DTrace and Erlang: a new beginning

basho

Scott Lystig Fritchie

scott@basho.com

@slfritchie

https://github.com/slfritchie

Mug Shot

Outline

The Visibility Problem

DTrace in Erlang: past and present

DTrace and Erlang

Erlang's Future with DTrace

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

Measurement Methods Change As Old Data Is Examined

Measurement Method #1

Built-in Erlang tracing mechanism

No Riak code change

Custom trace event receiver/formatter

Offline analysis of trace events using R

Measurement Method #2

Erlang tracing

Riak code changes, sometimes daily

More custom event receiver & offline R analysis

Basho staff doing hot code upgrades on customer production systems

Measurement 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, We Need More Tools!

Erlang Toolkit Problems

Erlang 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

What Makes Erlang Special?

Virtual machine (not the JVM or CLR or ...)

Processes for fault isolation (not threads)

Message passing (not shared state)
Messaging & monitoring across an IP network

Processes migrate to different threads/CPUs

Hot code loading

Fault Tolerance Demo (Elevator Controller)

If time permits at the end of this presentation.

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

Only 1 consumer => only 1 CPU core

It works better if code is structured appropriately

Best with lots of small/short functions

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

RELEASE Project

EUC funded, multi-year project to scale Erlang to 10,000+ CPU cores

DTrace 2011: 3-way merge

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, FreeBSD, and Linux (via DTrace->SystemTap compatibility layer)

Erlang DTrace Provider
60 probes and more

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

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:
Examples

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);
...
```

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).
```

Function call starts on CPU X,

Call returns on CPU Y,

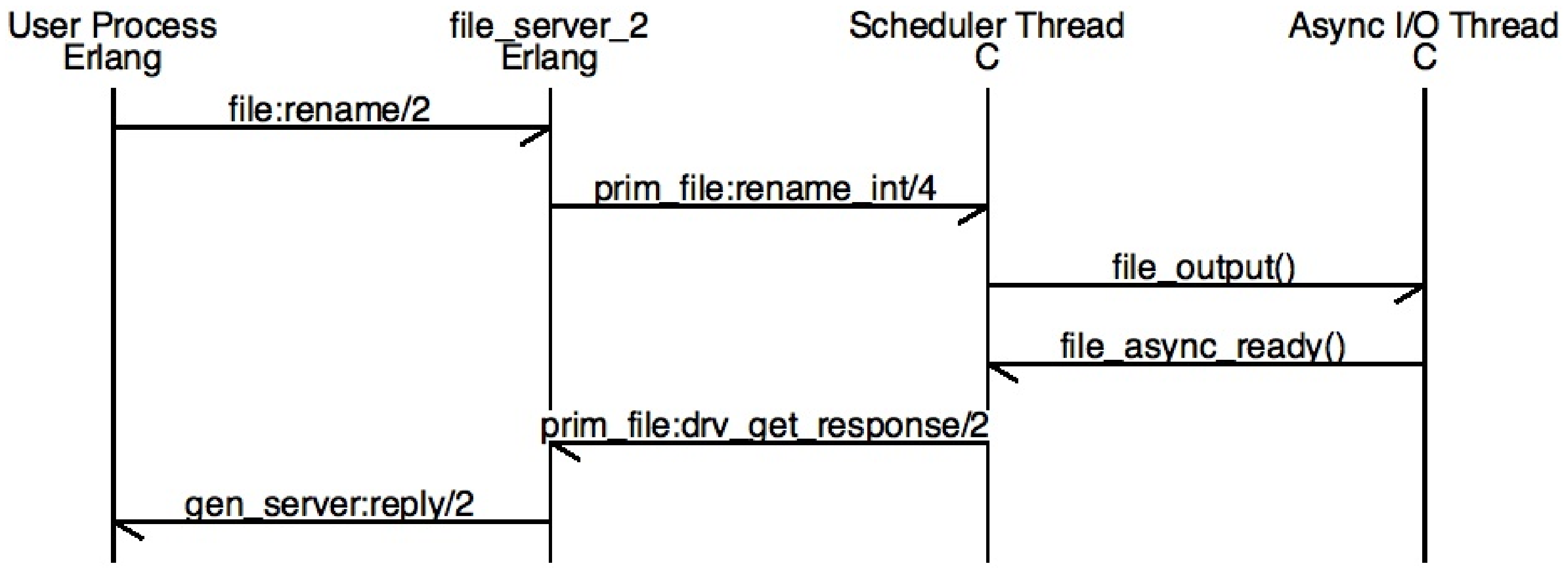
X /= Y

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

`file:rename/2` Message Path



Probes From Erlang Code

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

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

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

FOURTH (and last):

Erlang and DTrace: The Future

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.

Where's the Code Right Now?

Erlang/OTP R15B01 will be released this week.

For DTrace + R14B04:

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

`dtrace-r14b04` branch

Thanks For Your Time!

basho

Email: `scott@basho.com`

Twitter: `@slfritchie`

GitHub: `slfritchie`

... any time remainging for an Erlang fault isolation demo?

Appendix / Extra Material

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 *);
```