Systemtap and new connections: uprobes, dyninst, pcp

Josh Stone
David Smith

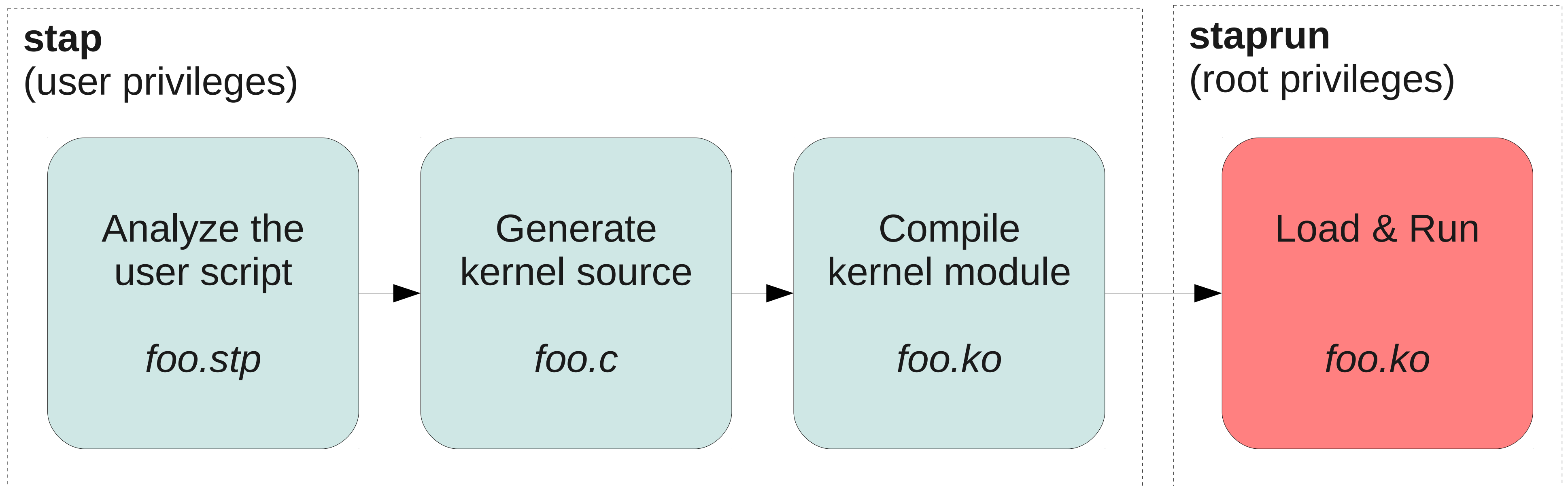
August 30, 2012

Systemtap and uprobes

... see Srikar's previous talk...

Systemtap and Dyninst

SystemTap traditional operation



SystemTap limitations

- Generating kernel modules
  - Kernel doesn't keep a fixed API
  - Inherently out-of-tree, thus politically incorrect

- Running kernel modules
  - Requires privilege to load
    - Root, also groups stapdev and stapusr
  - Mistakes are fatal
    - Mitigated by protected stap language

Introducing Dyninst

- API to insert code into a running program
  - High performance
  - Machine independent
- Examples tools:
  - Open|SpeedShop, TAU, COBI, Extrae, STAT, codeCoverage, unstrip, CRAFT
- Project lead by the University of Maryland and the University of Wisconsin-Madison
- Released under LGPL 2.1+
- http://www.dyninst.org/

Dyninst attractions for SystemTap

- No special privilege required for own processes
- Dynamic operation
  - Run processes directly
  - Attach to live processes
- Insert arbitrary code
  - e.g. Run a SystemTap handler
- Runs instrumentation in-process
  - Not even a ring transition
- Fast!

SystemTap+Dyninst operation

stap
(user privileges)

Analyze the user script

foo.stp

Generate kernel source

foo.c

Compile kernel module

foo.ko

Generate user source

foo.c

Compile user module

foo.so

staprun
(root privileges)

Load & Run

foo.ko

stapdyn
(user privileges)

Load & Run

foo.so

SystemTap+Dyninst limitations

- Limited process visibility
  - Only what's accessible by ptrace
  - No practical system-wide monitoring
- Subset of SystemTap events
  - YES: process.*, timers
  - NO: kernel.*, perf
- Only a single mutator for any given process
  - Thanks ptrace!

Example: Tracing SDT

- SDT = Statically Defined Tracepoints
- dynsdt: standalone dyninst app
  - Discover SDT in target app and libraries
  - Instrument each point with a printf
  - Run and trace!

