Locating System Problems Using Dynamic Instrumentation

Vara Prasad *IBM*
`prasadav@us.ibm.com`

William Cohen *Red Hat, Inc.*
`wcohen@redhat.com`

Frank Ch. Eigler *Red Hat, Inc.*
`fche@redhat.com`

Martin Hunt *Red Hat, Inc.*
`hunt@redhat.com`

Jim Keniston *IBM*
`jkenisto@us.ibm.com`

Brad Chen *Intel Corporation*
`brad.chen@intel.com`

Abstract

Diagnosing complex performance or kernel debugging problems often requires kernel modifications with multiple rebuilds and reboots. This is tedious, time-consuming work that most developers would prefer to minimize.

Systemtap uses the kprobes infrastructure to dynamically instrument the kernel and user applications. Systemtap instrumentation incurs low overhead when enabled, and zero overhead when disabled. SystemTap provides facilities to define instrumentation points in a high-level language, and to aggregate and analyze the instrumentation data. Details of the SystemTap architecture and implementation are presented, along with an example of its application.

1 Introduction

This paper introduces SystemTap, a new performance and kernel troubleshooting infrastructure for Linux. SystemTap provides a scripting environment that can eliminate the modify-build-test loop often required for understanding details of Linux kernel behavior. SystemTap is designed to be sufficiently robust and efficient to support applications in production environments. Our broad goals are to reduce the time and complexity for analyzing problems that involve kernel activity, to greatly expand the community of engineers to which such analyses are available, and to reduce the need to modify and rebuild the kernel as a troubleshooting technique.

Today, identifying functional problems in Linux systems often involves modifying kernel source with diagnostic print statements. The process can be time-consuming and require detailed knowledge of multiple subsystems. SystemTap uses dynamic instrumentation to make this same level of data available without the need to modify kernel source or rebuild the kernel. It delivers this data via a powerful scripting facility. Interesting problem-analysis tools can be implemented as simple scripts.

SystemTap is also designed for analyzing system-wide performance problems. While existing Linux performance tools like `iostat`, `vmstat`, `top`, and `oprofile` are valuable for understanding certain types of performance problems, there are many kinds of problems that they don't readily expose, including:

- Interactions between applications and the operating system
- Interactions between processes
- Interactions between kernel subsystems
- Problems that are obscured by ordinary behavior and require examination of an activity trace

Often these problems are difficult to reproduce in a test environment, making it desirable to have a tool that is sufficiently flexible, robust and efficient to be used in production environments. These scenarios further motivate our work on SystemTap.

SystemTap builds on, and extends, the capabilities of the `kprobes` [6, 7] kernel debugging infrastructure. SystemTap has been influenced by a number of earlier systems, including kerninst [9], Dprobes [6], the Linux Trace Toolkit (LTT) [10], the Linux Kernel State Tracer (LKST) [1], and Solaris DTrace [5, 8].

This paper starts with a brief discussion of the existing dynamic instrumentation provided by Kprobes in the Linux 2.6 kernel, and explains the disadvantages of this approach. Next we describe a few key aspects of the SystemTap design, including the programming environment, the tapset abstraction, and safety in SystemTap. We continue with an example that illustrates the power of SystemTap for troubleshooting performance problems that are difficult to address with existing Linux tools. We close the paper with conclusions and future work.

2 Kprobes

Kprobes, a new feature in the Linux 2.6 kernel, allows for dynamic, in-memory kernel instrumentation. To use kprobes, the developer creates a loadable kernel module with calls into the kprobes interface. These calls specify a kernel instruction address, the *probe point*, and an analysis routine or *probe handler*. Kprobes arranges for control flow to be intercepted by patching the probe point in memory, with control passed to the probe handler. Kprobes has been carefully designed to allow safe insertion and removal of probes and to allow instrumentation of almost any kernel routine. It lets developers add debugging code into a running kernel. Because the instrumentation is dynamic, there is no performance penalty when probes are not used.

