DTrace and Erlang: a new beginning

basho

Scott Lystig Fritchie
scott@basho.com
@slfritchie
https://github.com/slfritchie

Outline

The Visibility Problem

DTrace in Erlang: past and present

Tour of DTrace and SystemTap

DTrace in Erlang: the future

FIRST:

The Visibility Problem

riak

Customer's Environment

Application: messaging

Front end (stateless): custom app (Java)

Back end (stateful): Riak (Erlang)

DevOps group monitors end-to-end latency

99th and 100th percentile has SLA limits

Riak

Highly scalable, highly available distributed database

Distributed computing platform

For the purpose of this case study...

... Riak is the "backend database"

99th Percentile Latency Alarms

Alarm rings once every 1-4 days

SLA: latency alarm should *never* ring

Not reproducible in Basho's lab or customer's lab.

Must test in production environment

Step One: Measure in Parallel

Customers alarms are end-to-end latency

Need to measure Riak end-to-end latency

Does Riak's 99th latency match the app's 99th?

```
case correlated(App, Riak) of true ->
    get_busy(Basho)
```

Step One: Measure Riak's FSM latency

Riak "get" and "put" operations use FSMs

Each FSM coordinates data replication within Riak

Use Erlang's built-in tracing mechanism

Result: yes, occasionally FSM takes a long time

Step Two: Measure Riak's vnode latency

Riak vnode: local data storage on a cluster member

Measure what?

Vnode operations rates (gets/second, puts/seconds)
Vnode mailbox sizes

Results:

No "hot" vnodes: evenly distributed
No unusual Erlang process mailbox sizes

Step Three: Measure Bitcask's latency

Bitcask: local key-value storage manager, used by vnodes

Bitcask: high throughput, predictable low latency

Key index stored entirely in RAM

1 get = 0 or 1 `open(2)` calls + 1 `pread(2)`
1 put = 0 or 1 `open(2)` calls + 1 `write(2)`

Step Three: Measure Bitcask's latency

Measure what?

Latency of each op (get, put, ...)
Get op: found vs. not-found latency

Results:

No unusual not-found latency
Yes, occasionally unusual get latency

Measurement Methods Change As Old Data Is Examined

Measurement Method One

Built-in Erlang tracing mechanism

No Riak code change

Custom trace event receiver/formatter

Offline analysis of trace events using R

Measurement Method Two

Erlang tracing

Riak code changes, sometimes daily

More custom event receiver & offline R analysis

Basho staff doing hot code upgrades on customer production systems

Time

Bursts of activity: 1-4 times/week for 1-4 hours each

Basho senior developer(s) involved almost every time

Daily code upgrades for new measurements were common

Wall clock time: about 5 weeks

This Situation Must Change, Need More Tools!

Measurement Toolkit

`cprof` & `eprof` & `fprof`

Erlang tracing infrastructure

Custom event generators & handlers, e.g. `gen_event`

Toolkit Problems

Profilers: not suitable for production environments

Erlang tracing: beware of event tsunami and memory explosion!

Custom code: can make customer's Ops & QA staff nervous

Change Is Not Always Good

```
extern App_t *packaged, *running;

while (is_equal(packaged, running)) {
    hot_patch(running,
              get_new_experiment());
}
kill(get_qa_department_pid(), SIGURG);
```

SECOND:

DTrace: Past and Present

First, A Word About Erlang's Tracing Mechanism

Erlang's Tracing

It's really cool

It's actually awesome

It's easy to overwhelm a production system

It works better if code is structured appropriately

What is DTrace?

Probe: code annotation that "fires" when enabled

Probes are dynamic: *absent* when disabled
e.g. 2 NOOP instruction placeholder

Kernel hot-patches running executables to enable probes

Probe event gathering infrastructure

Why DTrace?

Unified: kernel & user space probes are the same

Unified: same tool for both kernel & user tracing

Disabled probes: zero overhead

Unified: same pkg for production & tracing+debug

Unified: C, C++, Java, Python, MySQL, PostgreSQL, ...

Enabled probes: extremely small overhead

Easy to add event probes to any app

Easy event post-processing events: D scripting language

Stable: never crash the kernel or app

DTrace and Erlang, 2008

Garry Bulmer presents at EUC 2008

Initial D probes added to R12 virtual machine

Driver allows Erlang code to fire probes

Bulmer & Becker pass project to Ericsson

DTrace, 2009-2011

No public work available.

