DTrace: Dynamic Tracing For Solaris

Bryan Cantrill  Mike Shapiro
(bmc@eng)  (mws@eng)

Solaris Kernel Technologies
DTrace: Dynamic Tracing For Solaris

Bryan Cantrill (bc30992@japan)  Mike Shapiro (ms36066@sfbay)

Solaris Kernel Technologies
A Modern Tracing Framework

- Must have zero probe effect when disabled
- Must allow for novel tracing technologies
- Must allow for thousands of probes
- Must allow arbitrary numbers of consumers
- Unwanted data must be pruned as early as possible in the data chain
- Data must be coalesced whenever possible, and as early as possible
The DTrace Vision

• Build a tracing framework that provides concise answers to arbitrary questions
• Enable quantum leap in performance analysis and engineering
• Improve RAS through continuous tracing
• Accelerate project development
• Eliminate DEBUG and other special kernels: all facilities available in production
IBM MVS Tracing

- MVS provided wealth of tracing facilities, notably GTF and CTRACE
- IPCS console provided commands to enable, filter, and display GTF, CTRACE trace records
- Extensive probes provided for base operating system, channel programs
- GTRACE() assembler macro used to record data in a user program; can later be merged
GTF Example

- Operator console:

```plaintext
START GTF.EXAMPLE1
AHL103I TRACE OPTIONS SELECTED--SYSM,USR,DSP 00
AHL125A RESPECIFY TRACE OPTIONS OR REPLY U
REPLY 00,U
AHL031I GTF INITIALIZATION COMPLETE
```

- IPCS GTFTRACE output:

```plaintext
DSP ASCB 00F44680 CPU 001 PSW 070C1000
  TCB 00AF2370 R15 80AF2858
  R0 00000001 R1 FDC9E5D4

GMT-07/02/89 00:29:08.155169
```
GTF Example

- Operator console:

```
START GTF.EXAMPLE1
AHL103I TRACE OPTIONS SELECTED--SYSM,USR,DSP
00 AHL125A RESPECIFY TRACE OPTIONS OR REPLY U
AHL031I GTF INITIALIZATION COMPLETE
```

- dispatch event on CPU1
- address space control block
- processor status word
- task control block
- saved register values
VTRACE

- Kernel tracing framework developed early in Solaris 2 (1991)
- Provided a C macro to designate a probe site; some probe effect even when disabled
- Additionally, applications could issue fast trap to record events in in-kernel buffer
- In-kernel buffers could be continuously read out and streamed to disk
VTRACE, cont.

- Scalable and lightweight when enabled
- Fair coverage: \( \approx 1,000 \) trace points
- Used to solve real performance problems
- As a result of disabled probe effect, required a special kernel
- Fell into disrepair during 64-bit port
TNF

- *Trace Normal Form* tracing framework introduced in Solaris 2.5
- Originally a user-level framework (LSARC 1993/650); kernel support tacked on (PSARC 1994/165)
- Like VTRACE, provides C macro to designate a probe site; induces load, compare and branch even if disabled
TNF, cont.

- Uses pseudo per-CPU buffering, resulting in suboptimal CPU scaling

**TNF Performance**

Nanoseconds per probe vs. CPUs
TNF, cont.

• Some of TNF’s failings:
  – Too few probes (≈30 probes in common kernel code)
  – Crude filtering (only based on process ID, and even then doesn’t work for scheduling events)
  – No control over data generated by each probe
  – Doesn’t allow for continuous collection of data
  – Doesn’t correlate kernel data to application activity
  – Bizarre data format designed for use only in postmortem analysis
KernInst

• Kernel instrumentation tool developed at Wisconsin [Tamches, Miller, et al.]

• User-level daemon performs run-time register analysis of kernel object code

• Code patches, trampoline code, and instrumentation are inserted using driver

• Overcomplicated by living outside of core OS

• Does not provide sufficient predicate support

• Unsafe probe insertion causes OS failure!
Linux DProbes

- Dynamic instrumentation kit for Linux kernel [Moore, IBM LTC, et al.]
- Replaces kernel text with breakpoint trap that vectors to user RPN probe program
- Also provides access to Intel debug registers
- Currently under active development
- DProbes facility not part of stock kernel
- Significant safety issues (more later ...)