The basic control flow interception facility of kprobes has been enhanced with a number of additional facilities. *Jprobes* makes it easy to trace function calls and examine function call parameters. *Kretprobes* is used to intercept function returns and examine return values. Although it is a powerful system for dynamic instrumentation, a number of limitations prevent kprobes from broader use:

- Kprobes does very little safety checking of its probe parameters, making it easy to crash a system through accidental misuse.
- Safe use of kprobes often requires detailed knowledge of the code path to be instrumented. This limits the group of developers who will use kprobes.
- Due to references to kernel addresses and specific kernel symbols, the portability of the instrumentation code using the kprobes interface is poor. This lack of portability also limits re-usability of kprobes-based instrumentation.

- Kprobes does not provide a convenient mechanism to access a function's local variables, except for a jprobe's access to the arguments passed into the function.

- Although using kprobes doesn't require a kernel build-install-reboot, it does require knowledge to build a kernel module and lacks the support library routines for common tasks. This is a significant barrier for potential users. A script-based system that provides the support for common operations and hides the details of building and loading a kernel module will serve a much larger community.

These limitations are part of our motivation for creating SystemTap.

3 SystemTap

SystemTap [2] is being designed and developed to simplify the development of system instrumentation. The SystemTap scripting language allows developers to write custom instrumentation and analysis tools to address the performance problems they are examining. It also improves the reuse of existing instrumentation. Thus, people can build on the expertise of other developers who have already created instrumentation for specific kernel subsystems.

Portability is a concern of SystemTap. The intent is to provide SystemTap on all architectures to which kprobes has been ported.

Safety of the SystemTap instrumentation is another major concern. The tools minimize the chance that the SystemTap instrumentation will cause system crashes or corruption.

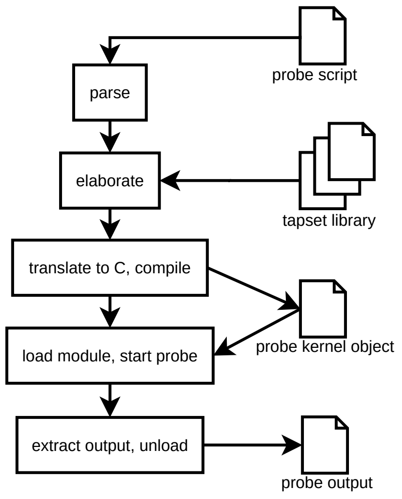

Figure 1: SystemTap processing steps

3.1 SystemTap processing steps

The steps SystemTap uses to convert an instrumentation script into executable instrumentation and extract collected data are shown in Figure 1. SystemTap takes a compilation approach to generate instrumentation code, unlike the interpreter approach other similar systems have taken [6, 5, 8]. A compiler converts the instrumentation script and tapset library into C code for a kernel module. After compilation and linking with the SystemTap runtime, the kernel module is loaded to start the data collection. Data is extracted from module into user-space via reliable and high performance transport. Data collection ends when the module is unloaded from the kernel. The elaboration, translation, and execution steps are described in greater detail in the following subsections.

3.2 Probe language

The SystemTap input consists of a script, written in a simple language described in Section 4. The language describes an association of handler subroutines with probe points. A *probe* *point* may be a particular place in kernel/user code, or a particular event (timers, counters) that may occur at any time. A *handler* is a subroutine that is run whenever the associated probe point is hit.

The SystemTap language is inspired by the UNIX scripting language `awk` [4] and is similar in capabilities to `DTrace's` "D" [5]. It uses a simplified C-like syntax, lacking types, declarations, and most indirection, but adding associative arrays and simplified string processing. The language includes some extensions to interoperate with the target software being instrumented, in order to refer to its data and program state.

