Crash Dump Analysis
DTrace & SystemTap

Jakub Jermář
Martin Děcký
DTrace

- **Dynamic Tracing**
  - Observing production systems
    - Safety
    - Zero overhead if observation is not activated
    - Minimal overhead if observation is activated
    - No special debug/release builds
  - Merging and correlating data from multiple sources
    - Total observability
      - Global view of the system state
**Terminology**

- **Probe**
  - A place in code or an event which can be observed
    - If a probe is *activated* and the code is executed (or the event happens), the probe is *fired*
      - A special *script* written in *D language* is executed

- **Provider**
  - Registers probes to DTrace infrastructure
    - Does the *dirty work* of activation, tracing and inactivation

- **Consumer**
  - Consumes and postprocesses the data from fired probes
Overview

- dtrace script
- dtrace(1M)
- lockstat(1M)
- plockstat(1M)
- intrstat(1M)
- libdtrace(3LIB)
- dtrace(3D)
- DTrace
- D virtual machine
- pid
- sysinfo
- sdt
- fasttrap
- syscall
- fbt
- usdt

Consumers:
- user-space
- kernel

Providers:
- D compiler
- communication device
- user-space provider

Crash Dump Analysis – MFF UK – DTrace
DTrace history

- **31st January 2005**
  - Official part of Solaris 10
    - Released as open source ( CDDL)
      - First piece of OpenSolaris to be released
- **27th October 2007**
  - Ported to Mac OS X 10.5 (Leopard)
- **2nd September 2008**
  - Ported to FreeBSD 7.1 (released 6th January 2009)
- **21st February 2010**
  - Ported to NetBSD (only for i386, not enabled by default)
DTrace history (2)

- **Linux**
  - Cannot be directly integrated (CDDL vs. GPL)
  - Beta releases (since 2008)
    - Standalone kernel module with no modifications to core sources
    - Only some providers (fbt, syscall, usdt)
    - Development snapshots available regularly

- **SystemTap**
  - Linux-native analogy
  - A script in *SystemTap language* is converted to a C source code of a kernel module
    - Loaded and executed natively in the running kernel
    - Embedded C enabled in guru mode
DTrace history (3)

- QNX
  - Port in progress

- 3rd party software with DTrace probes
  - Apache
  - MySQL
  - PostgreSQL
  - X.Org
  - Firefox
  - Oracle JVM
  - Perl, Ruby, PHP
D language

```d
probe /predicate/ {  
  actions
 }
```

- Describe what is executed if a probe fires
  - Similar to C or AWK
    - Without dangerous constructs (branching, loops, etc.)
  - Many of the fields can be absent
    - Default predicate/action
D probes

probe /predicate/ {  
  actions
}

- A pattern consisting of fields split by colon
  - **provider:module:function:name**
    - Fields can be omitted (other are read from right to left)
      - **foo:bar** match function *foo* and name *bar* in all modules provided by all providers
    - Fields can be empty (interpreted as *any*)
      - **syscall:::** match all probes provided by the *syscall* provider
D probes (2)

probe /predicate/ { 
  actions
}

- Shell pattern matching
  - Wild characters *, ?, []
    - Can be escaped by \\n    - syscall::*lwp*:entry

- Special probes
  - BEGIN, END, ERROR
    - Implemented by dtrace provider

match all probes provided by the syscall provider, in any module, in all functions (syscalls) containing the string lwp and matching syscall entry points
D probes (3)

```dtrace
probe /predicate/ {
    actions
}
```

- Displaying all configured probes
  ```dtrace```
  ```-l```
D predicates

probe /predicate/ { 
    actions 
}

- Boolean expression guarding the actions
  - Any expression which evaluates as integer or pointer
    - Zero is considered as false, non-zero as true
    - Any D operators, variables and constants
    - Can be absent
      - Implicitly true
D actions

probe /predicate/ {  
    actions  
}

• List of statements
  • Separated by semicolon
  • No branching, no loops
  • Default action if empty
    − Usually the probe name is printed out
D types

• Basic data types reflect C language
  - **Integer types and aliases**
    - (unsigned/signed) char, short, int, long, long long
    - int8_t, int16_t, int32_t, int64_t, intptr_t, uint8_t, uint16_t, uint32_t, uint64_t, uintptr_t
  - **Floating point types**
    - float, double, long double
    - Values can be assigned, but no floating point arithmetics is implemented in DTrace
D types (2)

