systemtap

An integration platform for
Linux system-wide probing.

Frank Ch. Eigler <fche@redhat.com>
2008-01-28

goal

- Provide debugger-quality visibility into entire running system, ideally any variable at any statement in any thread.
- Do not require recompilation, restart, or embedded instrumentation (though support it if present).
- Run fast, non-intrusively, to minimize probe effect and gross disruption.
- Configure by small domain-specific scripting language, avoid hard-coded assumptions about what information might be interesting.

a taste*

```
probe syscall.* {
    if (execname()==”bash”) log(name . “ “ . argstr))
}
probe kernel.mark(“context_switch”) {
    residency[$arg2, cpu()] ++
}
probe program(“/lib/glibc.so”).function(“malloc”) {
    if ($size > 1024*1024) log(“hog!”)
}
probe timer.profile {
    hotfuncs[execname(), probefunc()] <<< 1
}
global residency, hotfuncs
```

** some of these will be only supported in near future*

++++++ solution scope ------

- Supports need to collect data from places with disparate or nonexistent compiled-in instrumentation.
- Supports need to filter, aggregate, act on data *in situ.*
- Does not currently deal with distributed systems.
- Supports arbitrary textual or binary output, but has no trace browser GUI yet.
- Single-platform multi-architecture Linux free software.

implementation

- Compile phase:
  - Parse probe script, resolve refs to tapset library.
  - Process external information to resolve all probes: DWARF debugging information, marker lists, symbol tables, as needed.
  - Translate to *safety-enhanced* C kernel module.
  - Invoke kernel build system to compile into loadable kernel object. Cache it for script reruns.
- Run phase:
  - Load module. Quickly copy trace buffers to user space. Unload module.

future

- Finish key features previewed above.
- Investigate distributed operation for probe script compilation, launching, and data collection.
- Simplify deployment (prerequisites, user privileges).
- Investigate ways to interoperate with other reporting/visualization tools.
- Take guidance (and assistance!) for distributed operation.

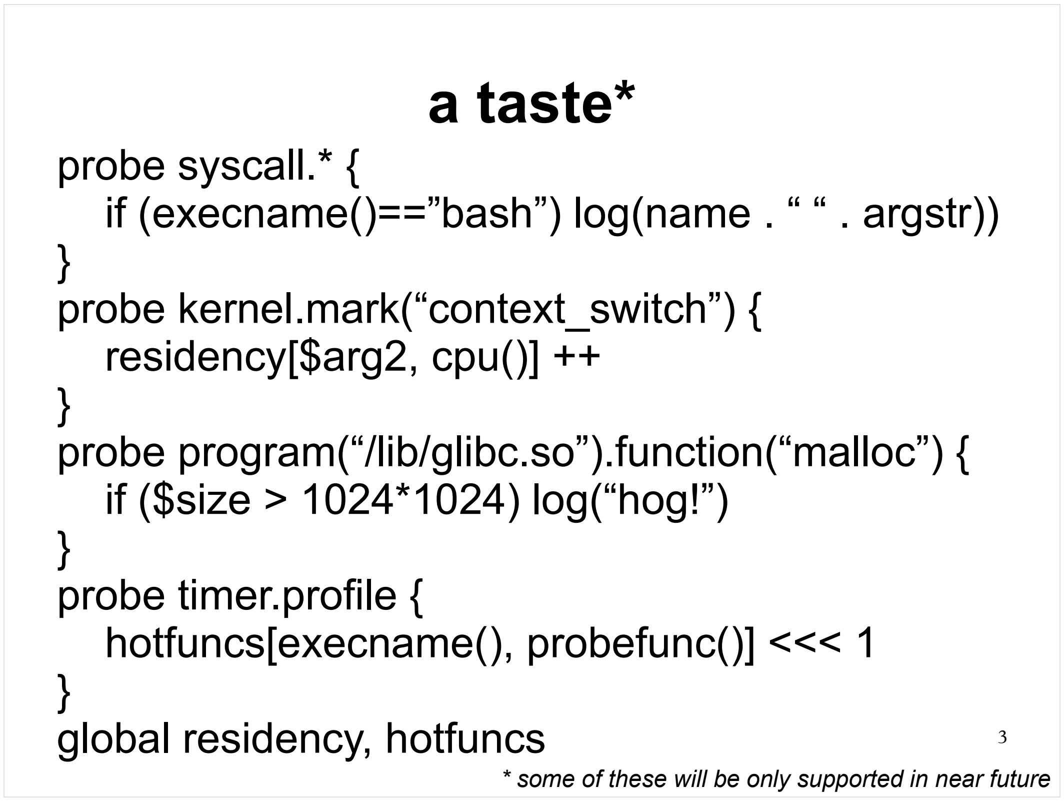
systemtap

An integration platform for
Linux system-wide probing.

