Dynamic Instrumentation of Production Systems

Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal
Solaris Kernel Development
Sun Microsystems
{bmc, mws, ahl}@eng.sun.com

Abstract

This paper presents DTrace, a new facility for dynamic instrumentation of production systems. DTrace features the ability to dynamically instrument both user-level and kernel-level software in a unified and absolutely safe fashion. When not explicitly enabled, DTrace has zero probe effect — the system operates exactly as if DTrace were not present at all. DTrace allows for many tens of thousands of instrumentation points, with even the smallest of systems offering on the order of 30,000 such points in the kernel alone. We have developed a C-like high-level control language to describe the predicates and actions at a given point of instrumentation. The language features user-defined variables, including thread-local variables and associative arrays. To eliminate the need for most postprocessing, the facility features a scalable mechanism for aggregating data and a mechanism for speculative tracing. DTrace has been integrated into the Solaris operating system and has been used to find serious systemic performance problems on production systems — problems that could not be found using preexisting facilities.

1 Introduction

As systems grow larger and more complicated, performance analysis is increasingly performed by the system integrator in production rather than by the developer in development. Trends towards componentization and application consolidation accelerate this change: system integrators increasingly combine off-the-shelf components in ways that the original developers did not anticipate. Performance analysis infrastructure has generally not kept pace with the shift to in-production performance analysis: the analysis infrastructure is still focussed on the developer, on development systems, or both. And where performance analysis infrastructure *is* designed for production use, it is almost always *process-centric* — and therefore of little help in understanding systemic problems.

To be acceptable for use on production systems, performance analysis infrastructure must have zero probe effect when disabled, and must be absolutely safe when enabled. That is, its mere presence must not make the system any slower, and there must be no way to accidentally induce system failure through misuse. To have systemic scope, the entire system must be instrumentable, and there must exist ways to easily coalesce data to highlight systemic trends.

We have developed a facility for systemic dynamic instrumentation that can gather and coalesce arbitrary data on production systems. This facility — DTrace — has been integrated into Solaris and is publicly available[9]. DTrace features:

- Dynamic instrumentation. Static instrumentation always induces some disabled probe effect; to achieve the zero disabled probe effect required for production use, DTrace uses *only* dynamic instrumentation. When DTrace is not in use, the system is just as if DTrace were not present at all.
- Unified instrumentation. DTrace can dynamically instrument both user *and* kernel-level software, and can do so in a *unified* manner whereby both data and control flow can be followed across the user/kernel boundary.
- Arbitrary-context kernel instrumentation. DTrace can instrument virtually all of the kernel, including delicate subsystems like the scheduler and synchronization facilities.
- Data integrity. DTrace always reports any errors that prevent trace data from being recorded. In the absence of such errors, DTrace *guarantees* data integrity: there are no windows in which recorded data can be silently corrupted or lost.

- Arbitrary actions. The actions taken at a given point of instrumentation are not defined or limited *a priori* — the user can enable any probe with an arbitrary set of actions. Moreover, DTrace guarantees absolute safety of user-defined actions: run-time errors such as illegal memory accesses are caught and reported.

- Predicates. A logical predicate mechanism allows actions to be taken only when user-specified conditions are met, thereby pruning unwanted data *at the source*. DTrace thus avoids retaining, copying and storing data that will ultimately be discarded.

- A high-level control language. Predicates and actions are described in a C-like language — dubbed "D" — that supports all ANSI C operators and allows access to the kernel's variables and native types. D offers user-defined variables, including global variables, thread-local variables, and associative arrays. D also supports pointer dereferencing; coupled with the run-time safety mechanisms of DTrace, structure chains can be safely traversed in a predicate or action.

- A scalable mechanism for aggregating data. DTrace allows data to be aggregated based on an arbitrary tuple of D expressions. The mechanism coalesces data as it is generated, reducing the amount of data that percolates through the framework by a factor of the number of data points. By allowing aggregation based on D expressions, DTrace permits users to aggregate by virtually anything.

- Speculative tracing. DTrace has a mechanism for speculatively tracing data, deferring the decision to commit or discard the data to a later time. This feature eliminates the need for most post-processing when exploring sporadic aberrant behavior.

- Heterogeneous instrumentation. Tracing frameworks have historically been designed around a single instrumentation methodology. In DTrace, the instrumentation providers are formally separated from the probe processing framework by a well-specified API, allowing novel dynamic instrumentation technologies to plug into and exploit the common framework.

- Scalable architecture. DTrace allows for many tens of thousands of instrumentation points (even the smallest systems typically have on the order of 30,000 such points) and provides primitives for subsets of probes to be efficiently selected and enabled.

- Virtualized consumers. Everything about DTrace is virtualized per consumer: multiple consumers can enable the same probe in different ways, and a single consumer can enable a single probe in different ways. There is no limit on the number of concurrent DTrace consumers.

The remainder of this paper describes DTrace in detail. In Section 2, we discuss related work in the area of dynamic instrumentation. Section 3 provides an overview of the DTrace architecture. Section 4 describes some of the instrumentation providers we have implemented for DTrace. Section 5 describes the D language. Section 6 describes the DTrace facility for aggregating data. Section 7 describes the user-level instrumentation provided by DTrace. Section 8 describes the DTrace facility for speculative tracing. Section 9 describes in detail a performance problem on a production system that was root-caused using DTrace. Finally, Section 10 discusses future work and Section 11 provides our conclusions.

2 Related work

The notion of safely augmenting operating system execution with user-specified code has been explored in extensible systems like VINO[8] and SPIN[2]. However, these systems were designed to allow the user to *extend* the system where DTrace is designed to allow the user to simply *understand* it. So where the extensible systems allow much more general purpose augmentation, they have many fewer (if any) primitives for understanding system behavior.

