Ruby 2.0.0 & DTrace

DTrace

- Dynamic System Tracing.
- Real time monitoring of Applications through to Kernel.
- Originated in Solaris 10.
- Now in Mac OS X, FreeBSD, SmartOS, OpenIndiana, etc.
- (but not Windows or Linux*)

DTrace:::why?

- Production monitoring (no special debug profiling stuff added to your application)
- Dynamic: Tracing probes only triggered when someone is listening.
- Observability of the whole system: ruby method calls, disk i/o, file i/o, network i/o, virtual memory, and so on.

demo: dtrace -l | less; | wc

DTrace:::scripts

- Uses its own “D” scripting language. Programs have 1+ clauses:

```
probe-description
/predicate/
{ action; }
```

- Example:

```
ruby*:::raise
/pid == 12345/
{ @num = count(); }
```

demo: count.d

DTrace:::clauses

- probe-description: four components separated by colons: “provider:module:function:name”
- predicate: a condition required for action to happen.
- action: one or more lines of C-like code. No control flow statements.

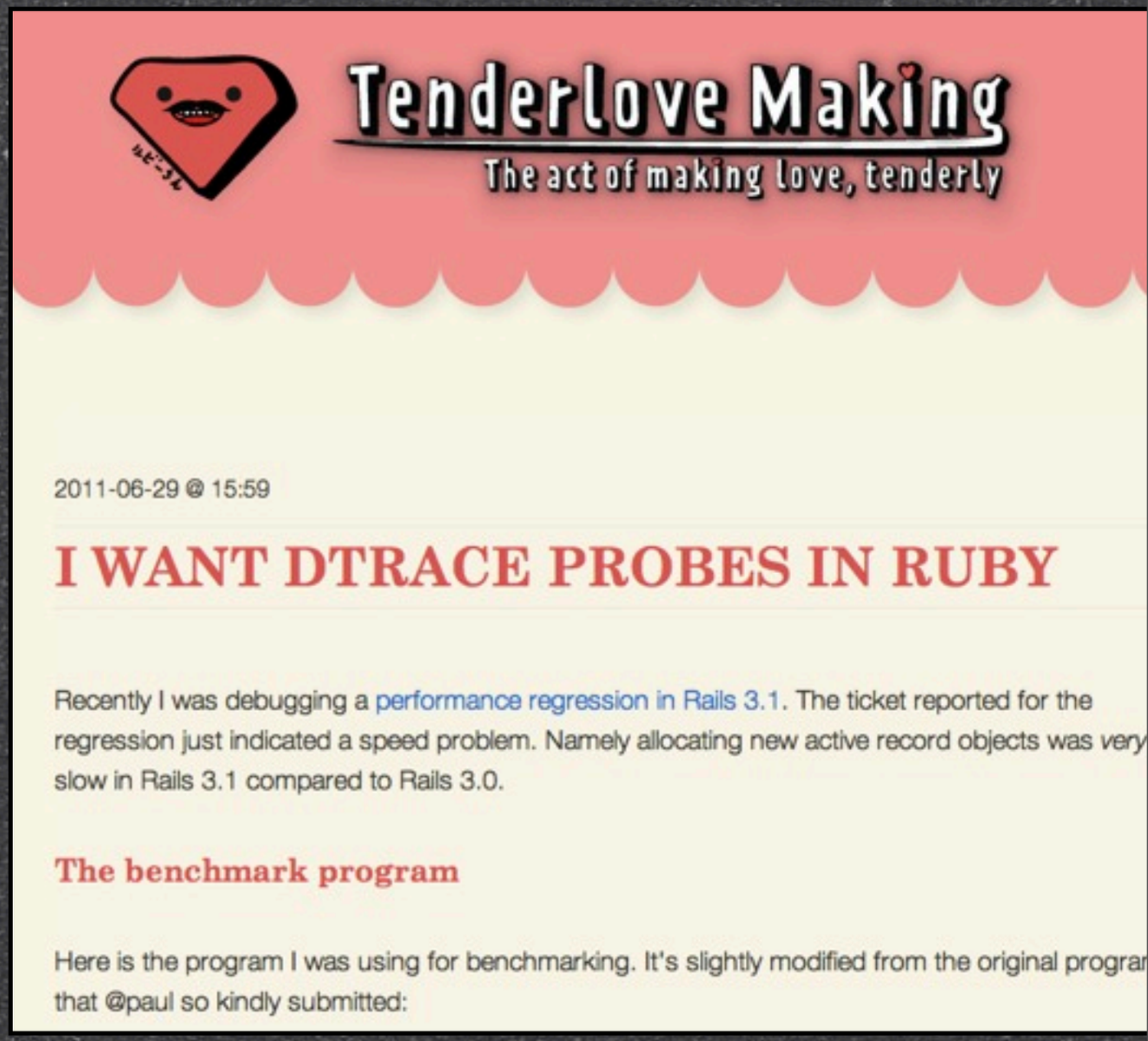
DTrace:::toolkit

- Great place to start: Brendan Gregg's DTraceToolkit.
- Over 200 example scripts.
- http://www.brendangregg.com/dtrace.html#DTraceToolkit
- execsnoop, iosnoop, rwsnoop

demo: execsnoop, iosnoop, rwsnoop

DTrace::ruby

- Apple's ruby had DTrace integration for ages.
- Aaron Patterson (Tenderlove) made some custom probes in 2011.
- Added DTrace probes to ruby core, included ruby 2.0.0.



Tenderlove Making

The act of making love, tenderly

2011-06-29 @ 15:59

I WANT DTRACE PROBES IN RUBY

Recently I was debugging a performance regression in Rails 3.1. The ticket reported for the regression just indicated a speed problem. Namely allocating new active record objects was very slow in Rails 3.1 compared to Rails 3.0.

The benchmark program

Here is the program I was using for benchmarking. It's slightly modified from the original program that @paul so kindly submitted:

http://tenderlovemaking.com/2011/06/29/i-want-dtrace-probes-in-ruby.html

DTrace::probes1

```
// old ruby probes (included in Apple's ruby 1.8.7):