DTrace: Cantrill, Shapiro 3/11/02
Sun Proprietary/Confidential: Internal Use Only
# Competitive Landscape

<table>
<thead>
<tr>
<th>Feature</th>
<th>GTF</th>
<th>vtrace</th>
<th>TNF</th>
<th>KInst</th>
<th>DProbe</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>user/kernel/merged</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>K</td>
<td>K</td>
<td>users want combined timeline of user and kernel events</td>
</tr>
<tr>
<td>probe coverage</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>framework must provide sufficient probes to solve most problems</td>
</tr>
<tr>
<td>disabled probe effect</td>
<td>●</td>
<td>✗</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>ideal framework has zero probe effect when disabled</td>
</tr>
<tr>
<td>scalability</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>concurrent probe firings must scale to arbitrary number of CPUs</td>
</tr>
<tr>
<td>safety</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>✗</td>
<td>✗</td>
<td>no way for user to induce fatal machine or OS failure</td>
</tr>
<tr>
<td>extensibility</td>
<td>✓</td>
<td>✓</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>framework should allow easy addition of probes and providers</td>
</tr>
<tr>
<td>data filtering</td>
<td>●</td>
<td>✗</td>
<td>✗</td>
<td>○</td>
<td>●</td>
<td>users should be able to filter on arbitrary conditions at probe site</td>
</tr>
<tr>
<td>arbitrary recording</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>●</td>
<td>●</td>
<td>users should be able to record arbitrary data on probe firing</td>
</tr>
<tr>
<td>self describing</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>✗</td>
<td>✗</td>
<td>type information available to consumers for all recorded data</td>
</tr>
<tr>
<td>run-time analysis</td>
<td>●</td>
<td>✗</td>
<td>✗</td>
<td>●</td>
<td>●</td>
<td>run-time analysis tools should be provided, not just post-mortem</td>
</tr>
<tr>
<td>stock availability</td>
<td>●</td>
<td>✗</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>tracing facilities must be available on production systems</td>
</tr>
<tr>
<td>stable abstractions</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>✗</td>
<td>✗</td>
<td>framework must provide stable abstractions for layered tools</td>
</tr>
</tbody>
</table>
Providers

- Tracing frameworks have historically been tied to a single tracing methodology.
- Conversely, new tracing methodologies have had to invent their own frameworks.
- In DTrace, the tracing framework is formally separated from tracing providers.
- Allows for faster adoption of and provides significant leverage for novel tracing methodologies.
Probes

• A trace point in DTrace is called a *probe*

• A probe is identified by a tuple consisting of Provider, Module, Function and Name

• Probes may have Module and Function unspecified (such probes are said to be *unanchored*)

• Each probe has a unique 32-bit ID
Predicates and Actions

- Idea: Provide flexible boolean expressions that can control tracing activities, e.g.

```plaintext
if (pid == process of interest)
then trace data of interest
```

- Must allow completely arbitrary queries to be formulated by user or layered tool

- Must evaluate at probe firing time to prune data stream at earliest opportunity
Provider Interface

- Provider makes available all known probes
- Framework calls into provider to enable a specific probe
- Framework handles multiplexing of multiple consumers of a single probe
- Provider indicates that an enabled probe is hit by calling `dtrace_probe()`, specifying probe ID
dtrace_probe()

• `dtrace_probe()` is called to take appropriate actions (if any) when an enabled probe is hit

• Can be called from any context in which C may be called, e.g.:
  – From high-level interrupt context
  – While interrupts are disabled
  – In synchronization primitives (e.g. `mutex_enter()`)
  – While dispatcher locks are held
dtrace_probe(), cont.

- Disables interrupts for its duration
  - Substantially simpler than implementing lock-free data structures
  - Prevents preemption, CPU migration
  - As fast as performing an atomic memory operation
  - Synchronous cross calls can be used to guarantee that no threads remain in critical section

- Converts probe ID to internal data structure for further processing
dtrace_probe(), cont.

- Iterates over a per-probe chain of *enabling control blocks* (ECBs)
- Each ECB corresponds to an *enabling* of a probe
- The ECB abstraction allows:
  - A given consumer to have multiple, different enablings of a single probe
  - Disjoint consumers to have disjoint enablings of a single probe
Enabling Control Blocks

- Each ECB contains:
  - An optional *predicate*
  - A list of one or more *actions*
  - A pointer to an array of per-CPU *buffers*
- Each ECB has a corresponding *enabled probe ID* (EPID)
- EPID space is per consumer
Enabling Control Blocks, cont.

• Actions are taken on an ECB if and only if:
  – There does not exist a predicate, or
  – The predicate evaluates to a non-zero value

• Actions may identify data to be stored into a trace record

• Actions need not generate trace data
  – May update variable state (more later)
  – May affect system state in a defined way (e.g. BREAKPOINT, PANIC)
Trace Records

- Record size is constant per ECB (and therefore per EPID)
- Records consist of 32-bit EPID, followed by some amount of data
- Library determines record size and layout using a separate EPID dictionary
Trace Records, cont.