There is a large body of work dedicated to systemic and dynamic instrumentation. Some features of DTrace, like predicates, were directly inspired by other work[6]. Some other features, like the idea of a higher-level language for system monitoring, exist elsewhere[1, 3, 7] — but DTrace has made important new contributions. Other features, like aggregations, exist only in rudimentary form elsewhere[1, 3]; DTrace has advanced these ideas significantly. And some features, like speculative tracing, don't seem to exist in any form in any of the prior work.

2.1 Linux Trace Toolkit

The Linux Trace Toolkit (LTT) is designed around a traditional static instrumentation methodology that induces a non-zero (but small) probe effect for each instrumentation point[13]. To keep the overall disabled probe effect reasonably low, LTT defines only a limited number of instrumentation points — comprising approximately 45 events. LTT cannot take arbitrary actions (each statically-defined event defines an event-specific "detail"), and lacks any sort of higher-level language to describe such actions. LTT has a coarse mechanism for pruning data, whereby traced events may be limited only to those pertaining to a given PID, process group, GID or UID, but no other predicates are possible. As LTT has few mechanisms for reducing the data flow via pruning or coalescing, substantial effort has naturally gone into optimizing the path of trace data from the kernel to user-level[14].

2.2 DProbes

DProbes is a facility originally designed for OS/2 that was ported to Linux and subsequently expanded[7]. Superficially, DProbes and DTrace have some similar attributes: both are based on dynamic instrumentation (and thus both have zero probe effect when not enabled) and both define a language for arbitrary actions as well as a simple virtual machine to implement them. However, there are significant differences. While DProbes uses dynamic instrumentation, it uses a technique that is lossy when a probe is hit simultaneously on different CPUs. While DProbes has user-defined variables, it lacks thread-local variables and associative arrays. Further, it lacks any mechanism for data aggregation, and has no predicate support. And while DProbes has made some safety considerations (for example, invalid loads are handled through an exception mechanism), is was not designed with absolute safety as a constraint; misuse of DProbes can result in a system crash.[1]

2.3 K42

K42 is a research kernel that has its own static instrumentation framework[11]. K42's instrumentation has many of LTT's limitations (statically defined actions, no facilities for data reduction, etc.), but — as in DTrace — thought has been given in K42 to instrumentation scalability. Like DTrace, K42 has lock-free, per-CPU buffering — but K42 implements it in a way that sacrifices the integrity of traced data.[2] Recently, the scalable tracing techniques from K42 have been integrated into LTT, presumably rectifying LTT's serious scalability problems (albeit at the expense of data integrity).

2.4 Kerninst

Kerninst is a dynamic instrumentation framework that is designed for use on commodity operating system kernels[10]. Kerninst achieves zero probe effect when disabled, and allows instrumentation of virtually any text in the kernel. However, Kerninst is highly aggressive in its instrumentation; users can erroneously induce a fatal error by accidentally instrumenting routines that are not actually safe to instrument.[3] Kerninst allows for some coalesence of data, but data may not be aggregated based on arbitrary tuples. Kerninst has some predicate support, but it does not allow for arbitrary predicates and has no support for arbitrary actions.

3 DTrace Architecture

The core of DTrace — including all instrumentation, probe processing and buffering — resides in the kernel. Processes become DTrace *consumers* by initiating communication with the in-kernel DTrace component via the DTrace library. While any program may be a DTrace consumer, `dtrace`(1M) is the canonical DTrace consumer: it allows generalized access to all DTrace facilities.

3.1 Providers and Probes

The DTrace framework itself performs no instrumentation of the system; that task is delegated to instrumentation *providers*. Providers are loadable kernel modules that communicate with the DTrace kernel module using a well-defined API. When they are instructed to do so by the DTrace framework, instrumentation providers determine points that they can potentially instrument. For every point of instrumentation, providers call back into the DTrace

[1] Examples of such misuse include erroneously specifying a non-instruction boundary to instrument or specifying an action that incorrectly changes register values.

[2] For example, rescheduling during data recording can silently corrupt the data buffer.

[3] In particular, Kerninst on SPARC makes no attempt to recognize text as being executed at TL=1 or TL>1 — two highly constrained contexts in the SPARC V9 architecture. Instrumenting such text with Kerninst induces an operating system panic. This has been communicated to Miller et al.; a solution is likely forthcoming[5].

framework to create a *probe*. To create a probe the provider specifies the module name and function name of the instrumentation point, plus a semantic name for the probe. Each probe is thus uniquely identified by a 4-tuple:

$$< provider, module, function, name >$$

Probe creation does *not* instrument the system: it simply identifies a potential for instrumentation to the DTrace framework. When a provider creates a probe, DTrace returns a *probe identifier* to the provider.

Probes are advertised to consumers, who can enable them by specifying any (or all) elements of the 4-tuple. When a probe is enabled, an *enabling control block* (ECB) is created and associated with the probe. If there are no other ECBs associated with the probe (that is, if the probe is disabled), the DTrace framework calls the probe's provider to enable the probe. The provider dynamically instruments the system in such a way that when the probe fires, control is transferred to an entry point in the DTrace framework with the probe's identifier specified as the first argument. A key attribute of DTrace is that there are no constraints as to the context of a firing probe: the DTrace framework itself is non-blocking and makes no explicit or implicit calls into the kernel at-large.

When a probe fires and control is transferred to the DTrace framework, interrupts are disabled on the current CPU, and DTrace performs the activities specified by each ECB on the probe's ECB chain. Interrupts are then reenabled and control returns to the provider. The provider itself need not handle any multiplexing of consumers on a single probe — all multiplexing is handled by the framework's ECB abstraction.

