DTrace

D++: DTrace language extensions

Matt.Ahrens@delphix.com
Ryan Stone

dtrace.conf(2012)

- Themes
  - Userland CTF & dynamic translators
  - Interesting work done with dtrace
  - dtrace for linux, freebsd, MacOS, PS vita O_o
  - ZFS dtrace provider
  - The D language
- http://blog.delphix.com/ahl/tag/dtrace-conf
- http://wiki.smartos.org/display/DOC/dtrace.conf+Schedule

DTrace in FreeBSD

- Enabled by default in HEAD
- MIPS implementation of SDT probes (gonzo@)
- ip, tcp, udp providers in progress (markj@)
- pjd@ bringing in features from Illumos
- DTrace Toolkit ported to FreeBSD (gnn@)
- D script for gathering schedgraph data
  - people.freebsd.org/~rstone/dtrace/schedgraph.d
- Kernel tracing is solid; userland tracing remains a work in progress

D is a (safe) power tool

- Some D concepts (predicates) are unfamiliar to procedural programmers
- Some simple ideas require many steps
- These steps are (for the most part) fundamentally necessary but tedious
- Solution: syntactic sugar

if/else in D

```
vdev_queue_pending_remove:entry
/stringof(args[1]->io_spa->spa_name) == $$1 &&
    args[1]->io_type == ZIO_TYPE_READ/
{
        @bytes_read = sum(args[1]->io_size);
}

vdev_queue_pending_remove:entry
/stringof(args[1]->io_spa->spa_name) == $$1 &&
    args[1]->io_type == ZIO_TYPE_WRITE &&
    args[1]->io_bookmark.zb_level != -2/
{
        @bytes_written = sum(args[1]->io_size);
}
```

if/else in D++

```
vdev_queue_pending_remove:entry
{
        if (stringof(args[1]->io_spa->spa_name) == $$1) {
                if (args[1]->io_type == ZIO_TYPE_READ) {
                        @bytes_read = sum(args[1]->io_size);
                } else if (args[1]->io_type == ZIO_TYPE_WRITE &&
                    args[1]->io_bookmark.zb_level != 2) {
                        @bytes_written = sum(args[1]->io_size);
                }
        }
}
```

if/else converted to D

```
dtrace:::ERROR{ self->_DPP_error = 0x1; }
::vdev_queue_pending_remove:entry{ self->_DPP_error = 0x0; }

::vdev_queue_pending_remove:entry /!self->_DPP_error/
{ this->_DPP_condition1 = 0x1 && stringof(args[1]->io_spa->spa_name) == $$1; }

::vdev_queue_pending_remove:entry /!self->_DPP_error/
{ this->_DPP_condition2 = this->_DPP_condition1 && args[1]->io_type == ZIO_TYPE_READ; }

::vdev_queue_pending_remove:entry /(!self->_DPP_error) && this->_DPP_condition2/
{ @bytes_read = sum(args[1]->io_size); }

::vdev_queue_pending_remove:entry /!self->_DPP_error/
{ this->_DPP_condition3 = this->_DPP_condition1 && !this->_DPP_condition2; }

::vdev_queue_pending_remove:entry /!self->_DPP_error/
{ this->_DPP_condition4 =
    this->_DPP_condition3 && args[1]->io_type == ZIO_TYPE_WRITE && args[1]->io_bookmark.zb_level != 2; }

::vdev_queue_pending_remove:entry /!self->_DPP_error && this->_DPP_condition4/
{ @bytes_written = sum(args[1]->io_size); }
```

while loop in D

```
pid$target::zprop_free_list:entry { ll_user = arg0; printf("-- START --");
}

pid$target::zprop_free_list:entry /ll_user != NULL/ {
        ll_kern = copyin(ll_user, sizeof(zprop_list_t));
        printf("%d", ll_kern->pl_prop);
        ll_user = ll_kern->pl_next;
}

pid$target::zprop_free_list:entry /ll_user != NULL/ {
        ll_kern = copyin(ll_user, sizeof(zprop_list_t));
        printf("%d", ll_kern->pl_prop);
        ll_user = ll_kern->pl_next;
}

/* More copies go here */

pid$target::zprop_free_list:entry { printf("-- END --"); }
```

while loop in D++

```
pid$target::zprop_free_list:entry
{
        ll_user = arg0;
        printf("-- START --");
        while10 (ll_user != NULL) {
                ll_kern = copyin(ll_user, sizeof(zprop_list_t));
                printf("%d", ll_kern->pl_prop);
                ll_user = ll_kern->pl_next;
        }
        printf("-- END --");
}
```

while loop converted to D

```
pid$target::zprop_free_list:entry { ll_user = arg0; printf("-- START --"); }

pid$target::zprop_free_list:entry {this->_DPP_condition2 = (ll_user != 0x0);}
pid$target::zprop_free_list:entry /this->_DPP_condition2/ {
        ll_kern = copyin(ll_user, sizeof(zprop_list_t));
        printf(" %d", ll_kern->pl_prop);
        ll_user = ll_kern->pl_next;
}

pid$target::zprop_free_list:entry {this->_DPP_condition2 = (ll_user != 0x0);}
pid$target::zprop_free_list:entry /this->_DPP_condition2/ {
        ll_kern = copyin(ll_user, sizeof(zprop_list_t));
        printf(" %d", ll_kern->pl_prop);
        ll_user = ll_kern->pl_next;
}

/* ... repeat 8 more times ... */

pid$target::zprop_free_list:entry { printf("-- END --"); }
```

printing nvlists in D++

- manually-managed stack for nested nvlists
- print "function" implemented as C preprocessor macro
  - implement "inline" D++ functions?
- also straightforward to implement **`nvlist_lookup_{uint64,string,etc}`**
- definitely need to increase **`dtrace_dof_maxsize`**!