• Derivated and special data types
  ▪ Pointers
    – C-like pointers to other data types (including pointer arithmetics)
      • int *value; void *ptr;
        – Constant NULL is zero
      • DTrace enforces weak pointer safety
        – Invalid memory accesses are fully handled
        – However, this does not provide reference safety as in Java
D types (2)

- **Scalar arrays**
  - C-like arrays of basic data types
    - Similar to pointers, but can be assigned as a whole
    - `int values[5][6];`

- **Strings**
  - Special type descriptor `string` (instead of `char *`)
    - Can be assigned as a whole by value (`char *` copies reference)
    - Represented as NULL-terminated character arrays
    - Internal strings are always allocated as bounded
      - Cannot exceed the predefined maximum length (256 bytes)
D types (3)

- Composed data types
  - **Structures**
    - Records of several other types
    - Type declared in a similar way as in C
    - Variables must be declared explicitly
    - Members are accessed via . and -> operators

```c
struct callinfo {
    uint64_t ts;
    uint64_t calls;
};

struct callinfo info[string];

syscall::read:entry, syscall::write:entry {
    info[probefunc].ts = timestamp;
    info[probefunc].calls++;
}

END {
    printf("read %d %d\n", info["read"].ts, info["read"].calls);
    printf("write %d %d\n", info["write"].ts, info["write"].calls);
}
```
D types (4)

- Unions
- Bit-fields
- Enumerations
- Typedefs
  - All similar as in C
- Inlines
  - Typed constants
    - `inline string desc = "something";`

```
enum typeinfo {
  CHAR_ARRAY = 0,
  INT,
  UINT,
  LONG
};

struct info {
  enum typeinfo disc;
  union {
    char c[4];
    int32_t i32;
    uint32_t u32;
    long l;
  } value;
  int a : 3;
  int b : 4;
};

typedef struct info info_t;
```
DTrace operators

- Arithmetic
  - + - * / %

- Relational
  - < <= > >= == !=
    - Works also on strings (lexical comparison)

- Logical
  - && || ^^
    - Short-circuit evaluation

- Bitwise
  - & | ^ << >> ~

- Assignment
  - = += -= *= /= %= &= |= ^= <<= >>=
    - Return values as in C

- Increment and decrement
  - ++ --
DTrace operators (2)

- **Conditional expression**
  - Replacement for branching (which is absent in D)
    - `condition ? true_expression : false_expression`

- **Addressing, member access and sizes**
  - `& * . -> sizeof(type/expr) offsetof(type, member)`

- **Kernel variables access**
  - ``

- **Typecasting**
  - `(int) x, (int *) NULL, (string) expression, stringof(expr)`
DTrace variables

• Scalar variables
  ▪ Simple global variables
    - Storing fixed-size data (integers, pointers, fixed-size composite types, strings with fixed-size upper bound)
    - Do not have to be declared (but can be), duck-typing

```
BEGIN {
    /* Implicitly declare an int variable */
    value = 1234;
}

/* Explicitly declare an int variable (initial value cannot be assigned here) */
int val;

BEGIN {
    value = 1234;
}
```
**DTrace variables (2)**

- **Associative arrays**
  - Global arrays of scalar values indexed by a key
    - Key signature is a list of scalar expression values
      - Integers, strings or even a tuple of scalar types
      - Each array can have a different (but fixed) key signature
      - Declared implicitly by assignment or explicitly
        - `values[123, "key"] = 456;`
    - All values have also a fixed type
      - But each array can have a different value type
      - Declared implicitly by assignment or explicitly
        - `int values[unsigned int, string];`
DTrace variables (3)

- **Thread-local variables**
  - Scalar variables or associative arrays specific to a given **thread**
    - Identified by a special identifier `self`
    - If no value has been assigned to a thread-local variable in the given thread, the variable is considered zero-filled
  - Assigning zero to a thread-local variable deallocates it

```c
syscall::read:entry {
    /* Mark this thread */
    self->tag = 1;
}
```

```c
syscall::read:entry {
    /* Explicit declaration */
    self int tag;
    self->tag = 1;
}
```
DTrace variables (4)