3.2 Actions and Predicates

Each ECB may have an optional predicate associated with it. If an ECB has a predicate and the condition specified by the predicate is not satisfied, processing advances to the next ECB. Every ECB has a list of actions; if the predicate is satisfied, the ECB is processed by iterating over its actions. If an action indicates data to be traced, the data is stored in the per-CPU buffer associated with the consumer that created the ECB; see Section 3.3. Actions may also update D variable state; user variables are described in more detail in Section 5. Actions may *not* store to kernel memory, modify registers, or make otherwise arbitrary changes to system state.[4]

3.3 Buffers

Each DTrace consumer has a set of in-kernel per-CPU buffers allocated on its behalf and referred to by its consumer state. The consumer state is in turn referred to by each of the consumer's ECBs; when an ECB action indicates data to be traced, it is recorded in the ECB consumer's per-CPU buffer. The amount of data traced by a given ECB is always *constant*. That is, different ECBs may trace different amounts of data, but a given ECB always traces the same quantity of data. Before processing an ECB, the per-CPU buffer is checked for sufficient space; if there is not sufficient space for the ECB's data recording actions, a per-buffer *drop count* is incremented and processing advances to the next ECB.

It is up to consumers to minimize drop counts by reading buffers periodically.[5] Buffers are read out of the kernel using a mechanism that both maintains data integrity and assures that probe processing remains wait-free. This is done by having two per-CPU buffers: an active buffer and an inactive buffer. When a DTrace consumer wishes to read the buffer for a specified CPU, a cross-call is made to the CPU. The cross-call, which executes on the specified CPU, disables interrupts on the CPU, switches the active buffer with the inactive buffer, reenables interrupts and returns. Because interrupts are disabled in both probe processing and buffer switching (and because buffer switching always occurs on the CPU to be switched), an ordering is assured: buffer switching and probe processing cannot possibly interleave on the same CPU. Once the active and inactive buffers have been switched, the inactive buffer is copied out to the consumer.

The data record layout in the per-CPU buffer is an *enabled probe identifier* (EPID) followed by some amount of data. An EPID has a one-to-one mapping with an ECB, and can be used to query the kernel for the size and layout of the data stored by the corresponding ECB. Because the data layout for a given ECB is guaranteed to be constant over the lifetime of the ECB, the ECB metadata can be cached at user-level. This design separates the metadata

[4]There do exist some actions that change the state of the system, but they change state only in a well-defined way (e.g. stopping the current process, inducing a kernel breakpoint). These destructive actions are only permitted to users with sufficient privilege, and can be disabled entirely.

[5]Consumers may also reduce drops by increasing the size of in-kernel buffers.

stream from the data stream, simplifying run-time analysis tools considerably.

3.4 DIF

Actions and predicates are specified in a virtual machine instruction set that is emulated in the kernel at probe firing time. The instruction set, "D Intermediate Format" or DIF, is a small RISC instruction set designed for simple emulation and on-the-fly code generation. It features 64-bit registers, 64-bit arithmetic and logical instructions, comparison and branch instructions, 1-, 2-, 4- and 8-byte memory loads from kernel and user space, and special instructions to access variables and strings. DIF is designed for simplicity of emulation. For example, there is only one addressing mode and most instructions operate only on register operands.

3.5 DIF Safety

As DIF is emulated in the context of a firing probe, it is a design constraint that DIF emulation be absolutely safe. To assure basic sanity, opcodes, reserved bits, registers, string references and variable references are checked for validity as the DIF is loaded into the kernel. To prevent DIF from inducing an infinite loop in probe context, only *forward* branches are permitted. This safety provision may seem draconian — it eliminates loops altogether — but in practice we have not discovered it to present a serious limitation.[6]

Run-time errors like illegal loads or division by zero cannot be detected statically; these errors are handled by the DIF virtual machine. Misaligned loads and division by zero are easily handled — the emulator simply refuses to perform such operations. (Any attempt to perform such an operation aborts processing of the current ECB and results in a run-time error that is propagated back to the DTrace consumer.) Similarly, loads from memory-mapped I/O devices (where loads may have undesirable or dangerous side effects) are prevented by checking that the address of a DIF-directed load does not fall within the virtual address range that the kernel reserves for memory-mapped device registers.

Loads from unmapped memory are more complicated to prevent, however, because it is not possible to probe VM data structures from probe firing context. When the emulation engine attempts to perform such a load, a hardware fault will occur. The kernel's page fault handler has been modified to check if the load is DIF-directed; if it is, the fault handler sets a per-CPU bit to indicate that a fault has occurred, and increments the instruction pointer past the faulting load. After emulating each load, the DIF emulation engine checks for the presence of the faulted bit; if it is set, processing of the current ECB is aborted and the error is reported to the user. This mechanism adds some processing cost to the kernel's page fault path, but the cost is so extraordinarily small relative to the total processing cost of a page fault that the effect on system performance is nil.

[6] DProbes addressed this problem by allowing loops but introducing a user-tunable, "`jmpmax`," as an upper-bound on the number of jumps that a probe handler may make.

4 Providers

By formally separating instrumentation providers from the core framework, DTrace is able to accommodate heterogeneous instrumentation methodologies. Further, as future instrumentation methodologies are developed, they can be easily plugged in to the DTrace framework. We have implemented six different instrumentation providers, each with its own dynamic instrumentation methodology, that offer observability into different aspects of the system. While the providers differ in their methodology, *all* of the DTrace providers have zero probe effect when disabled. Some of the providers are introduced below, but the details of their instrumentation methodologies are largely beyond the scope of this paper.

4.1 Function Boundary Tracing

The Function Boundary Tracing (FBT) provider makes available a probe upon entry to and return from nearly every function in the kernel. As there are many functions in the kernel, FBT provides many probes — even on the smallest systems, FBT will provide more than 25,000 probes. As with other DTrace providers, FBT has zero probe effect when it is not explicitly enabled, and when enabled only induces a probe effect in probed functions. While the mechanism used for the implementation of FBT is highly specific to the instruction set architecture, FBT has been implemented on both SPARC and x86.

On SPARC, FBT works by replacing an instruction with an unconditional annulled branch-always (`ba,a`) instruction. The branch redirects control flow into an FBT-controlled trampoline, which prepares arguments and transfers control into DTrace.