3.3 Elaboration

Elaboration is a processing phase that analyzes the input script and resolves references to the kernel or user symbols and *tapsets*. Tapsets are libraries of script or C code used to extend the capability of a basic script, and are described in Section 5. Elaboration resolves external references in the script file to symbolic information and imported script subroutines in preparation for translation to C. In this way, it is analogous to linking an object file with needed libraries.

References to kernel data such as function parameters, local and global variables, functions, and source locations all need to be resolved to actual run-time addresses. This is done by processing the DWARF debugging information emitted by the compiler during the kernel build, as is done in a debugger. All debug data processing occurs prior to execution of the resulting kernel module.

Debugging data contains enough information to locate inlined copies of functions (very common in the Linux kernel), local variables, types, and declarations beyond what are ordinarily exported to kernel modules. It enables placement of probe points in the interior of functions. However, the lack of debug data in some user programs (for example, stripped binaries) will limit SystemTap's ability to place probes in such code.

3.4 Translation

Once a script has been elaborated, it is translated into C.

Each script subroutine is expanded to a block of C that includes necessary locking and safety checks. Looping constructs are augmented with checks to prevent infinite loops. Each variable shared by multiple probes is mapped to an appropriate static declaration, and accesses are protected by locks. To minimize the use of kernel stack space, local variables are placed in a synthetic call frame.

Probe handlers are registered with the kernel using one of the `kprobes` [6, 7] family of registration APIs. For location-type probe points in the kernel, probe points are inserted in kernel memory. For user-level locations, the probe point is inserted in the executable code loaded into user memory while the probe handler is executed in the kernel.

The translated script includes references to a common runtime that provides routines for generic associative arrays, constrained memory management, startup, shutdown, I/O, and other functions.

When translation is complete, the generated C code is compiled and linked with the runtime into a stand-alone kernel module. The final module may be cryptographically signed for safe archiving or remote use.

3.5 Execution

After linking, the SystemTap driver program simply loads the kernel module using `insmod`. The module will initialize itself, insert the probes, then wait for probe points to be hit. When a probe is hit, the associated handler routine is invoked, suspending the thread of execution. When all handlers for that probe point have been executed, the thread of execution resumes. Because thread of execution is suspended, handlers must not block. Probe handlers should hold locks only while manipulating shared SystemTap variables, or as necessary to access previously unlocked target-side data.

The SystemTap script concludes when the user sends an interrupt to the driver program, or when the script calls `exit`. At the end of the run, the module is unloaded and its probes are removed.

3.6 Data Collection and Presentation

Data collected from SystemTap in the kernel must be transmitted to user space. This transport must provide high throughput and low latency, and impose minimal performance impact on the monitored system. Two mechanisms are currently being tested: relayfs and netlink.

Relayfs provides an efficient way to move large blocks of data from the kernel to user space. The data is sent via per-cpu buffers. Relayfs can be compiled into the kernel or built as a loadable module.

Netlink allows a simple stream of data to be sent using the socket APIs. Performance testing suggests that netlink provides less bandwidth than relayfs for transferring large amounts of trace data.

By default, SystemTap output will be processed in batches and written to `stdout` at script exit. The output will also be automatically saved to a file. SystemTap can optionally produce a real-time stream as required by the application.

In user-space, SystemTap can report data as simple text, or in structured computer-parsable forms for consumption by applications such as graphics generators.

4 SystemTap Programming Language

A SystemTap script file is a sequence of top-level constructs, of which there are three types: probe definitions, auxiliary function definitions, and global variable declarations. These may occur in any order, and forward references are permitted.

A probe definition identifies one or more probe points and a body of code to execute when any of them is hit. Multiple probe handlers may execute concurrently on a multiprocessor. Multiple probe definitions may end up referring to the same event or program location: all of them are run in an unspecified sequence when the probe point is hit. For tapset builders, there is also a probe aliasing mechanism discussed in Section 5.1

An auxiliary function is a subroutine for probe handlers and other functions. In order to conserve stack space, Systemtap limits the number of outstanding nested or recursive calls. The translator provides a number of built-in functions, which are implicitly declared.