provider ruby {
        probe function-entry(classname, methodname, filename, lineno);
        probe function-return(classname, methodname, filename, lineno);
        probe raise(errinfo, filename, lineno);
        probe rescue(filename, lineno);
        probe line(filename, lineno);
        probe gc-begin();
        probe gc-end();
        probe object-create-start(object, filename, lineno);
        probe object-create-done(object, filename, lineno);
        probe object-free(object);
    };
```

demo: system irb && dtrace -l | grep ruby

DTrace::probes2

```
// new ruby 2.0.0 dtrace probes
provider ruby {
        probe method-entry(classname, methodname, filename, lineno);
        probe method-return(classname, methodname, filename, lineno);
        probe cmethod-entry(classname, methodname, filename, lineno);
        probe cmethod-return(classname, methodname, filename, lineno);
        probe require-entry(requiredfile, filename, lineno);
        probe require-return(requiredfile, filename, lineno);
        probe find-require-entry(requiredfile, filename, lineno);
        probe find-require-return(requiredfile, filename, lineno);
        probe load-entry(loadedfile, filename, lineno);
        probe load-return(loadedfile, filename, lineno);
        probe raise(classname, filename, lineno);
        probe object-create(classname, filename, lineno);
        probe array-create(length, filename, lineno);
        probe hash-create(length, filename, lineno);
        probe string-create(length, filename, lineno);
        probe parse-begin(sourcefile, lineno);
        probe parse-end(sourcefile, lineno);

        // optional (configured when compiling ruby):
        probe gc-mark-begin();
        probe gc-mark-end();
        probe gc-sweep-begin();
        probe gc-sweep-end();
    };
```

demo: ruby-2.0.0 irb && dtrace -l | grep ruby

DTrace:::example 1

- Example #1 Real time logging of exceptions:
  - ruby*:::raise

```
{
  printf("'%s' at %s:%d",
          copyinstr(arg0),
          copyinstr(arg1), arg2);
}
```

