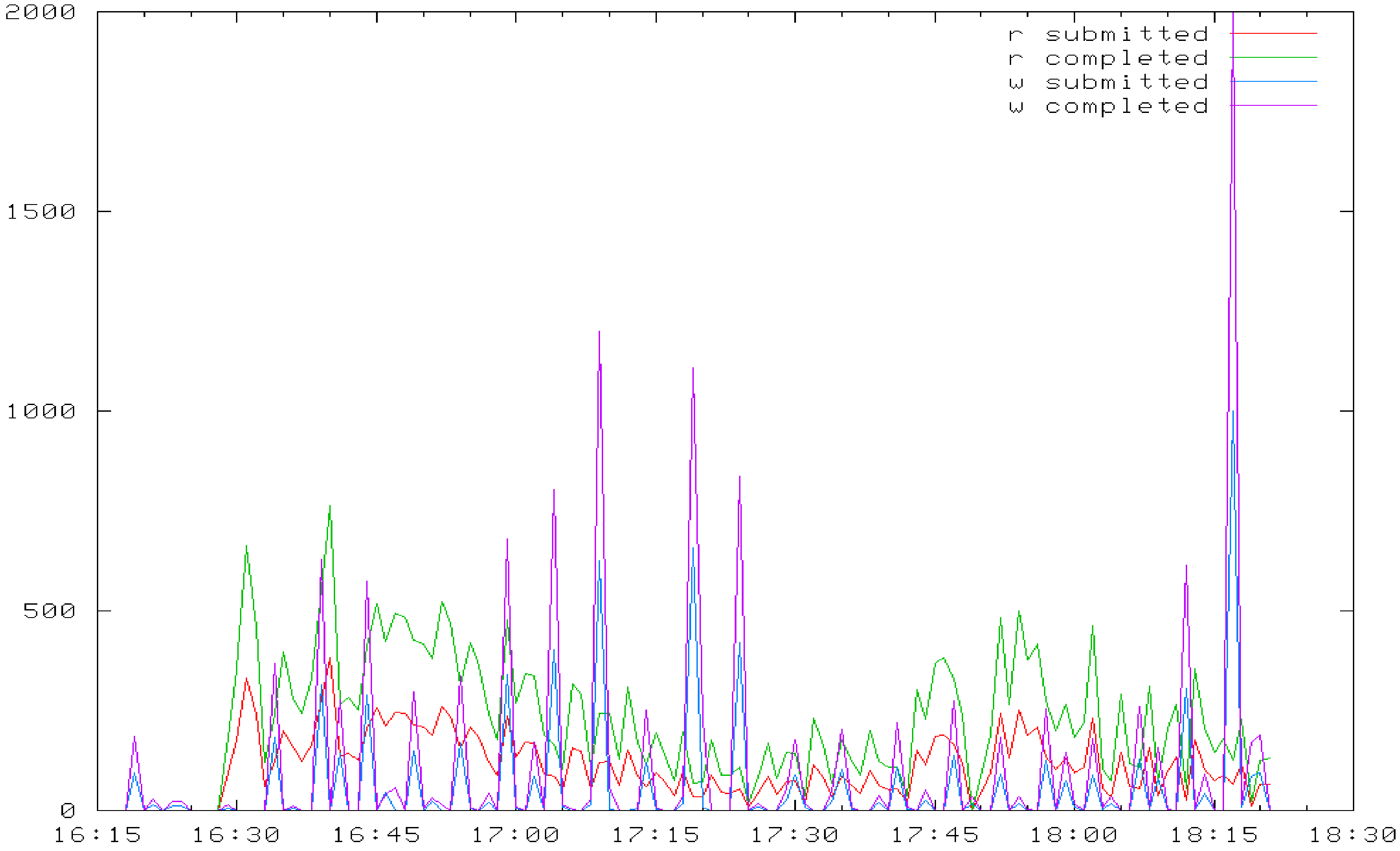
- **Use `pfiles.stp` to find out opened fd of processes**