A global variable declaration lists variables that are shared by all probe handlers and auxiliary functions. (If a variable is not declared global, it is assumed to be local to the function or probe that references it.)

A script may make references to an identifier defined elsewhere in the library of script tapsets. Such a cross-reference causes the entire tapset file providing the definition to be merged into the elaborated script, as if it was simply concatenated. See Section 5 for more information about tapsets.

Fatal errors that occur during script execution cause a cleanup of activity associated with the SystemTap script, and an early abort. Running out of memory, dividing by zero, exceeding an operation count limit and calling too many nested functions are a few types of errors that will terminate a script.

4.1 Probe points

A probe definition specifies one or more probe points in a comma-separated list, and an associated action in the form of a statement block. A trigger of any of the probe points will run the block. Each probe point specification has a "dotted-functor" syntax such as `kernel.function("foo").return`. The core SystemTap translator recognizes a family of these patterns, and tapsets may define new ones. The basic idea of these patterns is to provide a variety of user-friendly ways to refer to program spots of interest, which the translator can map to a kprobe on a particular PC value or an event.

The first group of probe point patterns relates to program points in the kernel and kernel modules. The first element, `kernel` or `module("foo")`, identifies the probe's target software as kernel or a kernel module named `foo.ko`. This first element is used to find the symbolic debug information to resolve the rest of the pattern.

For a probe point defined on a statically known symbol or other program structure, the translator can use debug information to expose local variables within the scopes of the active functions to the script.

4.1.1 Functions

To identify a function, the `function("fn")` element does so by name. If the function is inlineable, all points of inlining are included in the set. The function name may be suffixed by `@filename` or even `@filename:lineno` to identify a source-level scope within which the identifiers should be searched. The function name may include wildcard characters `*` and `?` to refer to all suitable matching names. These may expand to a huge list of matches, and therefore must be used with discretion. The optional element `return` may be added to refer to the moment of each function's return rather than the default `entry`. Below are some sample specifications for function probe points:

`kernel.function("sys_read")
.return`
: A return probe on the named function.

`module("ext3").function("*@fs/
ext3/inode.c")`
: Every function in the named source file, which is part of ext3fs.

4.1.2 Events

Probe points may be defined on abstract events, which are not associated with particular locations in the target program. Therefore, the translator cannot expose much symbolic information about the context of the probe hit to the script. Examples of probes that would fall in this category include probes that perform sampling based on timers or performance monitoring hardware, and probes that watch for changes in a variable's value.

SystemTap defines special events associated with initialization and shutdown of the instrumentation. The special element `begin` triggers a probe handler early during SystemTap initialization, before normal probes are enabled. Similarly, `end` triggers a probe during late shutdown, after all normal probes have been disabled.

4.2 Language Elements

Function and probe handler bodies are written using standard statement/expression syntax that borrows heavily from awk and C. The SystemTap language allows the C, C++, and awk style comments. White space and comments are treated as in C.

SystemTap identifiers have the same syntax as C identifiers, except that `$` is also a legal character. Identifiers are used to name variables and functions. Identifiers that begin with `$` are interpreted as references to variables in the target software, rather than to SystemTap script variables.

The language includes a small number of data types, but no type declarations: a variable's type is inferred from its use. To support this, the translator enforces consistent typing of function arguments and return values, array indexes and values. Similarly, there are no implicit type conversions between strings and numbers.

- Numbers are 64-bit signed integers. Literals can be expressed in decimal, octal, or hexadecimal, using C notation. Type suffixes (e.g., `L` or `U`) are not used.
- Strings. Literals are written as in C. Overall lengths are limited by the runtime system.
- Associative arrays are as in awk. A given array may be indexed by any consistent combination of strings and numbers, and may contain strings, numbers, or statistical objects.
- Statistics. These are special objects that compute aggregations (statistical averages, minima, histograms, etc.) over numbers.