Upon return from DTrace, the replaced instruction is executed in the trampoline before transferring control back to the instrumented code path. This is a similar mechanism to that used by Kerninst[10] — but it is at once less general (it instruments only function entry and return) and completely safe (it will never erroneously instrument code executed at TL>0).

On x86, FBT uses a trap-based mechanism that replaces one of the instructions in the sequence that establishes a stack frame (or one of the instructions in the sequence that dismantles a stack frame) with an instruction to transfer control to the interrupt descriptor table (IDT). The IDT handler uses the trapping instruction pointer to look up the FBT probe and transfers control into DTrace. Upon return from DTrace, the replaced instruction is *emulated* from the trap handler by manipulating the trap stack. The use of emulation (instead of instruction rewriting and reexecution) assures that FBT does not suffer from the potential lossiness of the DProbes mechanism.

4.2 System Call Tracing

The `syscall` provider makes available a probe at the entry to and return from each system call in the system. As system calls are the primary interface between user-level applications and the operating system kernel, the `syscall` provider can offer tremendous insight into application behavior with respect to the system. The `syscall` provider works by dynamically rewriting the corresponding entry in the system call table when a probe is enabled.

4.3 Lock Tracing

The `lockstat` provider makes available probes that can be used to obtain kernel lock contention statistics, or to understand virtually any aspect of kernel locking behavior. The `lockstat` provider works by dynamically rewriting the kernel functions that manipulate synchronization primitives. As with all other DTrace providers, this instrumentation only occurs as probes are explicitly enabled; the `lockstat` provider induces zero probe effect when not enabled. The `lockstat` provider's instrumentation methodology has existed in Solaris for quite some time — it has historically been the basis for the `lockstat`(1M) command. As part of the DTrace work, the in-kernel component was augmented to become the `lockstat` provider, the `lockstat` command was reimplemented as a DTrace consumer, and the legacy custom-built, single-purpose data-processing framework was discarded.

4.4 Profiling

The providers described above provide probes that are anchored to specific points in text. However, DTrace also allows for *unanchored probes* — probes that are not associated with any particular point of execution but rather with some asynchronous event source. Among these is the `profile` provider, for which the event source is a time-based interrupt of specified interval. These probes can be used to sample some aspect of system state every specified unit of time, and the samples can then be used to infer system behavior. Given the arbitrary actions that DTrace supports, the `profile` provider can be used to sample practically any datum in the system. For example, one could sample the state of the current thread, the state of the CPU, the current stack trace, or the current machine instruction.

5 D Language

DTrace users can specify arbitrary predicates and actions using the high-level D programming language. D is a C-like language that supports all ANSI C operators and allows access to the kernel's native types and global variables. D includes support for several kinds of user-defined variables, including global, clause-local, and thread-local variables and associative arrays. D programs are compiled into DIF by a compiler implemented in the DTrace library; the DIF is then bundled into an in-memory object file representation and sent to the in-kernel DTrace framework for validation and probe enabling. The `dtrace`(1M) command provides a generic front-end to the D compiler and DTrace, but other layered tools can be built on top of the compiler library as well, such as the new implementation of `lockstat`(1M) described earlier.

5.1 Program Structure

A D program consists of one or more *clauses* that describe the instrumentation to be enabled by DTrace. Each probe clause has the form:

probe-descriptions
/predicate/
{
    action-statements
}

Probe descriptions are specified using the form *provider*:*module*:*function*:*name*. Omitted fields match any value, and `sh`(1) globbing syntax is supported. The predicate and action statements may each be optionally omitted.

D uses a program structure similar to `awk`(1) because tracing programs resemble pattern matching programs in that execution order does not follow traditional function-oriented program structure; instead, execution order is defined by a set of external inputs and the tracing program "reacts" by executing the predefined matching clauses. During internal testing, the meaning of this program form was immediately obvious to UNIX developers and permitted rapid adoption of the language.

5.2 Types, Operators and Expressions

As C is the language of UNIX, D is designed to form a companion language to C for use in dynamic tracing. D predicates and actions are written identically to C language statements, and all of the ANSI C operators can be used and follow identical precedence rules. D also supports all of the intrinsic C data types, `typedef`, and the ability to define `struct`, `union`, and `enum` types. Users are also permitted to define and manipulate their own variables, described shortly, and access a set of predefined functions and variables provided by DTrace.

The D compiler also makes use of C source type and symbol information provided by a special kernel service, allowing D programmers to access C types and global variables defined in kernel source code without declaring them. The FBT provider exports the input arguments and return values of kernel functions to DTrace when its probes fire, and the C type service also allows the D compiler to automatically associate these arguments with their corresponding C data types in a D program clause that matches an FBT probe.

Unlike a traditional C source file, a D source file may access types and symbols from a variety of separate scopes, including the core kernel, multiple loadable kernel modules, and any type and variable definitions provided in the D program itself. To manage access to external namespaces, the backquote (`) character can be inserted in symbol and type identifiers to reference the namespace denoted by the identifier preceding the backquote. For example, the type `struct foo`bar` would name the C type `struct bar` in a kernel module named `foo`, and the identifier `rootvp` would match the kernel global variable `rootvp` and would have the type `vnode_t *` automatically assigned to it by the D compiler.

5.3 User Variables

D programs can declare global variables using C declaration syntax in the outer scope of the program, or they can be implicitly defined by assignment statements. When variables are defined by assignment, the left-hand identifier is defined and assigned the type of the right-hand expression for the remainder of the program. Our experience showed that D programs were rapidly developed and edited and often written directly on the `dtrace`(1M) command-line, so users benefited from the ability to omit declarations for simple programs.

5.4 Variable Scopes

In addition to global variables, D programs can create *clause-local* and *thread-local* variables of any type. Variables from these two scopes are accessed using the reserved prefixes `this->` and `self->` respectively. The prefixes serve to both separate the variable namespaces and to facilitate their use in assignment statements without the need for prior declaration. Clause-local variables access storage that is re-used across the execution of D program clauses, and are used like C automatic variables. Thread-local variables associate a single variable name with separate storage for each operating system thread, including interrupt threads.