- Becomes a one-liner in SystemTap:

```
probe process.mark("*") { println($$name) }
```

Integration status

- Basic workflow is complete
  - Generate DSO, loaded with stapdyn and instrument
- Userspace tapsets are available

- TODO
  - Data transport to mutator
  - Split runtime per-thread
  - Combine state across processes

Systemtap and PCP

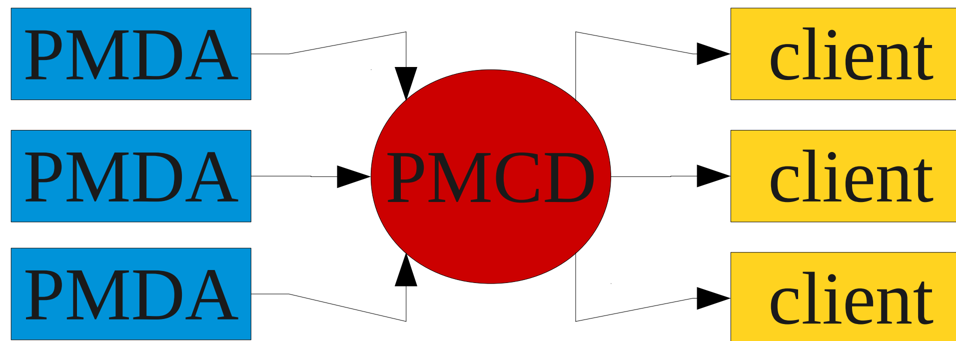
What is PCP (Performance Co-Pilot)?

“Performance Co-Pilot (PCP) provides a framework and services to support system-level performance monitoring and management. It presents a unifying abstraction for all of the performance data in a system, and many tools for interrogating, retrieving and processing that data. “

What is PCP?

- Toolkit for system-level performance analysis
- Open source project
- Extensible (monitors, collectors)
- Live and historical data (one common protocol)
- Distributed
- Multi-OS (Linux, Solaris, Mac OSX, and Windows)
- Rich metadata (dimensioned units, metric documentation)

PCP Basic operation



- PMDAs (*Performance Metrics Domain Agent)* are metric collectors.
- PMCD (*Performance Metric Collector Daemon*) is responsible for answering client requests for metrics, and calls into the various PMDAs to retrieve the requested metrics.
- Clients contact the PMCD (via TCP/IP) and request one or more metrics.

PCP Operation

PMDA
PMDA
PMCD
client

PMDA
PMDA
PMCD
client
client

Further PCP Information

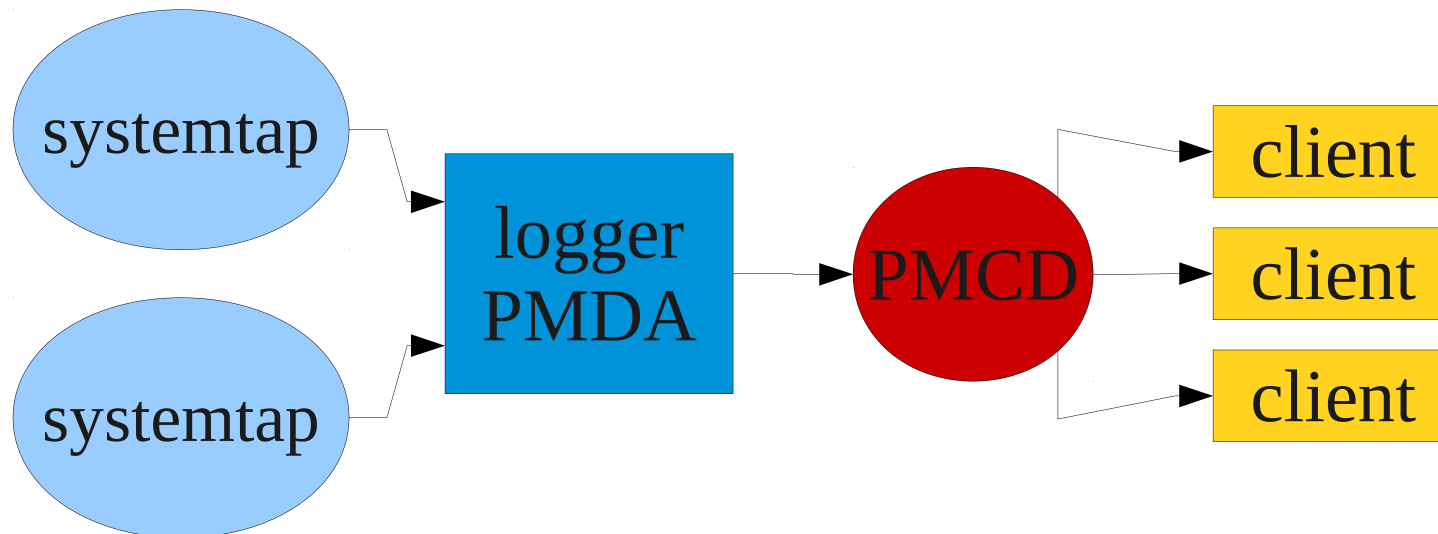
- Web: http://oss.sgi.com/projects/pcp/
- IRC: freenode.net #pcp
- Mail: pcp@oss.sgi.com

Why Systemtap + PCP?

- Over 35 PMDAs (kernel, MySQL, Apache, Sendmail, Postfix, Samba, etc.)
- Supports static metric and dynamic event collection

Systemtap + PCP: Where we are now

The logger PMDA monitors file paths or pipe “paths”. Using pipe “paths”, the logger PMDA can take output from systemtap scripts and make it available to the PMCD via dynamic events.



Systemtap + PCP: Next steps

- Allow PCP clients to be more interactive and “remote-control” PMDAs.
- Web-based PCP front-end client.

Questions?

David Smith <dsmith@redhat.com>

Josh Stone <jistone@redhat.com>