The language has traditional `if-then-else` statements and expressions of C and awk. The language also allows structured control statements such as `for` and `while` loops. Unstructured control flow operations such as labels and `goto` statements are not supported. The translator inserts runtime checks to bound the number of procedure calls and backward branches.

To support associative arrays, the SystemTap language has iterator and `delete` statements. The iterator statement allows the programmer to specify an operation to perform on all the elements in the associative array. The delete operation can remove one or all the elements in the associative array. The associative arrays allow selection of an item by one or more keys. The `in` operation allows the code to determine whether an entry exists in the associative array.

The typical set of arithmetic, bit, assignment, and unary operations in C are available in the SystemTap language, but they operate on 64-bit quantities. The assignment and comparison operations are overloaded for strings.

The SystemTap statistic type allows script writers to keep track of the typical statistics such as minimum, maximum, and average. The `<<<` operator updates a variable storing statistics information as shown in the example below:

```
global avg(s)
probe kernel.syscall("read") {
  process->s <<< $size
}
probe end {
  trace (s)
}
```

SystemTap does not support type casts, address-of operations, or following of arbitrary pointers through structures. However, macro operations will allow access to elements of a particular structure.

4.3 Auxiliary functions

An auxiliary function in SystemTap has essentially the same syntax and semantics as in awk. Specifically, an auxiliary function definition consists of the keyword `function`, a formal argument list and a brace-enclosed statement block. SystemTap deduces the types of the function and its arguments from the expressions that refer to the function. An auxiliary function must always return a value even if it is ignored.

5 Tapsets

When diagnosing systemic problems, one is faced with tracing various subsystems of the operating system and applications. To facilitate such diagnosis, SystemTap includes a library of instrumentation modules for various subsystems known as *tapsets*. The list of available tapsets is published for use in end-user scripts. There are two ways to create tapsets: via the SystemTap scripting language and via the C language.

5.1 Script tapsets

The simplest kind of tapset is one that uses the SystemTap script language to define new probes, auxiliary functions, and global variables, for invocation by an end-user script or another tapset. One can use this mechanism to define commonly useful auxiliary functions like `stp_print()` for special purpose formatting of output data. This facility can also be used to create global variables that can be referenced in the end user scripts as built-in functions. In Figure 2 a `tgid_history` global variable is created that gives a history of the last few scheduled tasks.

In addition, a script tapset can define a *probe alias*. Aliasing is a way of synthesizing a higher level probe from a lower level one. The example tapset shown in Figure 3 defines aliases for the `read` system call, so that a SystemTap user does not have to know the name of the corresponding kernel function.

Aliasing consists of renaming a probe point, and may include some script statements. These statements are all executed *before* the others that are within the user's probe definition (which referenced the alias), as if they were simply transcribed there. This way, they can prepare some useful local variables, or even conditionally reject a probe hit using the `next` statement.

Aliases can also be used to define a new "event" and supply some local variables for use by its handlers as in Figure 4.

An end-user script that uses the probe alias in Figure 4 may look like Figure 5.

5.2 C language tapsets

To allow kernel developers to work in a familar programming language, SystemTap supports a C interface for creating tapsets. A C tapset is a set of data-collection functions for a given subsystem. Data collection functions

```
global tgid_history # the last few tgids scheduled

global _histsize

probe begin {
  _histsize = 10
}

probe kernel.function("context_switch") {
  # rotate array
  for (i=_histsize-1; i>0; i--)
    tgid_history [i] = tgid_history [i-1];
  tgid_history [0] = $prev->tgid;
}
```

Figure 2: SystemTap script using global variable.

```
probe kernel.syscall.read = kernel.function("sys_read")
{ }
```

Figure 3: SystemTap script using probe alias.

```
probe kernel.resource.oom.nonroot =
  kernel.statement("do_page_fault").label("out_of_memory") {
   if ($tsk->uid == 0) next;

   victim_tgid = $tsk->tgid;
   victim_pid = $tsk->pid;
   victim_uid = $tsk->uid;
   victim_fault_addr = $address
}
```

