stapdyn: Porting SystemTap onto Dyninst

Josh Stone & David Smith
Performance Tools @ Red Hat
April 29, 2013
systemtap
stapdyn: Porting SystemTap onto Dyninst

• Motivation for porting to Dyninst
• Overview of SystemTap operation
• Porting all the probe types
• Porting the runtime and tapset
• Wish list for Dyninst
Motivation

- **User Privilege**
  - Attach to one's own processes freely
  - No setuid helper necessary
- **Performance**
  - Run instrumentation directly
- **Stability**
  - We always strive for probe safety, but...
  - Only participating processes are at risk
Anatomy of a SystemTap script

global foo

function total(p, n) {  
    return (foo[p] += n)
}

probe process.function("foo") {  
    t = total(pid(), $var->member)  
    if (t > 1000)  
        printf("%s(%d) total %d\n",  
            execname(), pid(), t)
}
SystemTap runtime modes

**stap**
(user privileges)

- Analyze the user script
  - `foo.stp`
- Generate kernel source
  - `foo.c`
- Compile kernel module
  - `foo.ko`
- Load & Run
  - `foo.ko`

**stapdyn**
(user privileges)

- Generate user source
  - `foo.c`
- Compile user module
  - `foo.so`
- Load & Run
  - `foo.so`

**staprun**
(root privileges)

- Generate kernel source
  - `foo.c`
- Compile kernel module
  - `foo.ko`
- Load & Run
  - `foo.ko`
Contents of a generated module

- Every compiled “foo.so” contains:
  - Metadata describing the probe types and locations
  - Instrumentation entry functions for each probe type
  - Code for the user's functions and probe handlers
  - Runtime code for shared memory, data output, etc.
High-level view of stapdyn

stapdyn (user privileges)

- foo.so
- libdyninstAPI.so

Shared Memory
- globals
- synchronization
- data transport

create/attach

fork/exec

target process

- foo.so
- SHM

child process

- foo.so
- SHM
Probe implementations

Examples of different probe points in **Kernel** and **Dyninst** modes
Probe the beginning and end of everything

begin end error

**Kernel**
- Called directly in-kernel
- Runs when the module loads and unloads

**Dyninst**
- Called directly in stapdyn
- Runs when stapdyn starts and finishes
Probe a specific process address

```
process.function[.call|.inline]
process.statement process.mark
```

### Kernel
- Uses uprobes
  - Inserts a breakpoint
- Runs in the process' context, but transitions to ring-0

### Dyninst
- Direct instrumentation
  - `fileOffsetToAddr()`
  - `insertSnippet()`
- Runs in-process, no ring or context switches at all
Probe when a function returns

process.function.return

**Kernel**
- Uses uretprobes
  - Breakpoint on entry
  - Replaces stack PC with a “trampoline” location
- Runs in the process' context, but transitions to ring-0, twice

**Dyninst**
- Direct instrumentation
  - fileOffsetToAddr()
  - getFunction()
  - findPoint(locExit)
  - insertSnippet()
- Runs in-process, no ring or context switches at all
Probe the beginning and end of a process

process.begin process.end

**Kernel**
- Uses utrace / tracepoints
  - On all forks and execs, it's an end and a begin
  - On exit, it's just an end
- Runs in the process' context, already ring-0 for kernel setup

**Dyninst**
- processCreate()
- processAttach()
- Callbacks for postFork, Exec, and Exit
- RPC oneTimeCode()
Probe the beginning and end of a thread

process.thread.begin process.thread.end

**Kernel**
- Uses utrace / tracepoints
  - A clone() is a begin
  - A sys_exit, is an end
- Runs in the process' context, already ring-0 for kernel setup

**Dyninst**
- Uses threadEvent callbacks
- RPC oneTimeCode()
Probe periodically

timer.{hz,s,ms,us,ns}

Kernel
- Uses hrtimers
- Runs in no specific context

Dyninst
- POSIX timer
  - timer_create()
- Runs directly in staples
Unimplemented probes

Impossible:
- kernel.*
- kprobe.*
- module.*
- netfilter.*
- perf.* (?)

Maybe possible:
- process.syscall
- timer.profile
- procfs.* (similar)
Porting the runtime and tapsets

- Much is shared code, forked for different APIs
  - Userspace APIs are more stable than the kernel's
- Runtime changes locking, shared memory, transport
- Tapset functions also change
  - `cpu()`: `smp_processor_id()` vs. `sched_getcpu()`
- Most tapset probe points don't really translate
  - `scheduler.*` `syscall.*` `vfs.*`

So far, the userspace code is not really Dyninst-specific
Dyninst wishlist

- Cooperation with exception handling
  - Instrumentation confuses the unwinder
- Fuller register access
  - Currently missing 32-bit x86
  - Direct \texttt{pt_regs} would be nice
- Hook system calls (e.g. \texttt{PTRACE SYSCALL})
- Support for ARM and AArch64
- Use elfutils' libdw instead of libdwarf
http://sourceware.org/systemtap
systemtap@sourceware.org
Josh Stone <jistone@redhat.com>
David Smith <dsmith@redhat.com>