DTrace, 2011

Basho: Riak

CouchBase: Membase & CouchBase

Erlang Solutions: RabbitMQ & other Erlang apps

DTrace 2011: 3-way merge

`https://github.com/slfritchie/otp`

Three branches of DTrace work on Erlang R14 and R15

Scott merges work of Dustin Sallings and Michal Ptaszek

Autoconf magic for OS X, Solaris, and Linux (via DTrace->SystemTap compatibility layer)

Erlang DTrace Provider

Processes: spawn, exit, hibernate, scheduled, ...

Messages: send, queued, received, exit signals

Memory: GC minor & major, proc heap grow & shrink

Data copy: within heap, across heaps

Function calls: function & BIF & NIF, entry & return

Network distribution: monitor, port busy, output events

Ports: open, command, control, busy/not busy

Drivers: callback API 100% instrumented

`efile_drv.c` file I/O driver: 100% instrumented

Fun Stuff Yet To Do

SMP scheduler: queue length, work stealing, ...

Locking: attempt, wait, acquire, release

ETS table events

ETS and SMP + locking

`inet_drv` driver (TCP, UDP, SCTP)

More sequential trace token tracing

True dynamic probe creation

Benchmark-related BIFs (see HiPE source)

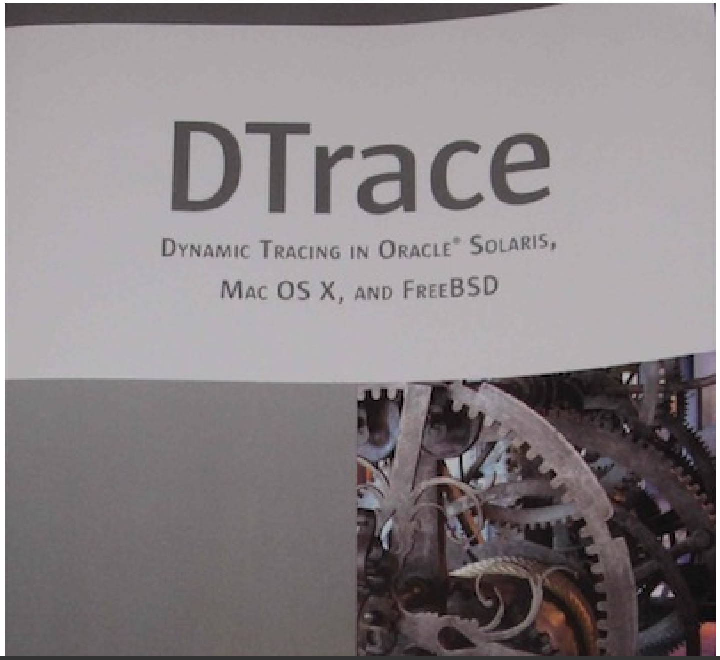
Task queuing (used by drivers)

... and many, many more ...

THIRD:

Tour of DTrace and SystemTap

The DTrace Book



Tour?

Not Enough Time!

This tour bus will take a short-cut

Process Events

```
/* dtrace -qs /path/to/this/script.d */
erlang*:::process-spawn {
    printf("pid %s mfa %s\n",
           copyinstr(arg0), copyinstr(arg1))
}

erlang*:::process-exit {
    printf("pid %s reason %s\n",
           copyinstr(arg0), copyinstr(arg1))
}
```

Births and Deaths?

```
/* dtrace -s /path/to/this/script.d */
BEGIN {
    spawns = exits = 0
}

erlang*:::process-spawn { spawns++ }
erlang*:::process-exit { exits++ }

profile:::tick-1sec {
    printf("Spawns %d exits %d", spawns, exits);
    spawns = exits = 0
}
```

Latency of sending a message -> receiving process is scheduled?