Figure 4: SystemTap script for new out of memory event.

```
probe kernel.resource.oom.nonroot {
  trace ("OOM for pid " . string (victim_pid))
}
```

Figure 5: SystemTap script using out of memory event.

in the tapset are called tapset functions. Tapset functions export data using one or more variables. The C API requires a tapset writer to register each probe point, corresponding data-collection function, and the data exported by the function. When an end-user script refers to the data exported by the corresponding tapset function in the action block, SystemTap calls the associated tapset function in the probe handler. The result is that local variables in the user script are initialized with values from the tapset function.

5.3 System call tapset

SystemTap provides tapsets for various subsystems of the kernel; the system call tapset is an example of one such tapset. As system calls are the primary interface for applications to interact with the kernel, understanding them is a powerful diagnostic tool. The system call tapset provides a probe handler for each system call entry and exit. A system call entry probe gives the values of the arguments to the system call, and the exit probe gives the return value of the system call.

6 Safety

SystemTap is designed for safe use in production systems. One implication is that it should be extremely difficult, if not impossible, to disable or crash a system through use or misuse of SystemTap. Problems like infinite loops, division by zero, and illegal memory references should lead to a graceful failure of a SystemTap script without otherwise disrupting the monitored system. At the same time, we'd like to compile extensions to native machine code, to benefit from the stability of the existing tool chain, minimize new kernel code, and approach native performance.

Our basic approach to safety is to design a safe scripting language, with some safety properties supported by runtime checks. Table 1 provides some details of our basic approach. SystemTap compiles the script file into native code and links it with the SystemTap runtime library to create a loadable kernel module. Version and symbol name checks are applied by `insmod`. The elaborator generates instrumentation code that gracefully terminates loops and recursion, if they run beyond a configurable threshold. We avoid privileged and illegal kernel instructions by excluding constructs in the script language for inlined assembler, and by using compiler options used for building kernel modules.

SystemTap incorporates several additional design features that enhance safety. Explicit dynamic memory allocation by scripts is not allowed, and dynamic memory allocation by the runtime is avoided. SystemTap can frequently use explicitly synthesized frames in static memory for local variables, avoiding usage of kernel stack. Language and runtime systems ensure that SystemTap-generated code for probe handlers is strictly terminating and non-blocking.

SystemTap safety requires controlling access to kernel memory. Kernel code cannot be invoked directly from a SystemTap script. SystemTap language features make it impossible to express kernel data writes or to store a pointer to kernel data. Additionally, a modified trap handler is used to safely handle invalid memory references. SystemTap supports a "guru" mode where certain of these constraints can be removed (e.g., in a tapset), allowing a tradeoff between safety and kernel debugging requirements.

6.1 Safety Enhancements

A number of options are planned that extend the safety and flexibility of SystemTap to match

	language design	translator	insmod checks	runtime checks	memory portal	static validator
infinite loops		x		o		o
recursion		x		o		o
division by zero		x		o		o
resource constraints		x		x		
locking constraints		x		x		
array bounds errors	x	x		x		o
invalid pointers		o		o		o
heap memory bugs	x					o
illegal instructions	x				o	
privileged instructions	x				o	
memory r/w restrictions	x			x	o	o
memory execute restrictions	x			x	o	o
version alignment		o	x			
end-to-end safety					x	x
safety policy specification facility					x	

Table 1: SystemTap safety mechanisms. An “x” indicates that an aspect of the implementation (columns) is used to implement a particular safety feature (rows). An “o” indicates optional functionality.

and exceed that of other systems. A memory and code "portal" directs references to kernel memory outside the loadable module through a special-purpose interpeter or "portal." This provides a single point of control for related safety issues, and facilitates a desireable separation of safety policy from mechanism. Trivial policies would support "guru mode" (no restrictions) and default mode (read restrictions to I/O memory, restricted write and code access). Other simple policies expand access incrementally, for example, allowing external calls to an explicit list of kernel subroutines. Eventually, the policy could be extended to support security goals such as secure non-root execution and restricting memory access based on user credentials.