- **Use `udpstat.stp` to find out udp statistics**

- **Hook the `kbd_event` to add functionalities to it**

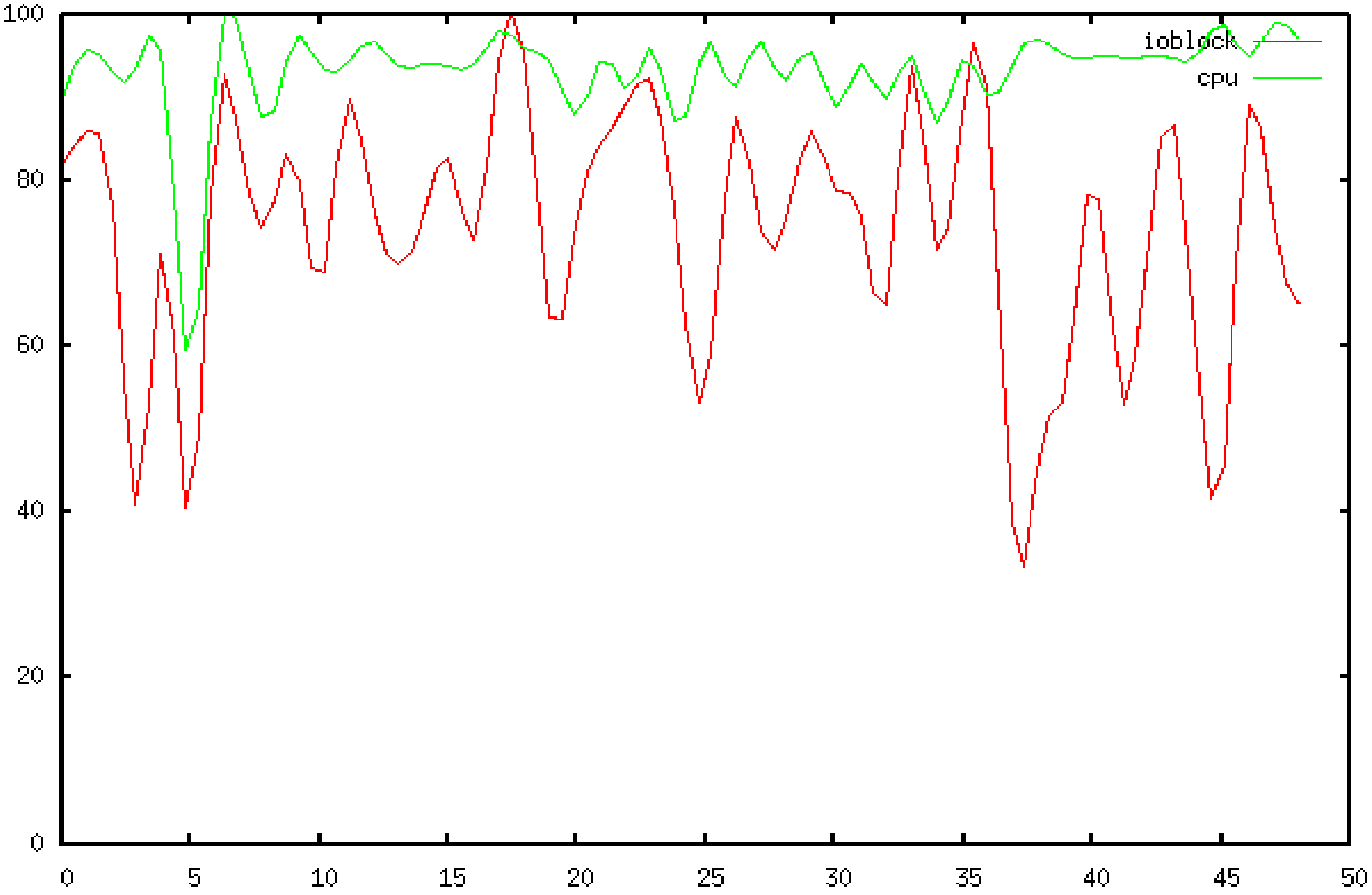
Things that you can write

- **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`

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

Dynamic Linux Kernel Instrumentation with SystemTap

張漢輝
紅帽亞太區
台北開放原始碼軟體使用者社群（TOSSUG）
10 月 03 日 （周二） Mix Coffee & Tea

Previous Linux Monitoring Tools

- **Examples: ps, netstat, vmstat, iostat, sar, strace, 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 them
- **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

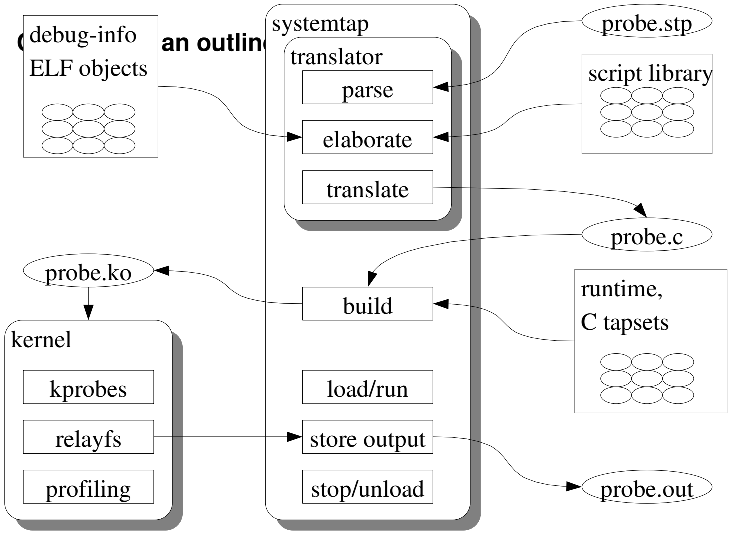
- **A tool to enable a deeper look into a running system:**
  - Provides a high-level script language to instrument unmodified running kernels
  - Exposes a live system activity and data
  - Provides performance and safety by careful translation to C
  - Includes growing library of reusable instrumentation 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