- Library’s EPID dictionary can be built dynamically: as a new EPID is seen in the data stream, the library queries for the corresponding record size and layout
- Separating the data stream from the metadata stream facilitates run-time analysis tools
- Lack of data/metadata separation is a serious deficiency in TNF
Buffers

- Buffers are per consumer, per CPU
- Buffers are always allocated in pairs: an active buffer and a spare buffer
- A buffer is consumed by:
  - Issuing a synchronous cross call to the corresponding CPU to switch active buffer with spare buffer
  - Copying out used portion of newly spare buffer (formerly the active buffer) to user-level
Buffer Management

• If buffer is full when a data-generating action is taken, a per-buffer *drop count* is incremented and no action is taken.

• It is up to consumers to minimize drop counts by reading buffers sufficiently often.

• Drop counts are copied out to user-level alongside buffer data; consumers always know if data is incomplete.
Buffer Management, cont.

- A consumer may optionally indicate that a buffer is to be treated as a *ring buffer*
- Ring buffers wrap on overflow, writing over older data
- Consumers can avoid data loss by reading buffer sufficiently often
- Useful primarily to provide “black box” style event recording
Function Boundary Tracing

- Would like a probe before every function entry and after every function return
- Would like to implement probes by hot patching kernel text only when enabled -- thereby avoiding performance effect when disabled
- But how to hot patch text?
Branch Insertion?

- Idea is to patch probe point to be an annulled branch-always into a jump table
- Must perform static analysis to ascertain dead registers
- Analysis must somehow statically determine trap level; failure to do so can induce RED state exception
- e.g. KernInst
Software Trap Insertion?

- Idea is to patch desired code to be a trap-always instruction
- Must perform static analysis to avoid placing trap-always instruction where trap level can be non-zero
- Failure to do so can induce RED state exception
- e.g. Hot Diagnosis
Branch Insertion, revisited

- If we *only* patch a function’s initial save instruction, we solve both of the problems with branch insertion:
  - Trap level is implicitly considered: code at TL > 0 may not arbitrarily issue a save
  - Register analysis is obviated by the save: immediately after the save, locals and outputs are dead
Entry Patching

Function

| save  | %sp, -0xc0, %sp |
| ldx   | [%i0+ 0x3b0], %l6 |
| ...   |                |

We patch the save instruction to be an annulled branch-always into a per-probe entry in a per-module jump table
Entry Patching

The jump table entry:
- Performs the patched-over save
- Moves the inputs into the outputs
- Sets %o7 to be (patched_pc - 4)
- Calls dtrace_probe()
Entry Patching, cont.

- The first instruction of a non-leaf function is not always a save instruction

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>sethi %hi(0x1494800),%g2</td>
</tr>
<tr>
<td>sethi %hi(0x140a000),%g1</td>
</tr>
<tr>
<td>save %sp, −0xb0, %sp</td>
</tr>
<tr>
<td>ldx [%g2 + 0x98], %g3</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

- Correctly patching the save instruction in this case would require static register analysis: live registers volatile across the call to `dtrace_probe()` must be preserved
Entry Patching, cont.

• We instead patch the first instruction to be the annulled branch-always

• In this case, the jump table entry:
  – Performs a \texttt{MINFRAME} save
  – Moves the inputs into the outputs
  – Calls \texttt{dtrace_probe()}
  – Performs a restore
  – Branches back to \((\texttt{patched}\_\texttt{pc} + 4)\) with the patched-over instruction in the delay slot
Entry Patching, cont.

Function

\begin{align*}
\text{ba, a} & \quad . + \text{offset} \\
\text{sethi} & \quad %hi(0x140a000), %g1 \\
\text{save} & \quad %sp, −0xb0, %sp \\
\text{ldx} & \quad [ %g2 + 0x98 ], %g3 \\
\ldots & & 
\end{align*}

Per module FBT table

\begin{align*}
\ldots & \\
\text{save} & \quad %sp, −\text{MINFRAME}, %sp \\
\text{set} & \quad \text{probe_id}, %o0 \\
\text{mov} & \quad %i0, %o1 \\
\ldots & \\
\text{call} & \quad \text{dtrace_probe} \\
\text{mov} & \quad %i4, %o5 \\
\text{restore} & \\
\text{ba} & \quad . + \text{offset} \\
\text{sethi} & \quad %hi(0x1494800), %g2 \\
\ldots & & 
\end{align*}
Return Patching

- ret/restore couplets can be patched in much the same way as save instructions
- The ret is patched to be an annulled branch-always into a jump table entry
Return Patching, cont.