- pseudocode follows
  - assume everything starts with "this->"

```
elem = list->nvl_first;
while20 (elem != NULL) {
    printf("%s", stringof(elem->name));
    if (elem->type == DATA_TYPE_UINT64) {
        printf(": %u\n", *(uint64_t *)elem->valuep);
        elem = elem->next;
    } else if (elem->type == DATA_TYPE_STRING) {
        printf(": %s\n", stringof((char *)elem->valuep));
        elem = elem->next;
    } else if (elem->type == DATA_TYPE_NVLIST) {
        stack[curframe].elem = elem;
        stack[curframe].list = list;
        curframe++;
        list = (nvlist_t *)elem->nvi_valuep;
        elem = list->nvl_first;
    }
    if (elem == NULL && curframe > 0) {
        curframe--;
        elem = stack[curframe].elem;
        list = stack[curframe].list;
    }
}
if (elem != NULL)
    printf("<iteration exhausted>\n");
```

entry_* variables in D

```
spa_sync:entry
{
        self->spa = args[0];
        self->txg = args[1];
        self->start = timestamp;
}

spa_sync:return
/self->start/
{
        printf("%s(%s, %u) took %ums",
            probefunc, self->spa->spa_name, self->txg,
            (timestamp - self->start)/1000/1000);
        self->spa = 0;
        self->txg = 0;
        self->start = 0;
}
```

Hope this function isn't recursive!

entry_* variables in D++

```
spa_sync:return
{
        printf("%s(%s, %u) took %ums",
            probefunc,
            entry_args[0]->spa_name, entry_args[1],
            (timestamp - entry_timestamp)/1000/1000);
}
```

entry_* converted to D

```
::spa_sync:entry
{
        self->_DPP_entry_args1[stackdepth] = args[1];
        self->_DPP_entry_args0[stackdepth] = args[0];
        self->_DPP_entry_timestamp[stackdepth] = timestamp;
}

::spa_sync:return
{ this->_DPP_condition1 = 0x1 && self->_DPP_entry_timestamp[stackdepth]; }

::spa_sync:return
/this->_DPP_condition1/
{
        printf("%s(%s, %u) took %ums",
            probefunc, self->_DPP_entry_args0[stackdepth]->spa_name,
            self->_DPP_entry_arg1[stackdepth],
            (timestamp - self->_DPP_entry_timestamp[stackdepth]) / 1000 / 1000);
}

::spa_sync:return
{
        self->_DPP_entry_arg1[stackdepth] = 0x0;
        self->_DPP_entry_args0[stackdepth] = 0x0;
        self->_DPP_entry_timestamp[stackdepth] = 0x0;
}
```

callers[] in D

```
resolvepath:entry,traverse:entry
{
        self->trace = 1;
}

zrl_add:entry
/ self->trace != 0 /
{
        @[stack()] = count();
}

resolvepath:return,traverse:return
{
        self->trace = 0;
}
```

What if resolvepath and traverse are both on the stack?

callers[] in D++

Now a one-liner:

```
zrl_add:entry / callers["resolvepath,traverse"] / {@[stack()] = count()}
```

- The value of callers[] is a count, not just a toggle
  - Useful for examining recursive functions.
- **`callers["resolvepath,traverse"]`**
  - Either function must be in the stack
- **`callers["resolvepath"] && callers["traverse"]`**
  - Both functions must be in the stack
- Possible performance considerations
  - Each element means two more probes, be cognizant of enabled probe effect.

callers[] converted to D

```
::resolvepath:entry,
::traverse:entry
{ ++self->_DPP_callers1; }

::zrl_add:entry
{ this->_DPP_condition1 = 0x1 && self->_DPP_callers1; }

::zrl_add:entry
/this->_DPP_condition1/
{ @_[stack()] = count(); }

::resolvepath:return,
::traverse:return
/self->_DPP_callers1/
{ --self->_DPP_callers1; }
```

How does it do that?

Changes are primarily in libdtrace

- Create parse tree
  - new nodes for "`if`", "`while`"
- Transform parse tree to remove D++
  - create new clauses
  - swap out entry_*, callers[]
- Finish compiling the (now strictly D) parse tree
- No kernel changes

How can I use it?

- https://github.com/ahrens/dpp
  - fork of illumos
  - 100% libdtrace, should be easy to port
  - still a little rough around the edges
    - needs code cleanup
    - a few known bugs
- To see the D generated from your D++
  - `dpp -x tree=8 ...`
- Bugs?
  - email `matt@delphix.com` your d++ script

What next?

- "for" and "do" loops; "break", "continue"(?)
- **`#pragma D option defaultscope=local`**
  - bare variables will have probe-local scope
    - i.e. implicit "this->"
  - use "global->" for global variables
  - use "thread->" for thread-local variables
- "inline" functions to replace preprocessor
- better error diagnosability?
  - `dtrace: error on enabled probe ID 463 (ID 33804: fbt:zfs:`**`put_nvlist:entry`**`): invalid address (0x0) in action #1 at DIF offset 36`
- entry_delta_{ns,us,ms,sec}

What next? (continued)

- Predicate Scoping
  - Applying a predicate to multiple clauses specified in a block:

```
/callers["spa_sync"]/
{
        spa_*:entry
        {
                trace(probefunc);
        }

        spa_*:return
        {
                trace(arg1);
        }
}
```