- Clause-local variables
  - Scalar variables or associative arrays specific to a given probe clause
    - Identified by a special identifier `this`
    - They are not initialized to zero
      - The value is kept for multiple clauses associated with the same probe

```c
syscall::read:entry {
    this->value = 1;
}
/* Explicit declaration */
this int value;

syscall::read:entry {
    this->value = 1;
}
```
DTrace aggregations

• Variables for storing statistical data
  ▪ Storing values of aggregative data computation
    - For aggregating functions \( f(\ldots) \) which satisfy the following property
      \[
      f(f(x_0) \cup f(x_1) \cup \ldots \cup f(x_n)) = f(x_0 \cup x_1 \cup \ldots \cup x_n)
      \]
  ▪ Aggregations are declared in a similar way as associative arrays

    ```
    @values[123, "key"] = aggfunc(args);
    @_[123, "key"] = aggfunc(args); /* Simple variable */
    @[123, "key"] = aggfunc(args); /* dtto */
    ```
DTrace aggregations (2)

- Aggregation functions
  - `count()`
  - `sum(scalar)`
  - `avg(scalar)`
  - `min(scalar)`
  - `max(scalar)`
  - `lquantize(scalar, lower_bound, upper_bound, step)`
    - Linear frequency distribution
  - `quantize(scalar)`
    - Power-of-two frequency distribution
DTrace aggregations (3)

- By default aggregations are printed out in END

```plaintext
syscall:::entry {
    @counts[probefunc] = count();
}

# dtrace -s counts.d
```

```
dtrace: script 'counts.d' matched 235 probes
^C
```

```
resolvepath 8
lwp_park 10
ctime 12
lwp_sigmask 16
stat64 46
pollsys 93
p_online 256
ioctl 1695
```

#
DTrace built-in variables

• Global variables defined by DTrace
  ▪ Contain various state-dependent values
    - int64_t arg0, arg1, ..., arg9
      • Input arguments for the current probe
    - args[]
      • Typed arguments to the current probe (e.g. the syscall arguments with the appropriate types)
    - uintptr_t caller
      • Instruction pointer of the code just before firing the probe
    - kthread_t *curthread
      • Current thread kernel structure
DTrace built-in variables (2)

- string cwd
  - Current working directory
- string execname
  - Name which was used to execute the current process
- pid_t pid, tid_t tid
  - Current PID, TID
- string probeprov, probemod, probefunc, probename
  - Current probe provider, module, function and name
Using action statements

- DTrace records output to a *trace buffer*
  - Most of the action statements produce some sort of output to the trace buffer
    - `trace(expr)`
      - Output value of an expression
    - `tracemem(address, bytes)`
      - Copy given number of bytes from the given address to the buffer
    - `printf(format, ...)`
      - Output formatted strings (format options covered later)
      - Safety checks
Using action statements (2)

- `printa(aggregation)`
  - `printa(format, aggregation)`
    - Start processing `aggregation` data
      - Parallel to other execution (output can be delayed)
  - `stack()`
  - `stack(frames)`
    - Output kernel stack trace
  - `ustack()`
  - `ustack(frames)`
    - Output user space stack trace
    - Addresses are not looked up by the kernel, but by the user space consumer (later)
Using action statements (3)

- **ustack(frames, string_size)**
  - Output user space stack trace with symbol lookup (in kernel)
    - The kernel allocates `string_size` bytes for the output of the symbol lookup
    - The probe provider must annotate the user space stack with run-time symbol annotations to make the lookup possible
      - Currently only JVM (1.5 or newer) supports this

- **jstack()**

  ```
  jstack(frames)
  jstack(frames, string_size)
  ```
  - Alias for `ustack()` with non-zero default `string_size`
printf() formatting

- Conversion formats
  - `%a`
    - Pointer as kernel symbol name
  - `%c`
    - ASCII character
  - `%C`
    - Printable ASCII or escape
  - `%d, %i, %o, %u, %x`
- `%e`
  - Float as [-]d.ddd e±dd
- `%f`
  - Float as [-]ddd.ddd
- `%p`
  - Hexadecimal pointer
- `%s`
  - ASCII string
- `%S`
  - ASCII string or escape
Subroutines