Thread-local variables are used frequently in D to associate data with a thread performing some activity of interest. For example, to trace the amount of time any thread spends in a `read`(2) system call, one can write the D program:

```
syscall::read:entry
{
        self->t = timestamp;
}

syscall::read:return
/self->t/
{
        printf("%d/%d spent %d nsecs in read\n",
            pid, tid, timestamp - self->t);
}
```

The thread-local variable `self->t` is instantiated on-demand when any thread fires the `syscall::read:entry` probe and is assigned the value of the built-in `timestamp` variable; the program then computes the time difference when the

system call returns. As with traced data, DTrace reports any failure to allocate a dynamic variable so data is never silently lost.

5.5 Associative Arrays

D programs can also create associative array variables where each array element is indexed by a tuple of expression values and data elements are created on-demand. For example, the D program statement `a[123, "hello"] = 456` defines an associative array `a` with tuple signature `[int, string]` where each element is an `int`, and then assigns the value 456 to the element indexed by tuple `[123, "hello"]`. D supports both global and thread-local associative arrays. As in other languages such as Perl, associative arrays permit D programmers to easily create and manage complex dictionary data structures without requiring them to manage memory and write lookup routines.

5.6 Strings

D provides a built-in `string` type to resolve the ambiguity of the C `char*` type, which can be used to represent an arbitrary address, the address of a single character, or the address of a NUL-terminated string. The D `string` type acts like the C type `char [n]` where n is a fixed string limit that can be adjusted at compile-time. The string limit is also enforced by the DTrace in-kernel component, so that it can provide built-in functions such as `strlen()` and ensure finite running time when an invalid string address is specified. D permits strings to be copied using the `=` operator and compared using the relational operators. D implicitly promotes `char*` and `char[]` to `string` appropriately.

6 Aggregating Data

When instrumenting the system to answer performance-related questions, it is often useful to think not in terms of data gathered by individual probes, but rather how that data can be aggregated to answer a specific question. For example, if one wished to know the number of system calls by user ID, one would not necessarily care about the datum collected at *each* system call — one simply wants to see a table of user IDs and system calls. Historically, this question has been answered by gathering data at each system call, and postprocessing the data using a tool like `awk`(1) or `perl`(1). However, in DTrace the aggregating of data is a first-class operation, performed *at the source*.

6.1 Aggregating Functions

We define an *aggregating function* to be one that has the following property:

$$f(f(x_0) \cup f(x_1) \cup ... \cup f(x_n)) = f(x_0 \cup x_1 \cup ... \cup x_n)$$

where x_n is a set of arbitrary data. That is, applying an aggregating function to subsets of the whole and then applying it again to the set of results gives the same result as applying it to the whole itself. Many common functions for understanding a set of data are aggregating functions, including counting the number of elements in the set, computing the maximum value of the set, and summing all elements in the set. Not all functions are aggregating functions, however; computing the mode and computing the median are two examples of non-aggregating functions.

Applying aggregating functions to data *in situ* has a number of advantages:

- The entire data set need not be stored. Whenever a new element is to be added to the set, the aggregating function is calculated given the set consisting of the current intermediate result and the new element. After the new result is calculated, the new element may be discarded. This reduces the amount of storage required by a factor of the number of data points — which is often quite large.
- A scalable implementation is allowed. One does not wish for data collection to induce pathological scalability problems. Aggregating functions allow for intermediate results to be kept *per-CPU* instead of in a shared data structure. When a system-wide result is desired, the aggregating function may then be applied to the set consisting of the per-CPU intermediate results.

6.2 Aggregations

DTrace implements aggregating functions as *aggregations*. An aggregation is a named structure indexed by an n-tuple that stores the result of an aggregating function. In D, the syntax for an aggregation is:

`@`*identifier* `[`*keys*`] = `*aggfunc*`(`*args*`);`

where *identifier* is an optional name of the aggregation, *keys* is a comma-separated list of D expressions, *aggfunc* is one of the DTrace aggregating func-

tions and *args* is a comma-separated list of arguments to the aggregating function. (Most aggregating functions take just a single argument that represents the new datum.)

For example, the following DTrace script counts `write`(2) system calls by application name:

```
syscall::write:entry
{
        @counts[execname] = count();
}
```

By default, aggregation results are displayed when `dtrace`(1M) terminates. (This behavior may be changed by explicitly controlling aggregation output with the `printa` function.) Assuming the above were named "`write.d`", running it might yield:

```
# dtrace -s write.d
dtrace: script 'write.d' matched 1 probe
^C
  dtrace                                            1
  cat                                               4
  sed                                               9
  head                                              9
  grep                                             14
  find                                             15
  tail                                             25
  mountd                                           28
  expr                                             72
  sh                                              291
  tee                                             814
  sshd                                           1996
  make.bin                                       2010
```

In the above output, one might perhaps be interested in understanding more about the `write` system calls from the processes named "`sshd`." For example, to get a feel for the distribution of write sizes per file descriptor, one could aggregate on `arg0` (the file descriptor argument to the `write` system call), specifying the "`quantize()`" aggregating function (which generates a power-of-two distribution) with an argument of `arg2` (the size argument to the `write` system call):

```
syscall::write:entry
/execname == "sshd"/
{
        @[arg0] = quantize(arg2);
}
```

Running the above yields a frequency distribution for each file descriptor. For example:

```
    5
 value  --------- Distribution --------- count
    16 |                                   0
    32 |                                   1
    64 |                                   0
   128 |                                   0
   256 |@@                                 13
   512 |@@                                 13
  1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  199
  2048 |                                   0
```

The above output would indicate that for file descriptor five, 199 writes were between 1024 and 2047 bytes. If one wanted to understand the origin of writes to this file descriptor, one could (for example) add to the predicate that `arg0` be five, and aggregate on the application's stack trace by using the `ustack` function:

```
syscall::write:entry
/execname == "sshd" && arg0 == 5/
{
        @[ustack()] = quantize(arg2);
}
```

7 User-level Instrumentation

DTrace provides instrumentation of user-level program text through the `pid` provider, which can instrument arbitrary instructions in a specified process. The `pid` provider is slightly different from other providers in that it actually defines a *class* of providers — each process can potentially have an associated `pid` provider. The process identifier is appended to the name of each `pid` provider. For example, the probe `pid1203:libc.so.1:malloc:entry` corresponds to the function entry of `malloc`(3C) in process 1203.