An optional static analyzer examines a disassembled kernel module and confirms that it satisfies certain safety properties. Simple checks include disallowing privileged instructions, locking primitives and instructions that are illegal in kernel mode. In the future, more elaborate checks may be included to confirm that loop counters, memory portals and other safety features are used.

6.2 Comparision to Other Systems

Solaris DTrace includes a number of unusual features intended to enhance the safety and security of the system. These features include a very restricted scripting language and the scripts being interpreted rather than compiled.

DTrace's D language does not support procedure declarations or a general purpose looping construct. This avoids a number of safety issues in scripts including infinite loops and infinite recursion.

Because D scripts are interpreted rather than executed directly, it is impossible for them to include illegal or privileged instructions or to invoke code outside of the DTrace execution environment. The interpreter can also catch invalid pointer dereferences, division by zero, and other run-time errors.

SystemTap will support kernel debugging features in guru mode that DTrace does not, including the ability to write arbitrary locations in kernel memory and the ability to invoke arbitrary kernel subroutines.

Because the language infrastructure used by SystemTap is common to all C programs, it tends to be better tested and more robust than the special-purpose interpreter used by DTrace.

The embedding of an interpreter in the Solaris kernel represents significant additional kernel functionality. This introduces an increased risk of kernel bugs that could lead to security or reliability issues.

Dprobes and Dtrace have many safety features in common. Both use an interpreted language. Like SystemTap, both use a modified kernel trap-handler to capture illegal memory references. Like kprobes, dprobes is intended for use primarily by kernel developers. Consequently, it exposes the kprobes layer in such a way that it is not crashproof. SystemTap seeks to address these safety issues.

6.3 Security

It is important that SystemTap can be used without significantly impacting the overall security of the system. Given that SystemTap is only available to privileged users, our initial security concerns are that the system be crashproof by design, and that its implementation is of sufficient quality and simplicity to protect users from unintentional lapses. A specific concern is the security of the communication layer;

that the kernel-to-user transport is secured from non-privileged users.

Future versions of SystemTap may provide features that support secure use of SystemTap by non-privileged users. Specific features that might be required include:

- Protection of kernel memory based on user credentials.
- Protection of kernel-to-user transport based on user credentials.
- Recognition of a restricted subset of the SystemTap language that is permissible for non-privileged users.

A security scheme based on a virtual machine monitor such as Xen [3] might provide a simpler and general solution to secure SystemTap use by non-privileged users.

7 Example SystemTap Script

The SystemTap scripting language lends itself to writing compact instrumentation. The following example demonstrates a simple script to collect information. On SMP machines, the interprocessor interrupt is an expensive operation. One can find how many interprocessor interrupts are performed on an SMP machine by examining the `LOC:` entry of `/proc/interrupts`. However, this entry does not give a complete picture of what is causing the interprocessor interrupts.

A developer would like to know the process (PID), the process name, and the backtrace to get a better context of what is triggering the interprocessor interrupts. Figure 6 shows the SystemTap script used to accumulate that information into an associative array. Each time `smp_call_function` is called, the appropriate associative array entry is incremented. The `$pid` provides the process id number, the `$pname` provides the name of the process, and `stack()` the back trace in the kernel. This data is recorded in an associative array `traces`. When the data collection is over and the instrumentation is removed, the "end probe" prints out information.

Figure 7 shows the beginning of the data generated from a dual processor x86-64 machine when a DVD has just been loaded on the machine. From the samples listed below, we see that process 4010, hald, has caused a number of interprocessor interrupts. With the stack backtrace as part of the hash key, we can see that the first entry has to do with the disk change in the CDROM drive, and the second entry is caused by `sys_close`.