Message Send -> Proc Scheduled Latenc

```
/* Example from Dustin Sallings */
/* dtrace -s /path/to/this/script */
BEGIN { printf("Press control-c to print histogram\n") }

erlang*:::message-send {
    sent[copyinstr(arg1)] = timestamp
}

erlang*:::process-scheduled
/ sent[copyinstr(arg0)] /
{
    @t = quantize(timestamp - sent[copyinstr(arg0)]);
    sent[copyinstr(arg0)] = 0
}
```

Message Send -> Proc Scheduled Latenc

```
sbb# dtrace -qs /tmp/send-to-sched.d
Press control-c to print histogram
^C
   value  ------------- Distribution ------------- count
    8192 |                                         0
   16384 |@@@@@@@@@@@@@@@@@@@@@@@@@                119
   32768 |@@@@@@@@@@@@                             58
   65536 |@@@                                      13
  131072 |                                         1
  262144 |                                         0

%% Run on an idle Erlang VM on power-saving laptop:
%%     application:start(sasl).
%%     application:start(crypto).
```

Func Calls Longer Than 1 Millisecond

```
erlang*:::function-entry {
    self->start[copyinstr(arg0),
                copyinstr(arg1)] = timestamp
}

erlang*:::function-return
/self->start[copyinstr(arg0),
             copyinstr(arg1)] != 0 &&
  timestamp - self->start[copyinstr(arg0),
                          copyinstr(arg1)] > 1000000 /
{
    duration = timestamp - (self->start[copyinstr(arg0),
                                        copyinstr(arg1)]);
    printf("CALL: %s - %s - %d nanoseconds\n",
          copyinstr(arg0), copyinstr(arg1), duration)
}
```

Call Latency

Function calls longer than X?

Tail call elimination can hit here

... Win some, lose some

BIF calls longer than X?

NIF calls longer than X?

Function call starts on CPU X,

Call returns on CPU Y,

$X \,/\!= Y$

```
/* dtrace -qs /path/to/this/script.d */
erlang*:::function-entry {
    _cpu[copyinstr(arg0), copyinstr(arg1), arg2] = cpu+1
}

erlang*:::function-return
/ _cpu[copyinstr(arg0), copyinstr(arg1), arg2] != 0 &&
  _cpu[copyinstr(arg0), copyinstr(arg1), arg2] != cpu+1 /
{
    depth = arg2;
    proc = copyinstr(arg0);
    mfa = copyinstr(arg1);
    callcpu = _cpu[proc, mfa, depth] - 1;
    printf("%s %s @ stack depth %d: CPU %d -> %d\n",
           proc, mfa, depth, callcpu, cpu);
}

erlang*:::function-return
{ _cpu[copyinstr(arg0), copyinstr(arg1), arg2] = 0 }
```

CPU X -> CPU Y Results

```
<0.91.0> erl_eval:add_bindings/2 @ stack depth 4: CPU 4 -> 2
<0.91.0> erl_eval:add_bindings/2 @ stack depth 4: CPU 2 -> 6
```

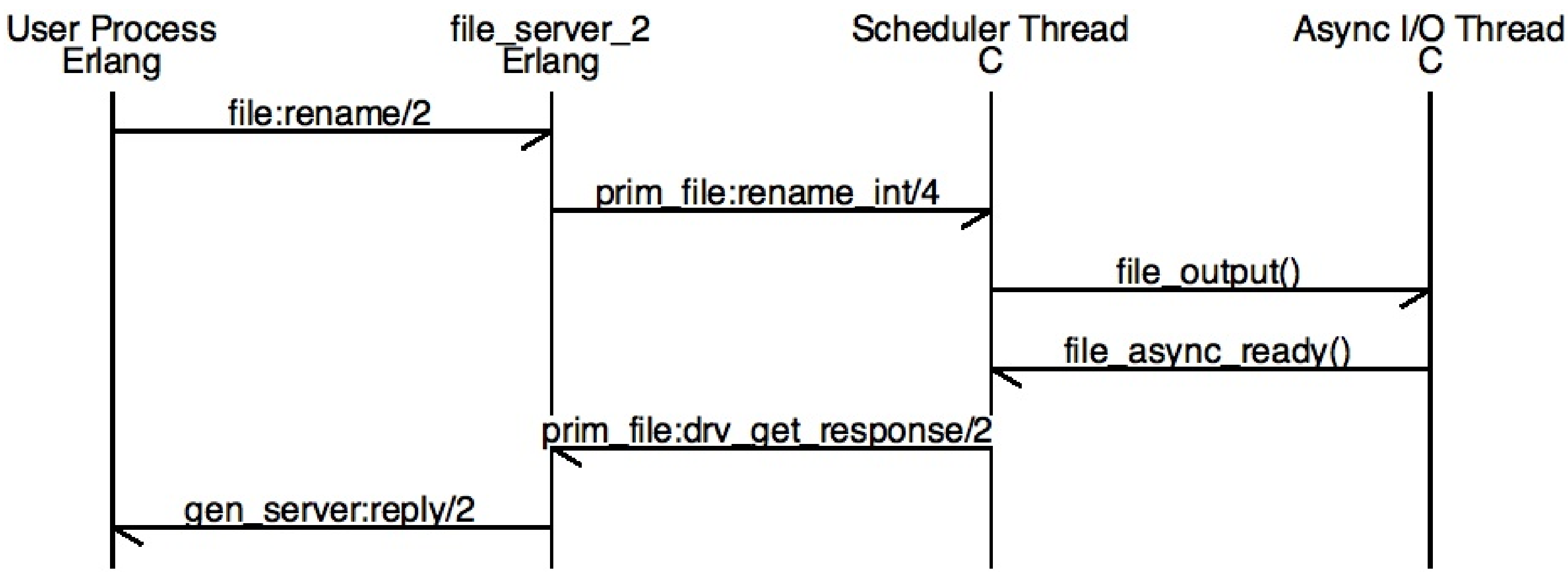
Largest I/O Worker Thread Pool Queue?

```
/* dtrace -s /path/to/this/script.d */
BEGIN { biggest = 0 }

erlang*:::aio_pool-add
/arg1 > biggest/ { biggest = arg1 }

profile:::tick-10sec {
    printf("Largest async work queue size:
           biggest);
    biggest = 0;
}
```