- Special actions which alter the state of DTrace
  - But do not produce any output to the trace buffer
  - Are completely safe
    - Usually manipulate the local memory storage of DTrace
    - *alloca(size)
      - Allocate size bytes of scratch memory
      - The memory is released after the current clause ends
    - bcopy(*src, *dest, size)
      - Copy size bytes from outside scratch memory to scratch memory
Subroutines (2)

- *copyin(addr, size)
  - Copy size bytes from the user memory of the current process to scratch memory
- *copyinstr(addr)
  - Copy NULL-terminated string from the user memory of the current process to scratch memory
- mutex_owned(*mutex)
  - Tell whether a kernel mutex is currently locked or not
- *mutex_owner(*mutex)
  - Return the pointer to kthread_t of the thread which owns the given mutex (or NULL)
- mutex_type_adaptive(*mutex)
Subroutines (3)

- `strlen(string)`
  - Return length of a NULL-terminated string
- `strjoin(*str, *str)`
  - Concatenate two NULL-terminated strings
- `basename(*str)`
  - Return a basename of a given filename
- `dirname(*str)`
- `cleanpath(*str)`
  - Return a filesystem path without elements such as ../
- `rand()`
  - Return a (weak) pseudo-random number
Destructive actions

- Changing the state of the system
  - In a deterministic way
    - But it can be still dangerous in production environment
    - Need to be explicitly enabled using `dtrace -w`
    - `stop()`
      - Stop the current process (e.g. to dump the core or attach mdb)
    - `raise(signal)`
      - Send a signal to the current process
    - `panic()`
Destructive actions (2)

- `copyout(*buffer, addr, bytes)`
  - Store given number of bytes from a buffer to the given address
  - Page faults are detected and avoided

- `copyoutstr(string, addr, maxlen)`
  - Store at most `maxlen` bytes from a NULL-terminated string to the given address

- `system(program, ...)`
  - Execute a program as it would be executed by a shell (`program` is actually a printf() format specifier)

- `breakpoint()`
  - Induce a kernel breakpoint (if a kernel debugger is loaded, it is executed)
Destructive actions (3)

- chill(nanoseconds)
  - Spin actively for a given number of nanoseconds
  - Useful for analyzing timing bugs

- exit(status)
  - Exit the tracing session and return the given status to the consumer
Speculative tracing

- Predicates are good for filtering out unimportant probes \textbf{before} they are fired.
- But how to effectively filter out unimportant probes eventually some time \textbf{after} they are fired?
  - You can tell that you are interested in the data from a probe $n$ only after probe $n+k$ ($k > 0$) is fired.
  - \textbf{Solution: Speculatively} record all the data, but decide later whether to \textit{commit} it or not.
Speculative tracing (2)

- **speculation()**
  - Create a new speculative buffer and return its ID
  - By default the number of speculative buffers is limited to 1

- **speculate(id)**
  - The rest of the clause will be recorded to the speculative buffer given by `id`
  - This must be the first data processing action in a clause
  - Disallowed actions: aggregating, destructive

- **commit(id)**
  - Commit the speculative buffer given by `id` to the trace buffer
Provider: syscall

- Tracing of kernel system calls
  - Probes for entry and exit points of a syscall
    - Access to (typed) arguments
    - Access to the return value (on exit)
    - Access to kernel errno
    - Access to kernel variables
  - Internally uses the original syscall tracing mechanism
Provider: *fbt*

- **Function boundary tracing**
  - Probes on function entry point and (all) exit points of almost all kernel functions
    - Inlined and leaf functions cannot be traced
  - In *`entry`*
    - All typed function arguments can be accessed via `args[]`
  - In *`return`*
    - Offset of the return instruction is stored in `arg0`
    - Typed return value is stored in `args[1]`
Provider: *fbt* (2)

**How does it work?**

```plaintext
ufs_mount:
ufs_mount+1:
ufs_mount+4:
ufs_mount+0xb:
.......  
ufs_mount+0x3f3:
ufs_mount+0x3f4:
ufs_mount+0x3f7:
ufs_mount+0x3f8:
```

```
pushq %rbp
movq %rsp,%rbp
subq $0x88,%rsp
pushq %rbx
pushq %rbx
movq %rbp,%rsp
```

```
int $0x3
movq %rsp,%rbp
subq $0x88,%rsp
pushq %rbx
```

```
popq %rbx
movq %rbp,%rsp
popq %rbp
ret
```