The jump table entry:

- Calls `dtrace_probe()`, passing both the return value and the offset of the ret
- On return from `dtrace_probe()`, performs the ret/restore couplet

<table>
<thead>
<tr>
<th>Function</th>
<th>Per module FBT table</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x17c: <code>mov 1, %i0</code></td>
<td><code>set probe_id, %o0</code></td>
</tr>
<tr>
<td>0x180: <code>ba,a + offset</code></td>
<td><code>mov 0x180, %o1</code></td>
</tr>
<tr>
<td>0x184: <code>restore</code></td>
<td><code>call dtrace_probe</code></td>
</tr>
<tr>
<td></td>
<td><code>mov %i0, %o2</code></td>
</tr>
<tr>
<td></td>
<td><code>ret</code></td>
</tr>
<tr>
<td></td>
<td><code>restore</code></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Return Patching, cont.

- ret/restore couplets are not the only way to return from a non-leaf routine
- call/restore and jmpl/restore couplets are used to implement tail-call elimination

Function

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x17c</td>
<td>stx [%g2], %g3</td>
</tr>
<tr>
<td>0x180</td>
<td>call mutex_exit</td>
</tr>
<tr>
<td>0x184</td>
<td>restore %g0,%10,%00</td>
</tr>
</tbody>
</table>
Return Patching, cont.

<table>
<thead>
<tr>
<th>Function</th>
<th>Per module FBT table</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x17c: stx [%g2], %g3</td>
<td>set probe_id, %o0</td>
</tr>
<tr>
<td>0x180: ba,a . + offset</td>
<td>mov 0x180, %o1</td>
</tr>
<tr>
<td>0x184: restore %g0,%10,%o0</td>
<td>call dtrace_probe</td>
</tr>
<tr>
<td></td>
<td>mov %i0, %o2</td>
</tr>
<tr>
<td></td>
<td>call mutex_exit</td>
</tr>
<tr>
<td></td>
<td>restore %g0,%10,%o0</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Principle is the same:

- control-transfer instruction is patched to be an annulled branch-always
- Jump table entry performs control-transfer/restore couplet upon return from `dtrace_probe()`
Return Patching, cont.

- Both `jmpl` and `restore` can operate on register operands

- Must preserve operands volatile across the call to `dtrace_probe()` (i.e., inputs and globals)

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x17c:</td>
<td><code>ldx [%g2], %g3</code></td>
<td></td>
</tr>
<tr>
<td>0x180:</td>
<td><code>jmpl %g3, %o7</code></td>
<td></td>
</tr>
<tr>
<td>0x184:</td>
<td><code>restore %g0, %g2, %o0</code></td>
<td></td>
</tr>
</tbody>
</table>
Return Patching, cont.

- The volatile registers are moved into unused locals
- The instructions using the volatile operands are restructured to be in terms of the local
Choosing Eligible Functions

- Always err on the side of caution: if a function looks like it’s trying to be clever or appears otherwise strange, don’t create probes for it.
- Only create probes for functions containing both a patchable entry and a patchable return.
- (Well, plus `resume_from_zombie()`.)
dtrace(1M) syntax

dtrace [ -i id ]

[ -P prov ]

[ -m [ prov: ] mod ]


Language Design

- The kernel is written in C, so the natural choice for low-level predicates is C:
  
  ```c
  curthread->t_cpu->cpu_id == 0 &&
  curthread->t_cpu->cpu_idle_thread == curthread ...
  ```

- The Kernel Stabs project (PSARC 2001/021) provided native type info in CTF, so it is possible to build a dynamic evaluator

- Same language for predicates and actions
Introducing “D”

- Complete access to native kernel C types
- Complete access to statics and globals
- Complete support for all ANSI-C operators
- Support for strings as a first-class citizen
- Support for built-in variables (timestamp, curthread, arguments, machine regs, etc.)
- Compiler provided as a library API
Implementing D

Clients send D expressions to library for compilation

Compiler stack produces D Intermediate Format (DIF) objects that can be bound to probe locations

DTrace driver stores DIF in the kernel and executes program at probe firing time
DIF Architecture

- Small RISC architecture ideal for simple emulation or on-the-fly code generation
  - variable number of 64-bit registers (%r0 = 0)
  - 64-bit arithmetic and logical instructions
  - 1, 2, 4, and 8-byte safe memory loads
  - standard branches and condition codes
  - instructions to access variables, strings
  - ~50 opcodes, ~200 line emulator (plus some supporting routines for loads, variables, etc.)
DIF Example