In keeping with the DTrace philosophy of dynamic instrumentation, target processes need not be restarted to be instrumented and, as with other providers, there is no `pid` provider probe effect when the probes are not enabled.

The techniques used by the `pid` provider are ISA-specific, but they all involve a mechanism that rewrites the instrumented instruction to induce a trap into the operating system. The trap-based mechanism has a higher enabled probe effect than branch-based mechanisms used elsewhere[12], but it completely unifies kernel- and user-level instrumentation: any DTrace mechanism that may be used with kernel-level probes may also be used with user-level probes.

For example, one can use a thread-local D variable

to follow *all* activity — user-level *and* kernel-level — from a specified user-level function:

```
#!/usr/sbin/dtrace -s
#pragma D option flowindent

pid$1::$2:entry
{
        self->trace = 1;
}

pid$1:::entry, pid$1:::return, fbt:::
/self->trace/
{
        printf("%s", curlwpsinfo->pr_syscall ?
            "K" : "U");
}

pid$1::$2:return
/self->trace/
{
        self->trace = 0;
}
```

The above script uses the D macro argument variables "`$1`" and "`$2`" to allow the target process identifier and the user-level function to be specified as arguments to the script. Assuming that this script were named "`all.d`," one could run it this way:

```
# ./all.d `pgrep xclock` XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0  -> XEventsQueued                          U
  0    -> _XEventsQueued                       U
  0      -> _X11TransBytesReadable             U
  0      <- _X11TransBytesReadable             U
  0      -> _X11TransSocketBytesReadable       U
  0      <- _X11TransSocketBytesReadable       U
  0      -> ioctl                              U
  0        -> ioctl                            K
  0          -> getf                           K
  0            -> set_active_fd                K
  0            <- set_active_fd                K
  0          <- getf                           K
  0          -> get_udatamodel                 K
  0          <- get_udatamodel                 K
...
  0          -> releasef                       K
  0            -> clear_active_fd              K
  0            <- clear_active_fd              K
  0            -> cv_broadcast                 K
  0            <- cv_broadcast                 K
  0          <- releasef                       K
  0        <- ioctl                            K
  0      <- ioctl                              U
  0    <- _XEventsQueued                       U
  0  <- XEventsQueued                          U
```

Note the crossing of the user/kernel boundary in the above: while other instrumentation frameworks allow for some unified tracing, this is perhaps the clearest display of control flow across the user/kernel boundary.

DTrace also allows for tracing of data from user processes. The `copyin()` and `copyinstr()` subroutines can be used to access data from the current process. For example, the following script aggregates on the name (`arg0`) passed to the `open`(2) system call:

```
syscall::open:entry
{
        @files[copyinstr(arg0)] = count();
}
```

By tracing events in both the kernel and user processes, and combining data from both sources, DTrace provides the complete view of the system required to understand systemic problems that span the user/kernel boundary.

8 Speculative Tracing

In a tracing framework that offers coverage as comprehensive as that of DTrace, the challenge for the user quickly becomes figuring out what *not* to trace. In DTrace, the primary mechanism for filtering out uninteresting events is the predicate mechanism discussed in Section 3.2. Predicates are useful when it is known at the time that a probe fires whether or not the probe event is interesting. For example, if one is only interested in activity associated with a certain process or a certain file descriptor, one can know when the probe fires if it associated with the process or file descriptor of interest. However, there are some situations in which one may not know whether or not a given probe event is interesting until some time *after* the probe fires.

For example, if a system call is failing sporadically with a common error code (e.g. `EIO` or `EINVAL`), one may wish to better understand the code path that is leading to the error condition. To capture the code path, one could enable every probe — but only if the failing call can be isolated in such a way that a meaningful predicate can be constructed. If the failures were sporadic or nondeterministic, one would be forced to trace all events that *might* be interesting, and later postprocess the data to filter out the ones that were not associated with the failing code path. In this case, even though the number of interesting events may be reasonably small, the

number of events that must be traced is very large — making postprocessing difficult if not impossible.

To address this and similar situations, DTrace has a facility for *speculative tracing*. Using this facility, one may tentatively trace data; later, one may decide that the traced data is interesting and *commit* it to the principal buffer, or one may decide that the traced data is uninteresting, and *discard* it. For example, to speculatively trace all `ioctl`(2) system calls that return failure:

```
#pragma D option flowindent

syscall::ioctl:entry
/pid != $pid/
{
        self->spec = speculation();
}

fbt:::
/self->spec/
{
        speculate(self->spec);
        printf("%s: %d", execname, errno);
}

syscall::ioctl:return
/self->spec && errno != 0/
{
        commit(self->spec);
        self->spec = 0;
}

syscall::ioctl:return
/self->spec && errno == 0/
{
        discard(self->spec);
        self->spec = 0;
}
```