**uninstrumented**

**instrumented**
Provider: *sdt*

- Static kernel probes
  - Probes declared on arbitrary places in the kernel code (via a macro)
  - Currently just a few of them actually defined
    - `interrupt-start`
    - `interrupt-complete`
      - `arg0` contains pointer to dev_info structure
Provider: *sdt* (2)

- How does it work?

```plaintext
squeue_enter_chain+0x1af:
xorl %eax,%eax

squeue_enter_chain+0x1b1:
nop

squeue_enter_chain+0x1b2:
nop

squeue_enter_chain+0x1b3:
nop

squeue_enter_chain+0x1b4:
nop

squeue_enter_chain+0x1b5:
nop

squeue_enter_chain+0x1b6:
movb %bl,%bh

uninstrumented

instrumented
```
Provider: proc

- Probes corresponding to process and thread life-cycle
  - Creating a process (using fork() and friends)
  - Executing a binary
  - Exiting a process
  - Creating a thread, destroying a thread
  - Receiving signals
Provider: *sched*

- Kernel scheduler abstraction probes
  - Changing of priorities
  - Thread being scheduled
  - Thread being preempted
  - Thread going to sleep
  - Thread waking up
Provider: \textit{io}

- Input/output subsystem probes
  - Starting an I/O request
  - Finishing an I/O request
  - Waiting for a device
Provider: \textit{pid}

- Tracing user space functions
  - Does not enforce serialization
    - Traced process in never stopped
    - Boundary probes similar to \textit{fbt}
      - Function \textit{entry} and \textit{return}
        - Arguments in \textit{arg0, arg1, ... arg9} are raw unfiltered int64_t values
    - Arbitrary function offset
  - User space symbol information is required to support symbolic function names
    - On Solaris, standard shared libraries contain symbol information
Other providers

• Many other providers exist
  - Application specific providers (X.Org, PostgreSQL, Firefox, etc.)
    • Via DTrace total observability you can correlate information such as which SQL transaction is generating a particular I/O load in the kernel
  - VM based providers (JVM, PHP, Perl, Ruby)
  - More kernel providers
    • Memory management provider (vminfo)
    • Network stack provider (mid)
    • Profiling provider (profile)
      • Interval-based probes
DTrace and mdb

- Accessing DTrace data from a crash dump
  - Analyzing DTrace state
    - Display trace buffers, consumers, etc.

```plaintext
> ::dtrace_state
    ADDR MINOR    PROC NAME      FILE
ccaba400     2        - <anonymous>             -
ccab9d80     3 d1d6d7e0 intrstat      cda37078
cbfb56c0     4 d71377f0 dtrace       ceb51bd0
ccabb100     5 d713b0c0 lockstat     ceb51b60
d7ac97c0     6 d713b7e8 dtrace       ceb51ab8
```
# DTrace and mdb (2)

- Displaying the contents of a trace buffer

```markdown
> ccaba400::dtrace

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>344</td>
<td>resolvepath:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>close:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>202</td>
<td>xstat:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>202</td>
<td>xstat:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>open:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>206</td>
<td>fxstat:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>186</td>
<td>mmap:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>186</td>
<td>mmap:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>186</td>
<td>mmap:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>190</td>
<td>munmap:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>344</td>
<td>resolvepath:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>216</td>
<td>memcntl:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>close:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
<tr>
<td>0</td>
<td>202</td>
<td>xstat:entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init</td>
</tr>
</tbody>
</table>
```

...
DTrace and mdb (3)

- Interpreting the results
  - The output of ::dtrace is the same as the output of dtrace utility
  - The order is always oldest to youngest within each CPU
  - The CPU buffers are displayed in numerical order (you can use ::dtrace -c cpu to show only a specific CPU)
  - Only in-kernel data which has not yet been processed by an user space consumer can be displayed
    - To keep as much data as possible in the kernel buffer, the following dtrace options can be used
      dtrace -s ... -b 64k -x bufpolicy=ring
Resources

- Richard McDougall, Jim Mauro, Brendan Gregg: *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*

- *Solaris Dynamic Tracing Guide*
  - [http://docs.sun.com/app/docs/doc/817-6223](http://docs.sun.com/app/docs/doc/817-6223)