- D expression: "curthread->t_cpu"
- DIF code:
  - `ldgs 256, %r1` ! 256 = "curthread"
  - `setx 0x00000000.000000a8, %r2`
  - `add %r1, %r2, %r1`
  - `ldx [%r1], %r1`
  - `ret %r1`
- DIF object: DIF

- text section
- string table
- variable table
- return type
DIF Safety

- All DIF objects are validated by the kernel:
  - valid opcodes
  - valid string refs
  - reserved bits
  - valid registers
  - valid variables must be zero
- Only **forward** branches are permitted
- Limit on maximum size of DIF object
- DTrace runtime handles invalid loads, misaligned loads, and division by zero
- DTrace runtime prevents access to i/o space addresses using new vmem arena
DIF Load Safety

- DIF engine load routines check alignment and i/o space arena before issuing load
- Per-CPU DTrace fault protection flag is set
- If search fails and protection is on, sfmmu sets fault flag and issues `done` instead of calling `sfmmu_pagefault()`
- Failed load aborts processing of current ECB
D Strings

- First-class strings provided to avoid ambiguity of `char*` and `char[]` in C
- Quoted strings are assigned string type
- Scalars can be promoted to string type using new `stringof()` operator
- Operators `<`, `<=`, `>`, `>=`, `!=`, `==` overloaded as `strcmp(3C)`; promote `char*` and `char[]`:

```
curthread->t_procp->p_user.u_comm == "ksh"
```
D Limitations

- Still need to find some solution for #defines that are used as flag bits:
  
  ```c
  (curthread->t_proc_flag & TP_PRVSTOP)
  ```

- Preprocessor approach possible but messy

- Ideally extend compiler or tools and CTF to support association directly in C source

- Solution would also benefit other debugging tools (e.g. mdb(1) ::print)
Linux DProbes Comparison

• RPN-like IR developed in advance of forthcoming high-level language

• Safety issues not thoroughly considered:
  – user can induce panic if probes are placed improperly
  – user can modify registers, memory, write to i/o ports
  – validation performed in tool and libraries, not kernel
  – infinite loop problem handled by forcing user to specify `jmp_max=123` in probe program
dtrace(1M) syntax

dtrace [ -i id [ predact ]]
[ -P prov [ predact ]]
[ -m [ prov: ] mod [ predact ]]

predact ⇒ [ / predicate / ] { action }
dtrace(1M) example

```bash
# dtrace -n 'write32:entry

/ curthread->t_procp->p_user->u_comm == "ksh" /

{ trace(curthread->t_procp->p_pipd->pid_id) }`

dtrace: 'write32:entry' matched 1 probe.

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6796</td>
<td>write32:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>115575</td>
</tr>
<tr>
<td>0</td>
<td>6796</td>
<td>write32:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>115575</td>
</tr>
<tr>
<td>0</td>
<td>6796</td>
<td>write32:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>115575</td>
</tr>
<tr>
<td>0</td>
<td>6796</td>
<td>write32:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100392</td>
</tr>
<tr>
<td>0</td>
<td>6796</td>
<td>write32:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100392</td>
</tr>
<tr>
<td>0</td>
<td>6796</td>
<td>write32:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100392</td>
</tr>
</tbody>
</table>
```
Variables

• Idea: allow D actions to instantiate, manipulate, and record variables
• Variables can also be used by predicates to alter control flow and correlate events
• Variable species:
  – scalar variables: integers, strings
  – associative arrays: arbitrary (key, value) collections
• Variable scope:
  – global per client instance
  – thread-local per client instance
Scalar Variables in D

- Variables instantiated by D assignment operators: `= += -= *= /= %= &= ^= |= <<= >>= ++ --`
- Variable type determined by first assignment
- Variables initially assigned zeroes
- Definitions cached in libdtrace client state

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Type</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;foo&quot;</td>
<td>int</td>
<td>global</td>
</tr>
<tr>
<td>2</td>
<td>&quot;bar&quot;</td>
<td>str</td>
<td>global</td>
</tr>
<tr>
<td>3</td>
<td>&quot;baz&quot;</td>
<td>int</td>
<td>thread</td>
</tr>
</tbody>
</table>

ldgs 1, %r1
...