- C an outlin



Tapsets

- **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
  ```
  kp.addr = (kprobe_opcode_t *)
  kallsyms_lookup_name("<kernel function name>");
  ```
- Probe Handler: An instrumentation routine, as function pointer
  ```
  kp.pre_handler=handler_pre;
  kp.post_handler=handler_post;
  kp.fault_handler=handler_fault;
  ```
- 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

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**

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:**
  `STAP_MARK_NN(context_switch,prev->pid,next->pid);`
- **To use from SystemTap:**
  `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)**

```
/*
 * context_switch - switch to the new MM and the new
 * thread's register state.
 */
static inline struct task_struct *
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next)
{
        struct mm_struct *mm = next->mm;
        struct mm_struct *oldmm = prev->active_mm;

...
        /* Here we just switch the register state and the stack. */
        STAP_MARK_NN(context_switch, prev->pid, next->pid);
        switch_to(prev, next, prev);

        return prev;
}
```

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
  }
  ...
  ```

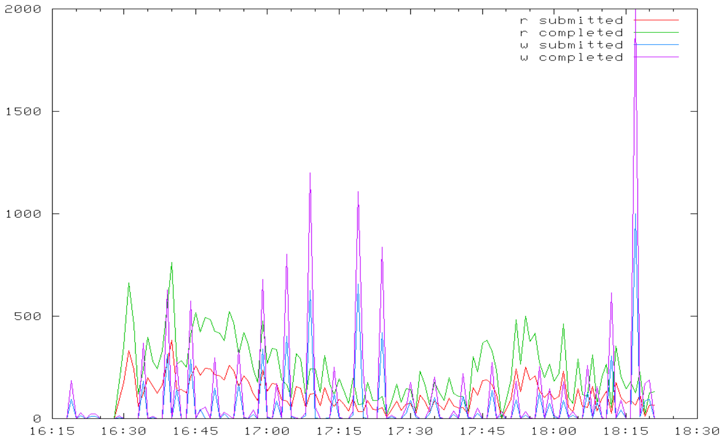
-
  ```
  # stap mark-top.stp
    pid             execname     cycles  (1813 switches)
      0              swapper  764411819
   4473                    X   51465833
   4538       gnome-terminal   33217978
   4745          firefox-bin   24762308
   ...
  ```

Demonstrations

- Let's trace and analyze open(2)
  ```
  int open(const char *pathname, int flags);
  int open(const char *pathname, int flags, mode_t
  mode);
  ```
- Which system calls were executed when you run bash?
- What happens when you run a command?
- Which are the top 10 processes that use `sys_ioctl`?
- Use `plimits.stp` to find out resource limits of processes
- Use `pfiles.stp` to find out opened fd of processes
- Use `udpstat.stp` to find out udp statistics
- Hook the `kbd_event` to add functionalities to it

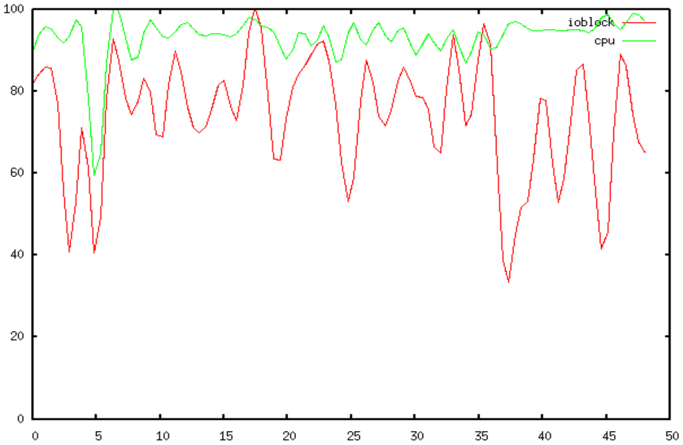
Things that you can write

- **Block I/O submissions & completions**



Things that you can write

- **Is CPU busy now?**



redhat.

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`

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