`file:rename/2` Message Path



Probes From Erlang Code

```
%% erlang*:::user_trace-i4s4 disabled
> dtrace:p("Hello, world!").
false

%% erlang*:::user_trace-i4s4 enabled
> dtrace:p(42).
true

%% Up to 4 integers and 4 iolist() args
dtrace:p(1, 2, 4, 8, "a", "b", "c", "and d").
```

SystemTap

Linux's answer to DTrace

Not well documented or supported by most Linux distributions

Erlang VM annotations compile cleanly

Therefore, it works.
If not, please tell me.

FOURTH (and last):

Erlang and DTrace: The Future

Question: The End Goal Is?

Answer: Erlang/OTP R15

Universal adoption

OS X, Solaris, FreeBSD 9, Linux

Lots of Work Remains

More probes: 60 isn't enough, really!

Support `ustack()` somehow

Truly dynamic probes from Erlang code

More helpful D scripts, examples and real/deployable stuff.

Testing, testing, bullet-proofing, and more testing....

Where's the Code Right Now?

`https://github.com/slfritchie/otp`

Main dev branch is `dtrace-experiment+michal2`

Branches `dtrace-r14b04` and `dtrace-r15` have periodic merges from main dev branch

Community Support!

More mailing list response from RabbitMQ users than `erlang-questions@erlang.org`

More community support -> warmer & fuzzier feelings @ Ericsson.

Thanks For Your Time!

Source: `https://github.com/slfritchie/otp`

Branch: `dtrace-experiment+michal2`

Email: `scott@basho.com`

Twitter: `@slfritchie`

GitHub: `slfritchie`

Appendix / Extra Material

`pid` provider: automatic probes!

```
# dtrace -ln 'pid78917:::entry' | grep beam.smp | wc -l
63636

# dtrace -ln 'pid78917::copy_*:entry' | grep beam.smp
ID     MODULE              FUNCTION  NAME
203607 beam.smp            copy_ref entry
204445 beam.smp         copy_object entry
204447 beam.smp         copy_struct entry
204448 beam.smp        copy_shallow entry
205794 beam.smp     copy_object_rel entry
205810 beam.smp        copy_to_comp entry
206658 beam.smp       copy_utf8_bin entry
208792 beam.smp          copy_block entry
```

`pid` provider: automatic values

`pid<PID>:::entry`

The function's arguments are `(uint64_t) arg0`, `(uint64_t) arg1`, and so on through `(uint64_t) argN`.