1 "foo" int global
3 "baz" int thread
Variable Scope

- Global variables named by plain identifiers
- TLS variables accessed and instantiated by overloading "curthread->"

D compiler precedence rules:
- C type names defined in kernel (ANSI-C rules)
- DTrace built-in variable names
- Global and static variable names defined in kernel
- User-defined variable names
Associative Arrays in D

- Arrays named by *identifier [ expression-list ]*
- Initially filled with zeroes just like scalars
- Entry can be freed by setting it to zero
- Type signature of key and value determined by first assignment; enforced thereafter
- Examples:
  
  ```
  uids[curthread->t_procp->p_cred->cr_uid]++;
  a[curthread, args[0]] = args[1];
  ```
Value Consistency

• Important to provide semantic consistency for data used in both predicate and actions:

  / foo == 3 / { trace(foo) }
  / dnlc_nentries < 10 / { trace(dnlc_nentries) }

• Kernel data consistency can be achieved by taking a snapshot prior to ECB processing

• Variables pose greater challenges:

  / dnlc_nentries < 10 / { foo = 0 }
  / foo++ && bar++ / { trace(foo + bar) }
Variable Consistency

- Variable group must be self-consistent and consistent w.r.t. other modifying ECBs:
  
  ```
  / dnlc_nentries < 10 / { foo = 0 }  
  / foo++ && bar++ / { trace(foo + bar) }  
  ```

- Consistency is achieved by locking variables in a defined order (by ID)

- Only need to do this when variables are used in both predicate and action
**dtrace(1M) syntax**

```
dtrace [ -i id [ predact ]]
[ -P prov [ predact ]]
[ -m [ prov: ] mod [ predact ]]
```

`predact` \(\Rightarrow\) `[ / predicate / ]\{ action \}`
Aggregations

• An *aggregating function* is a function $f(x)$, where $x$ is a sequence of arbitrary length, for which there exists an aggregating function $f'(x)$ such that:

$$f'(f(x_0), f(x_1), \ldots f(x_n)) = f(x_0, x_1, \ldots x_n)$$

• E.g., COUNT, MEAN, MAXIMUM, and MINIMUM are aggregating functions; MEDIAN, and MODE are not
Aggregations, cont.

• When data is to be processed using an aggregating function, the implementation can be made very efficient:
  – Trace records need not be generated; only the intermediate results from the aggregating function need to be stored
  – Intermediate results from aggregating functions can be stored *per CPU*, thereby eliminating data sharing
  – Aggregating function can be periodically performed on all per CPU intermediate results to derive system-wide result
Aggregations, cont.

• An aggregation is an associative table keyed by an n-tuple where each value is the result of an aggregating function

• n-tuple consists of a list of D expressions

• Aggregating functions are provided by the framework

• Framework provides a single aggregation per consumer
Aggregations, cont.

• Current aggregating functions:
  – \( \text{MAX}(expr) \): the intermediate result is set to the greater of the intermediate result and \( expr \)
  – \( \text{COUNT} \): increments the intermediate result
  – \( \text{QUANTIZE}(expr) \): the intermediate result consists of 64 power-of-two buckets; the bucket corresponding to \( expr \) is incremented
  – \( \text{AVG}(expr) \): the intermediate result consists of a count and a total; the count is incremented and the total is increased by \( expr \)
Aggregation Example

• For example, maximum kernel \texttt{bcopy()} size by command name:
  – Enable probe with function “\texttt{bcopy}”, name “\texttt{entry}”
  – Aggregate on:
    \texttt{curthread->t_procp->p_user.u_comm}
  – Set aggregating function to “\texttt{max(arg2)}”
Aggregation Implementation

- Aggregations are implemented using the same buffer infrastructure as trace buffers
- Buffer switching and copying thus fall out
- Aggregations are an associative table; buffering is complicated by the presence of hash table metadata
Aggregation Implementation

- **Data** grows from start of buffer
- **Metadata** grows from end of buffer
- **Only** data is copied out
- **EPID** is in data record, but is *not* considered to be part of the key

<table>
<thead>
<tr>
<th>EPID</th>
<th>key</th>
<th>value</th>
<th>EPID</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>value</td>
<td>EPID</td>
<td>key</td>
</tr>
<tr>
<td>value</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

hash buckets

hash keys
Aggregation Implementation

- Library applies aggregating function to consumed data, using formerly consumed data as intermediate result
- Allows the kernel to discard the metadata contents of consumed aggregation buffers
- Allows drops to be easily eliminated in long-running aggregations
dtrace(1M) syntax

dtrace [ -i id [ aggact ]]
[ -P prov [ aggact ]]
[ -m [ prov: ] mod [ aggact ]]

aggact ⇒ [ / predicate / ]

"[" expr-list "]" = aggregating-func (arg-list)
DTrace During Boot

- Normally, consumer is a process with DTrace pseudodevice open
- However, would like to be able to trace during boot — *before* processes can run
- Introduce *anonymous* state:
  - Predicates and actions for anonymous state are specified via driver's configuration file
  - Command has option to generate configuration file
  - Anonymous state may later be *grabbed* by a consuming process
DTrace During Boot, cont.