Note that this script uses the "`$pid`" variable to avoid tracing any failing `ioctl` calls by `dtrace` itself. Running the above:

```
# dtrace -s ./ioctl.d
dtrace: script './ioctl.d' matched 27778 probes
CPU FUNCTION
  0  -> ioctl                    dhcpagent: 0
  0    -> getf                   dhcpagent: 0
  0      -> set_active_fd        dhcpagent: 0
  0      <- set_active_fd        dhcpagent: 0
  0    <- getf                   dhcpagent: 0
  0    -> fop_ioctl              dhcpagent: 0
  0      -> ufs_ioctl            dhcpagent: 0
  0      <- ufs_ioctl            dhcpagent: 0
  0    <- fop_ioctl              dhcpagent: 0
  0    -> releasef               dhcpagent: 0
  0      -> clear_active_fd      dhcpagent: 0
  0      <- clear_active_fd      dhcpagent: 0
  0      -> cv_broadcast         dhcpagent: 0
  0      <- cv_broadcast         dhcpagent: 0
  0    <- releasef               dhcpagent: 0
  0    -> set_errno              dhcpagent: 0
  0    <- set_errno              dhcpagent: 25
  0  <- ioctl                    dhcpagent: 25
```

9 Experience

DTrace has been used extensively inside Sun to understand system behavior in both development and production environments. One production environment in which DTrace has been especially useful is a SunRay server in Broomfield, Colorado. The server — which is run by Sun's IT organization and has 10 CPUs, 32 gigabytes of memory, and approximately 170 SunRay users — was routinely exhibiting sluggish performance. DTrace was used to resolve many performance problems on this production system; the following is the detailed description of the resolution of one such problem.

By looking at the output of `mpstat`(1), a traditional Solaris monitoring tool, it was noted that the number of cross-calls per CPU per second was quite high. (A cross-call is a function call directed to be performed by a specified CPU.) This led to the natural question: who (or what) was inducing the cross-calls? Traditionally, there is no way to answer this question concisely, as there is no static instrumentation in the cross-call mechanism. The DTrace "`sysinfo`" provider, however, can dynamically instrument every increment of the counters consumed by `mpstat`. So by using DTrace and `sysinfo`'s "`xcalls`" probe, this question can be easily answered:

```
sysinfo:::xcalls
{
        @[execname] = count();
}
```

Running the above gives a table of application names and the number of cross-calls that each induced; running it on the server in question revealed that virtually all application-induced cross calls were due to the "`Xsun`" application, the Sun X server. This wasn't *too* surprising — as there is an X server for each SunRay user, one would expect them to do much of the machine's work. Still, the high number of cross-calls merited further investigation: what were the X servers doing to induce the cross-calls? To answer this question, the following D script was written:

```
syscall:::entry
/execname == "Xsun"/
{
        self->sys = probefunc;
}

sysinfo:::xcalls
/execname == "Xsun"/
{
        @[self->sys != NULL ?
            self->sys : "<none>"] = count();
}

syscall:::return
/self->sys != NULL/
{
        self->sys = NULL;
}
```

This script uses a thread-local variable to keep track of the current system call name; when the `xcalls` probe fires, it aggregates on the system call that induced the cross-call. In this case, the script revealed that nearly all cross-calls from "`Xsun`" were being induced by the `munmap`(2) system call. The fact that `munmap` activity induces cross-calls is not surprising (memory demapping induces a cross call as part of TLB invalidation), but the fact that there was so *much* `munmap` activity (thousands of `munmap` calls per second, system wide) was unexpected.

Given that ongoing `munmap` activity must coexist with ongoing `mmap`(2) activity, the next question was what were the X servers `mmap`'ing? And were there some X servers that were `mmap`'ing more than others? Both of these questions can be answered at once:

```
syscall::mmap:entry
/execname == "Xsun"/
{
        @[pid, arg4] = count();
}

END
{
        printf("%9s %13s %16s\n",
            "PID", "FD", "COUNT");
        printa("%9d %13d %16@d\n", @);
}
```

This script aggregates on both process identifier and `mmap`'s file descriptor argument to yield a table of process identifiers and `mmap`'ed file descriptors. It uses the special DTrace `END` probe and the `printa` function to precisely control the output. Here is the tail of the output from running the above D script on the production SunRay server:

```
      PID            FD           COUNT
      ...            ..             ...
    26744             4              50
     2219             4              56
    64907             4              65
    23468             4              65
    45317             4              68
    11077             4            1684
    63574             4            1780
     8477             4            1826
    55758             4            1850
    38710             4            1907
     9973             4            1948
```

As labelled above, the first column is the process identifier, the second column is the file descriptor, and the third column is the count. (`dtrace`(1M) always sorts its aggregation output by aggregation value.) The data revealed two things: first, that all of the `mmap` activity for each of the X servers orginated from file descriptor 4 in each. And second, that six of the 170 X servers on the machine were responsible for most of the `mmap` activity. Using traditional process-centric tools (e.g., `pfiles`(1)) revealed that in each X server file descriptor 4 corresponded to the file "`/dev/zero`," the `zero`(7D) device present in most UNIX variants. `mmap`'ing `/dev/zero` is a technique for allocating memory, but why were the X servers allocating (and deallocating) so much memory so frequently? To answer this, we wrote a script to aggregate on the user stack trace of the X servers when they called `mmap`:

```
syscall::mmap:entry
/execname == "Xsun"/
{
        @[ustack()] = count();
}
```

Running this yields a table of stack traces and counts. In this case, *all* `Xsun` `mmap` stack traces were identical:

```
  libc.so.1‘mmap+0xc
  libcfb32.so.1‘cfb32CreatePixmap+0x74
  ddxSUNWsunray.so.1‘newt32CreatePixmap+0x20
  Xsun‘ProcCreatePixmap+0x118
  Xsun‘Dispatch+0x17c
  Xsun‘main+0x788
  Xsun‘_start+0x108
```

The stack trace indicated why the X servers were allocating (and deallocating) memory: they were creating (and destroying) Pixmaps. This answered the immediate question, and raised a new one: what applications were ordering their X servers to create and destroy Pixmaps? Answering this required a somewhat more sophisticated script:

```
syscall::poll:entry
/execname == "Xsun"/
{
        self->interested = 0;
}

syscall::mmap:entry
/execname == "Xsun"/
{
        self->interested = 1;
}

fbt::sleepq_unlink:entry
/self->interested/
{
        this->u = &args[1]->t_procp->p_user;
        @[stringof(this->u->u_comm)] = count();
}
```

This script exploits some implementation knowledge of both X servers and the kernel. An X server works by calling `poll`(2) on its connections to wait for requests; when a request arrives, the X server (a single-threaded process) processes the request and sends the response. Sending the response causes the X server to awaken the blocking client, after which the X server again polls on its connections. To determine for whom the X servers were creating Pixmaps, we set a thread-local variable ("`interested`") when the X server called `mmap`. We then enabled the FBT probe in the kernel's routine to awaken another thread ("`sleepq_unlink()`"); if (and only if) `interested` was set, we aggregated on the process that we were waking. The core assumption was that the process that the X server awakened immediately after having performed an `mmap` was the process for whom that `mmap` was performed.