`pid<PID>:::return`

The program counter is `(uint64_t) arg0`
The return value is `(uint64_t) arg1`

`pid` provider: alas, no interpretation!

```
/*
 * copy_ref is the first function mentioned on the
 * previous slide.  It was chosen pseudo-randomly to
 * illustrate a point about the `pid` provider.
 * The pid provider cannot pretty-format `ref` or return
 * value, or tell you if `ref` is a tuple/list/port/...
 * or tell you any diagnostic info about the heap that
 * `*hp` points to.  If you want to know those kinds
 * of things, then you need to add a DTrace probe.
 */
static Eterm copy_ref(Eterm ref, Eterm *hp)
{
    RefThing *ptr = ref_thing_ptr(ref);
    memcpy(hp, ptr, sizeof(RefThing));
    return (make_internal_ref(hp));
}
```

Provider: messages

```
probe message__send(char *sender, char *receiver,
                    uint32_t size,
                    int token_label, int token_previous,
                    int token_current);
probe message__send__remote(char *sender, char *node_name,
                            char *receiver, uint32_t size,
                            /* 3 token-related ints... */);
probe message__queued(char *receiver, uint32_t size,
                      uint32_t queue_len,
                      /* 3 token-related ints... */);
probe message__receive(char *receiver, uint32_t size,
                       uint32_t queue_len,
                      /* 3 token-related ints... */);
```

Provider: function calls

```
/* @param p the PID (string form) of the process
 * @param mfa the m:f/a of the function
 * @param depth the stack depth                      */
probe function__entry(char *p, char *mfa, int depth);
probe function__return(char *p, char *mfa, int depth);
probe bif__entry(char *p, char *mfa);
probe bif__return(char *p, char *mfa);
probe nif__entry(char *p, char *mfa);
probe nif__return(char *p, char *mfa);
```

Provider: processes

```
/* @param p the PID (string form) of newly scheduled process
 * @param mfa the m:f/a of the function it should run next */
probe process__scheduled(char *p, char *mfa);
probe process__unscheduled(char *p);
probe process__hibernate(char *p, char *mfa);
probe process__port_unblocked(char *p, char *port);
probe process__heap_grow(char *p,
                         int old_size, int new_size);
probe process__heap_shrink(char *p,
                           int old_size, int new_size);
```

Provider: network distribution

```
/* @param node the name of the reporting node
 * @param what the type of event, e.g., nodeup, nodedown
 * @param monitored_node the name of the monitored node
 * @param type the type of node, e.g., visible, hidden
 * @param reason the reason term, e.g., normal,
 *        connection_closed, or term()                    */
probe dist__monitor(char *node, char *what,
                    char *monitored_node,
                    char *type, char *reason);
probe dist__port_busy(char *node, char *port,
                      char *remote_node, char *pid);
probe dist__port_not_busy(char *node, char *port,
                          char *remote_node);
```

Provider: ports

```
probe port__open(char *process, char *port_name, char *port);
probe port__command(char *process, char *port,
                    char *port_name, char *command_type);
probe port__control(char *process, char *port,
                    char *port_name, int command_no);
probe port__exit(char *process, charf *port, char *port_name,
probe port__connect(char *process, char *port,
                    char *port_name, char *new_process);
probe port__busy(char *port);
probe port__not_busy(char *port);
```

Provider: async worker thread pool

```
/* @param port the Port (string form)
 * @param new queue length                                      */
probe aio_pool__add(char *, int);
probe aio_pool__get(char *, int);
/* @param thread-id number of the scheduler Pthread
 * @param tag number: {thread-id, tag} uniquely
 *                    names a driver operation
 * @param user-tag string
 * @param command number
 * @params: 2 optional strings, 4 optional ints, port name */
probe efile_drv__entry(int, int, char *, int,
                       char *, char *,
                       int64_t, int64_t, int64_t, int64_t,
                       char *);
```

DTrace Probe in `copy.c`

```
/* Simple DTrace probe, is implicitly disabled */
Eterm copy_struct(Eterm obj, Uint sz, Eterm** hpp,
                  ErlOffHeap* off_heap)
{
    char* hstart;
    Uint hsize;
    ... many other local vars omitted ...

    if (IS_CONST(obj))
        return obj;

    DTRACE1(copy_struct, (int32_t)sz);
    ...
```

DTrace Probe in `copy.c`

```
/* Skip expensive computation if probe disabled */
Eterm
copy_object(Eterm obj, Process* to)
{
    Uint size = size_object(obj);
    Eterm* hp = HAlloc(to, size);
    Eterm res;

    if (DTRACE_ENABLED(copy_object)) {
        char proc_name[64];
        erts_snprintf(proc_name, sizeof(proc_name),
                      "%T", to->id);
        DTRACE2(copy_object, proc_name, size);
    }
    res = copy_struct(obj, size, &hp, &to->off_heap);
...
```