8 Conclusions and Future Work

We have described current dynamic instrumentation facilities in the Linux kernel and the need for improvements. These motivate the SystemTap architecture and salient features of its scripting language. We described the tapset library and its importance in SystemTap. Safety is a very important consideration of SystemTap design and we described how safety considerations impacted our SystemTap design. We presented an example of how SystemTap is used to gather interesting data to diagnose a problem. The Systemtap project is still in development. In our continuing work, we plan to implement tapset libraries for various kernel subsystems, and expand SystemTap to trace user-level activity.

```
global traces

probe kernel.function("smp_call_function") {
  traces[$pid, $pname, stack()] += 1;
}

probe end {
  print(traces);
}
```

Figure 6: SystemTap script to collect interprocessor interrupt information.

```
root# stp scf.stp
Press Control-C to stop.
All kprobes removed
traces[4010, hald, trace for 4010 (hald)
0xffffffff8011a551 : smp_call_function+0x1/0x70
0xffffffff80182c0c : invalidate_bdev+0x1c/0x40
0xffffffff8019bc48 : __invalidate_device+0x58/0x70
0xffffffff80188f89 : check_disk_change+0x39/0xa0
0xffffffff80133c90 : default_wake_function+0x0/0x10
0xffffffff802abeef : cdrom_open+0xa0f/0xa60
0xffffffff80133c90 : default_wake_function+0x0/0x10
0xffffffff80132650 : finish_task_switch+0x40/0x90
0xffffffff80346bb9 : thread_return+0x54/0x8b
0xffffffff801419cd : __mod_timer+0x13d/0x150
] = 18
traces[4010, hald, trace for 4010 (hald)
0xffffffff8011a551 : smp_call_function+0x1/0x70
0xffffffff80182c0c : invalidate_bdev+0x1c/0x40
0xffffffff8018856e : kill_bdev+0xe/0x30
0xffffffff801890d6 : blkdev_put+0x76/0x1c0
0xffffffff80181eb2 : __fput+0x72/0x160
0xffffffff801806de : filp_close+0x7e/0xa0
0xffffffff80180793 : sys_close+0x93/0xc0
0xffffffff8010e51a : system_call+0x7e/0x83
] = 27
```

...

Figure 7: Run of SMP call instrumentation.

9 Acknowledgements

We would like to express our thanks to Ananth N. Mavinakayanahalli, Hien Q. Nguyen, Prasanna S. Panchamukhi, and Thomas Zanussi for their valuable contributions to the project. The authors are indebted to Ulrich Drepper and Roland McGrath for their help and advice in the project. Thanks are in order to Suparna Bhattacharya and Richard Moore for sharing their knowledge of Kprobes and Dprobes. We would also like to thank Rohit Seth, Rusty Lynch, and Anil Keshavamurthy for Linux kernel and Itanium expertise. Special thanks to Doug Armstrong and Victoria Gromova for their input on features for parallel program analysis. Thanks to K. Sridharan and Charles Spirakis for studying support of common profiling tasks.

10 Trademarks and Disclaimer

References

[1] Linux kernel state tracer, May 2005. `http://lkst.sourceforge.net/`.

[2] Systemtap, May 2005. `http://sourceware.org/systemtap/`.

[3] The xen virtual machine monitor, May 2005. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/`.

[4] Alfred V. Aho, Brian K. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[5] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Levinthal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Technical Conference*, pages 15–28, June 2004.

[6] Richard J. Moore. A universal dynamic trace for Linux and other operating systems. In *FREENIX*, 2001.

[7] Prasanna S. Panchamukhi. Kernel debugging with kprobes: Insert printk's into linux kernel on the fly, Aug 2004. `http://www-106.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnx%w07Kprobe`.

[8] Sun Microsystems, Santa Clara, California. *Solaris Dynamic Tracing Guide*, 2004.

[9] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[10] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *In Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.