Running the above on the production SunRay server produced the following (trimmed) output:

```
  ...
  gedit                                25
  soffice.bin                          26
  netscape-bin                         44
  gnome-terminal                       81
  dsdm                                487
  gnome-smproxy                       490
  metacity                            546
  gnome-panel                         549
  gtik2_applet2                      6399
```

This output was the smoking gun — it immediately focussed all attention on the application "`gtik2_applet2`," a stock ticker applet for the GNOME desktop. A further DTrace script that aggregated on user stack revealed the source of the problem: `gtik2_applet2` was creating (and destroying) an X graphics context (GC) *every 10 milliseconds*.[7] As any X programmer knows, GC's are expensive server-side objects — they are not to be created with reckless abandon[4]. While there were only six instances of `gtik2_applet2` running on the SunRay server, *each* was inducing this expensive operation from their X servers (and subsequently from the operating system) one hundred times per second; taken together, they were having a substantial effect on system performance. Indeed, stopping the six `gtik2_applet2` processes dramatically improved the system's performance: cross-calls dropped by 64 percent, involuntary context switches dropped by 35 percent, system time went down 27 percent, user time went down 37 percent and idle time went up by 15 percent.

This was a serious (and in retrospect, glaring) performance problem. But it was practically impossible to debug with traditional tools because it was a *systemic* problem: the `gtik2_applet2` processes were doing very little work themselves — they were inducing work on their behalf from other components of the system. To root-cause the problem, we made extensive use of aggregations and and thread-local variables, two features unique to DTrace.

10 Future Work

DTrace provides a stable and extensible foundation for future work to enhance our ability to observe production systems. We are actively developing extensions to DTrace, including:

- Performance counters. Modern microprocessors such as SPARC and x86 export performance counter registers that can be programmed to count branch mispredicts, cache misses, and other processor events. We plan to implement a DTrace provider that exports performance counter information and allows it to be accessed in D from a probe action.

- Helper actions. Complex middleware may wish to assist DTrace with actions that require knowledge specific to the middleware. We have developed a prototype of such a helper action that permits applications to provide assistance for DTrace in obtaining a user-level stack trace. We have implemented the helper action in the Java Virtual Machine, allowing for `ustack` to

[7] See `http://bugzilla.gnome.org/show_bug.cgi?id=99696` for details.

obtain a user-level stack trace that contains both Java and C/C++ stack frames.

- User lock analysis. The `pid` provider can instrument any function in a user process, including user-level synchronization facilities. We have developed a prototype user-level equivalent to the kernel `lockstat`(1M) utility, dubbed `plockstat`, that can perform dynamic lock-contention analysis of multi-threaded user processes.

11 Conclusions

We have described DTrace, a new facility for dynamic instrumentation of both user-level and kernel-level software in production systems. We have described the principal features of DTrace, including the details of D, its high-level control language. Although there remain other important features of DTrace for which space did not permit a detailed description (e.g. postmortem tracing, boot-time tracing) we have highlighted the major advances in DTrace over prior work in dynamic instrumentation: thread-local variables, associative arrays, data aggregation, seamlessly unified user-/kernel-level tracing, and speculative tracing. We have demonstrated the use of DTrace in root-causing an actual, serious performance problem on a production system — a problem that could not have been root-caused in a production environment prior to this work.

Acknowledgements

Many people at Sun were invaluable in the development of DTrace. We are especially grateful to Bart Smaalders, Gavin Maltby, Jon Haslam, Jonathan Adams, and Bill Moore; their experience, ideas, and tireless advocacy were integral to the success of DTrace. Further, we are grateful to Jarod Jenson of Aeysis, Inc., who agreed to be the Alpha customer for DTrace; it has been singularly rewarding to see Jarod using DTrace to find previously undiagnosable system performance problems. Many people at Sun reviewed drafts of this paper; in particular, it was much improved by the detailed comments of Val Henson, Gavin Maltby, Eric Lowe, Jon Haslam, and Glenn Skinner.

References

[1] M. Auguston, C. Jeffery, and S. Underwood. A monitoring language for run time and post-mortem behavior analysis and visualization. In *5th International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, 2003.

[2] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.

[3] J. K. Hollingsworth, B. P. Miller, M. J. R. Gonçalves, O. Naim, Z. Xu, and L. Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, Nov. 1997.

[4] E. F. Johnson and K. Reichard. *Professional Graphics Programming in the X Window System.* MIS Press, Portland, OR, 1993.

[5] B. P. Miller, 2003. Personal communication.

[6] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[7] R. J. Moore. A universal dynamic trace for linux and other operating systems. In *Proceedings of the FREENIX Track*, June 2001.

[8] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.

[9] Sun Microsystems, Santa Clara, California. *Solaris Dynamic Tracing Guide*, 2003.

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

[11] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *SC'2003 Conference CD*, 2003.

[12] Z. Xu, B. P. Miller, and O. Naim. Dynamic instrumentation of threaded applications. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1999.

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

[14] T. Zanussi, K. Yaghmour, R. Wisniewski, R. Moore, and M. Degenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *Proceedings of the Ottawa Linux Symposium 2003*, July 2003.