• Use of DTrace during boot revealed that each text page for libc is retrieved from disk three times! (see 4647351)

• Using probe in page_destroy() with an appropriate predicate revealed source to be calls to ufs_flush()

• ufs_flush() is called in fsck(1M) and again before remount of root filesystem

• Fixing this is a huge win: 8 seconds on X1
dtrace(1M) syntax

dtrace
- -a - claim anonymous state
- -A - generate .conf file for anonymous tracing
Lockstat Provider

- Lockstat provider implements hot patching of synchronization primitives
- Provides “block,” “spin,” “acquire” and “release” probes for `mutex_enter()`, `rw_enter()`, etc.
- `lockstat(1M)` will be reimplemented as a DTrace consumer
- Long-standing RFEs (e.g., lock statistics per thread or per process) simply fall out
Profile Provider

- Provides unanchored probes based on profile interrupt
- Probes implemented as high level cyclics
- Currently no way to specify arbitrary rate; provider makes available ten probes with different hard-coded rates
- Arbitrary rates may be available via private consumer/provider interface
TL>0 Provider

- UltraSPARC-specific provider will be implemented to allow probes when trap level (TL) is greater than zero

- Non-trivial: `dtrace_probe()` cannot be called from TL>0 context

- Provider will use dynamic trap table interposition, as used in `trapstat(1M)`, `ttrace` and `atrace`
TL>0 Provider, cont.

- Implementation will be excruciating, but payoff is substantial:
  - Huge quantity of data available via trap table interposition
  - Will benefit enormously from DTrace’s ability to prune and coalesce data
- Using technology first developed in atrace, TL>0 provider will be able to optionally provide address traces
TL>0 Provider, Cont.

- `trapstat(1M)` will be reimplemented as a DTrace consumer
- `TRAPTRACE` will be obviated; equivalent functionality will be dynamic
- Example questions answered:
  - TLB misses per process, per page
  - Window spill traps on a per-function basis
  - All memory references in a specific function for a specific process
Probes in C Source Code

• Permit users to define probes in C source as TNF and VTRACE did, but improve syntax:

```c
TNF_PROBE_5(strategy, "io blockio", /* CSTYLED */,
            tnf_device,       device,          bp->b_edev,
            tnf_diskaddr,    block,           bp->b_lblkno,
            tnf_size,        size,           bp->b_bcount,
            tnf_opaque,      buf,            bp,
            tnf_bioflags,    flags,          bp->b_flags);
```

• Probe should look like a normal function call

• Enhance compiler to handle code generation and argument type descriptions
Traditional Approaches

• D-cache hot but inflexible:
  
  ```
  if (tracing_on)
    trace(arg1, arg2, ...);
  ```

• D-cache cold but more flexible:
  
  ```
  if (this_probe_on)
    trace(arg1, arg2, ...);
  ```

• Both implementations bloat I-cache footprint
Compiler-Assisted Approach

\[
\text{maj} = \text{getmajor}(\text{bp} \rightarrow \text{b_edev});
\]

... 

\[
\begin{align*}
\text{ldx} & \ [\%i0 + 0xa8], \ %g2 \quad ! \ \text{bp} \rightarrow \text{b_edev} \\
\text{mov} & \ -1, \ %g3 \\
\text{srl} & \ %g3, \ 0, \ %g3 \\
\text{srlx} & \ %g2, \ 0x20, \ %g2 \\
\text{and} & \ %g2, \ %g3, \ %g2 \quad ! \ %g2 = \text{getmajor}(\text{b_edev}) \\
\text{ret} & \\
\text{restore} & \ %g0, \ 0, \ %o0 \quad ! \ \text{return} \ (0);
\end{align*}
\]
Compiler-Assisted Approach

```c
TRACE("myprobe", bp);
maj = getmajor(bp->b_eudev);
...

ldx [%i0 + 0xa8], %g2  ! treat as potential call
mov −1, %g3
srl %g3, 0, %g3
srlx %g2, 0x20, %g2
and %g2, %g3, %g2  ! %g2 = getmajor(b_eudev)
...
ret
restore %g0, 0, %o0  ! return (0);
nop  ! patch point for branch
mov %i0, %o0  ! assemble probe argument
.stabs "myprobe", id, location, arg-type, ...
```
Compiler-Assisted Approach

TRACE("myprobe", bp);
maj = getmajor(bp->b_edev);
...