Frank Ch. Eigler <fche@redhat.com>
2008-01-28

goal

- Provide debugger-quality visibility into entire running system, ideally any variable at any statement in any thread.
- Do not require recompilation, restart, or embedded instrumentation (though support it if present).
- Run fast, non-intrusively, to minimize probe effect and gross disruption.
- Configure by small domain-specific scripting language, avoid hard-coded assumptions about what information might be interesting.



a taste*

```
probe syscall.* {
   if (execname()==”bash”) log(name . “ “ . argstr))
}
probe kernel.mark(“context_switch”) {
   residency[$arg2, cpu()] ++
}
probe program(“/lib/glibc.so”).function(“malloc”) {
   if ($size > 1024*1024) log(“hog!”)
}
probe timer.profile {
   hotfuncs[execname(), probefunc()] <<< 1
}
global residency, hotfuncs
```

** some of these will be only supported in near future*

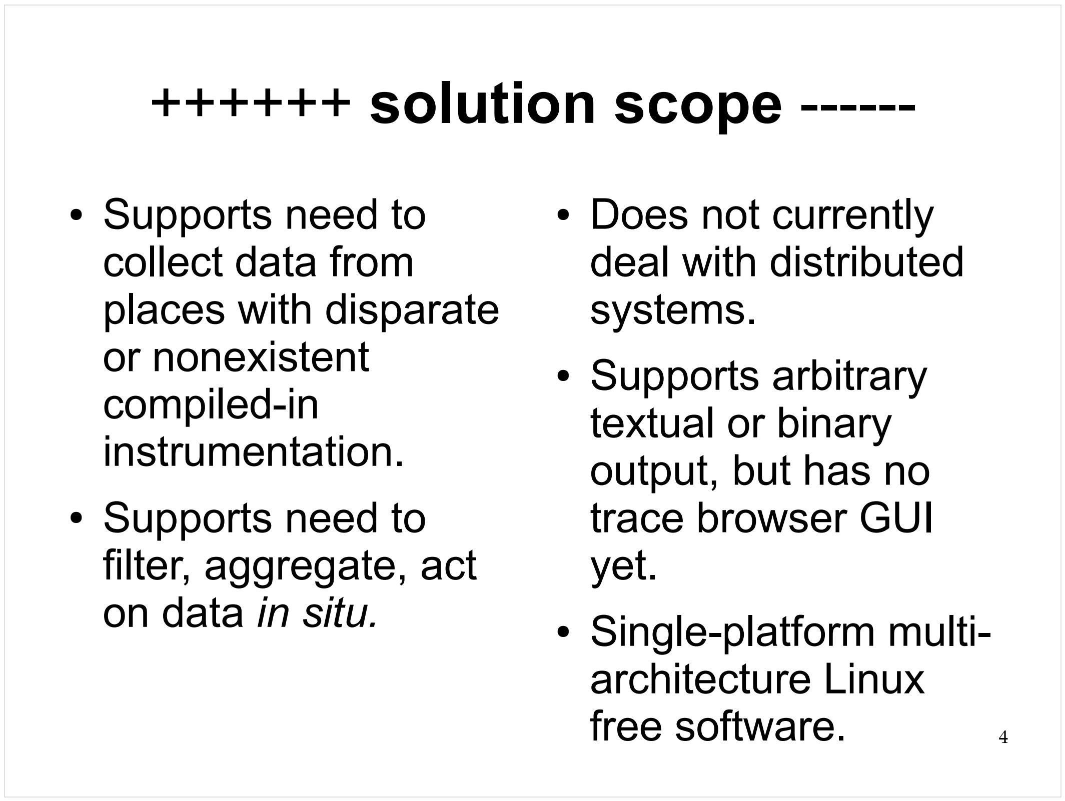
The first probe produces a one-line tracemessage for each system call performed by any bash process in the system. The “name” and “argstr” variables are filled in to a textual representation of the system call name and its argument list. This latter part is done by the “tapset” library – a bunch of scripts shipped with systemtap that define higher level probes and variables from lower level ones.

The second probe attaches to linux kernel markers. Parameters for markers are passed to the script as $arg1, $arg2, etc., with some limited type-conversion (char* -> strings, everything else -> 64-bit numbers). The residency lookup table, indexed by the destination thread-id and the cpu id will collect counts of dispatches of a given thread on a given CPU. This lookup table can be traversed and printed any time.

The third probe demonstrates one flavour of (future) user-space probes. This one would intercept glibc's malloc entry point, across all processes in the system. $size is a parameter to the malloc function, as located on the stack or in the registers. (Its runtime whereabouts are located by search through the glibc debugging data.)

The fourth probe demonstrates elementary profiling. Probefunc() returns a PC-to-function mapping for the current thread (though at the moment it applies only to kernel space). The “<<<” operation constitutes the collection of a statistical sample into an aggregation, from which histograms and other statistics can be trivially extracted and printed. This is SMP-friendly data structure.

The global declarations identify the two arrays as global (persistent for the duration of the script run, shared (with automated locking) amongst all probes. All types are inferred and checked. General control flow (loops, recursion, ...) are supported.