ba,a args1  ! branch to arg assembly
mov  -1, %g3
srl  %g3, 0, %g3
srlx %g2, 0x20, %g2
and  %g2, %g3, %g2  ! %g2 = getmajor(b_edev)
...
ret
restore %g0, 0, %o0  ! return (0);
call trampoline  ! jump to trampoline code
mov  %i0, %o0  ! assemble argument
.stabs "myprobe", id, location, arg-type, ...
C Probe Applications

- Probes in C source code can be used to convert all kernel ASSERT() instances into probes that can be enabled in production.
- Probes can also be placed in C source code to facilitate fault injection testing.
- Debug printf code that is not replaced by FBT probes can be replaced with C probes.
Basic Block Tracing

- Compiler -xa (tcov) and -xpg (gprof) options generate instrumented code that can be used for basic block coverage, profiles
- DTrace “BBT” provider can be implemented to publish block sites as DTrace probes
- Kernel modules can be compiled using these options for coverage testing
- RIP: uts/common/os/unix_bb.c
Fast-Trap Tracing

- DTrace will reserve fast-trap entry point(s) for tracing user-level activities
- DTrace “FTT” provider can also provide limited access to set of input arguments
- Traps can be generated using handcoded assembly or libc assembly wrapper
- Traps can also be inserted by user-level code generators (e.g. HotSpot JVM)
Interface Stability

• Problem: want to allow tools outside of O/N to reliably layer on top of DTrace

• Extend probe description tuple to include probe stability (i.e. attributes(5) data)

• Most of kernel is Unstable, but DDI routines and types can be Evolving

• D compiler can provide a “lint” mode to warn developers of unstable dependencies
Stable Abstractions

• We can also create more stable abstractions at DTrace API layer, e.g. proc(4) structures
• Compiler could provide \texttt{proc_t *psinfo}
• Library performs necessary transformations:

\[
\begin{align*}
\text{psinfo->pr_flag} & \Rightarrow \text{curthread->t_procp->p_flag} \\
\text{psinfo->pr_nlwp} & \Rightarrow \text{curthread->t_procp->p_lwpcnt} \\
\text{psinfo->pr_uid} & \Rightarrow \text{curthread->t_procp->p_cred->cr_uid} \\
\text{psinfo->pr_sid} & \Rightarrow \text{curthread->t_procp->p_sessp->s_sid}
\end{align*}
\]

...
Translators

- TNF users burdened with task of translating between abstraction and implementation:
  - `ls -lL /dev/dsk/*` to deal with dev_t mappings
  - PIDs/LWP ids ↔ proc_t/klwp_t addresses
  - filenames ↔ vnode_t addresses, inode numbers

- DTrace could support pluggable translators in or out of kernel to handle mappings

- D compiler can attempt to map predicate to available state by searching for translation
Trace Files

- DTrace will provide efficient access to saved trace data and data formatting features
- DTrace library will provide stable API for reading and writing trace files
- Lesson from crash dumps: put *everything* needed to interpret data in the trace file
- Files from one system, OS revision should be readable on another system or OS revision
File Format

- Header
- Indices
- EPID dictionary
- EPID data ...

File header and meta-data for each trace record generated in advance

- ASCII and DIF predicate
- EPID record description
- Data length
- Data types

Probe data can be streamed out to file using record id while tracing active

- Module name, id
- Symbol table
- String table
- CTF section

Module cache of symbol and type data can be written once tracing is complete
## Conclusion

<table>
<thead>
<tr>
<th>Feature</th>
<th>GTF</th>
<th>vtrace</th>
<th>TNF</th>
<th>KInst</th>
<th>DProbe</th>
<th>DTrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>user/kernel/merged</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>K</td>
<td>K</td>
<td>M</td>
</tr>
<tr>
<td>probe coverage</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>disabled probe effect</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>scalability</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>safety</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>extensibility</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>data filtering</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>arbitrary recording</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>self describing</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>run-time analysis</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>stock availability</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>stable abstractions</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
For more information ...

- Copies of this presentation and other documents available at http://dtrace.eng
- Questions to dtrace-interest@kiowa.eng
- E-mail dtrace-interest-admin@kiowa.eng to join the interest list
- Meetings for near-term consumers
- Project documentation, schedules, and other information forthcoming
Bibliography 1


Bibliography 2


Bibliography 3


- IBM OS/390 V2R10.0 MVS Authorized Assembler Services Reference, Volume 2 (ENFREQ-IXGWRITE) GC26-1765-13

- IBM OS/390 V2R10.0 MVS Diagnosis Procedures SY26-1082-03

- IBM OS/390 V2R10.0 MVS Diagnosis Reference SY26-1084-09

- IBM OS/390 V2R10.0 MVS IPCS Commands GC26-1754-09