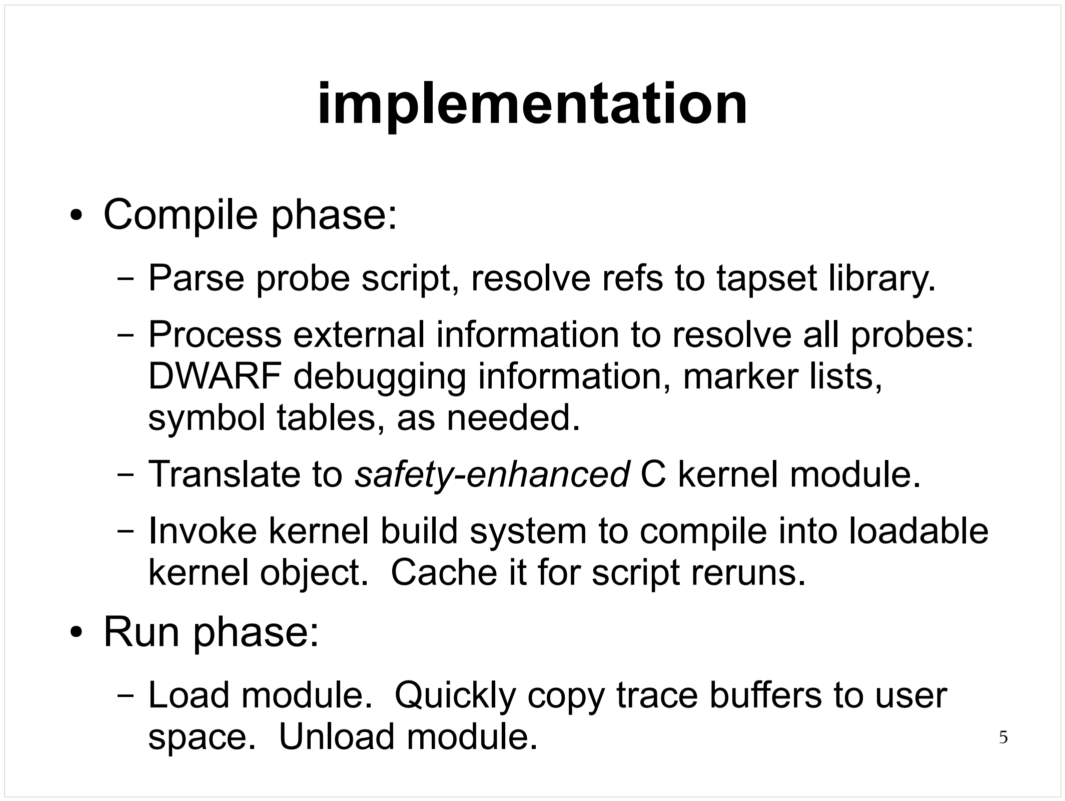
++++++ solution scope ------

- Supports need to collect data from places with disparate or nonexistent compiled-in instrumentation.
- Supports need to filter, aggregate, act on data *in situ.*
- Does not currently deal with distributed systems.
- Supports arbitrary textual or binary output, but has no trace browser GUI yet.
- Single-platform multi-architecture Linux free software.

4

A key point with systemtap is that it is agnostic with respect to where its data comes from. It is not tied to a single instrumentation mechanism – be it kprobes, kernel markers, /proc synthetic files, and other usable hooks. It's not hard to extend to support other instrumentation type APIs.

The other key point is that we view instrumentation as being larger than “just” tracing for later offline analysis. With systemtap, one can process the event data right there to generate statistics. One can compare values to collected statistics and act on changes right at that moment. One can vary the type and quantity of data being gathered as the probing process continues. If the script author wishes, of course systemtap can create gigabytes too. They have a choice.



implementation

- Compile phase:
  - Parse probe script, resolve refs to tapset library.
  - Process external information to resolve all probes: DWARF debugging information, marker lists, symbol tables, as needed.
  - Translate to *safety-enhanced* C kernel module.
  - Invoke kernel build system to compile into loadable kernel object. Cache it for script reruns.
- Run phase:
  - Load module. Quickly copy trace buffers to user space. Unload module.

There exist natural concerns abut this implementation strategy.

The most intuitive one is ... “are you seriously running this in the kernel? Isn't that unsafe?”.

Yes, it could be unsafe, but we work hard to keep it safe. The generated C code is highly stylized and fitted with numerous error and self- checks, locks (with timeouts), and aims to limit both space and time consumption at run time. The generated C code is open for inspection. And it becomes optimized machine code, so overall it runs very fast, even with the checks.

The second question is ... “does a user have to be root?”.

Decreasingly so. We support sudo as well as a group-membership-based setuid method for semi-privileged people to run systemtap scripts. We have a facility to let unprivileged users run only modules that someone else built/wrote. We're working on relaxing this further.

Trace buffers can be kept per-cpu or merged within the kernel. Trace buffers can be disconnected to allow a type of “flight recorder” operation, and can even be pulled out of kernel crash dump images.

future

- Finish key features previewed above.
- Investigate distributed operation for probe script compilation, launching, and data collection.
- Simplify deployment (prerequisites, user privileges).
- Investigate ways to interoperate with other reporting/visualization tools.
- Take guidance (and assistance!) for distributed operation.