demo: example1.rb, example1.d

DTrace:::example 2

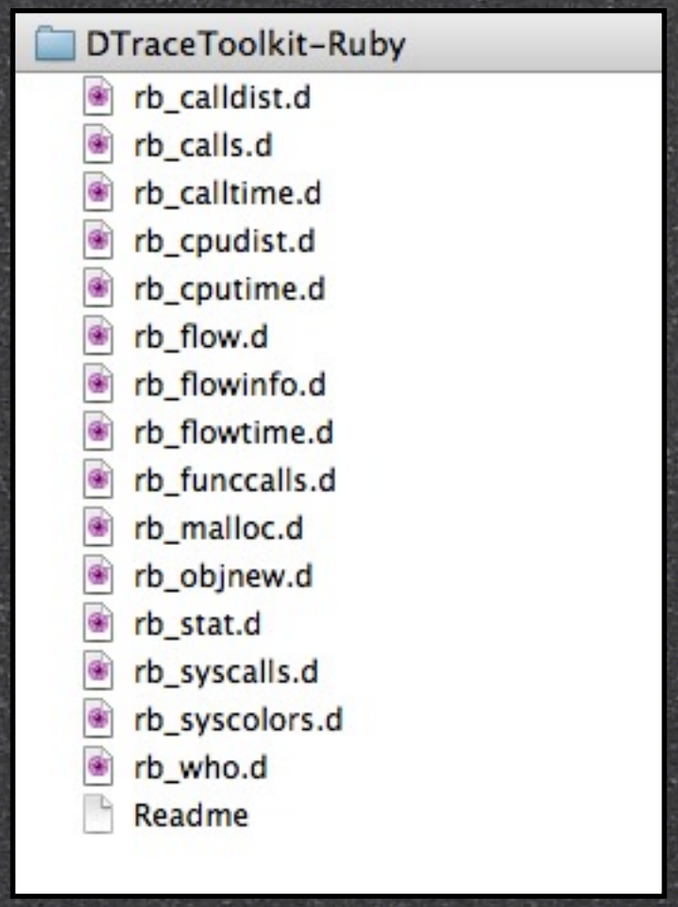
- Example #2 counting method calls.
- Uses aggregate `count()` function:

```
@sources[copyinstr(arg2)] = count();
```

```
demo: `dtrace -s example2.d -c 'ruby example2.rb'`
```

DTrace::toolkit:ruby2

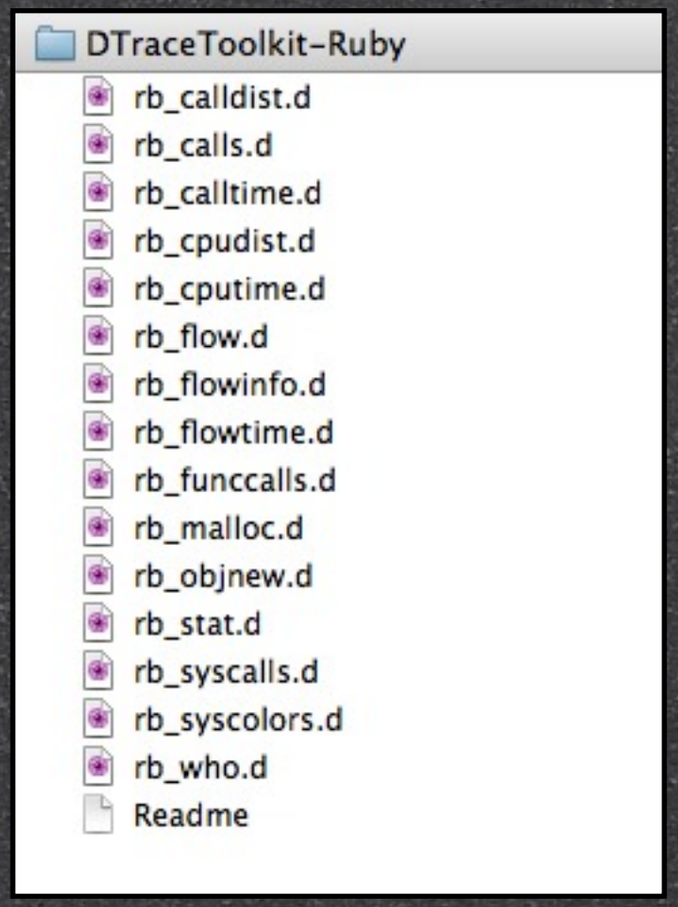
- Updated the toolkit examples to use ruby 2.0.0 probes
- Mostly work...



demo: `dtrace -s rb_who.d` , rb_objnew.d, rb_calls.d, rb_calltime.d

DTrace::toolkit:ruby2

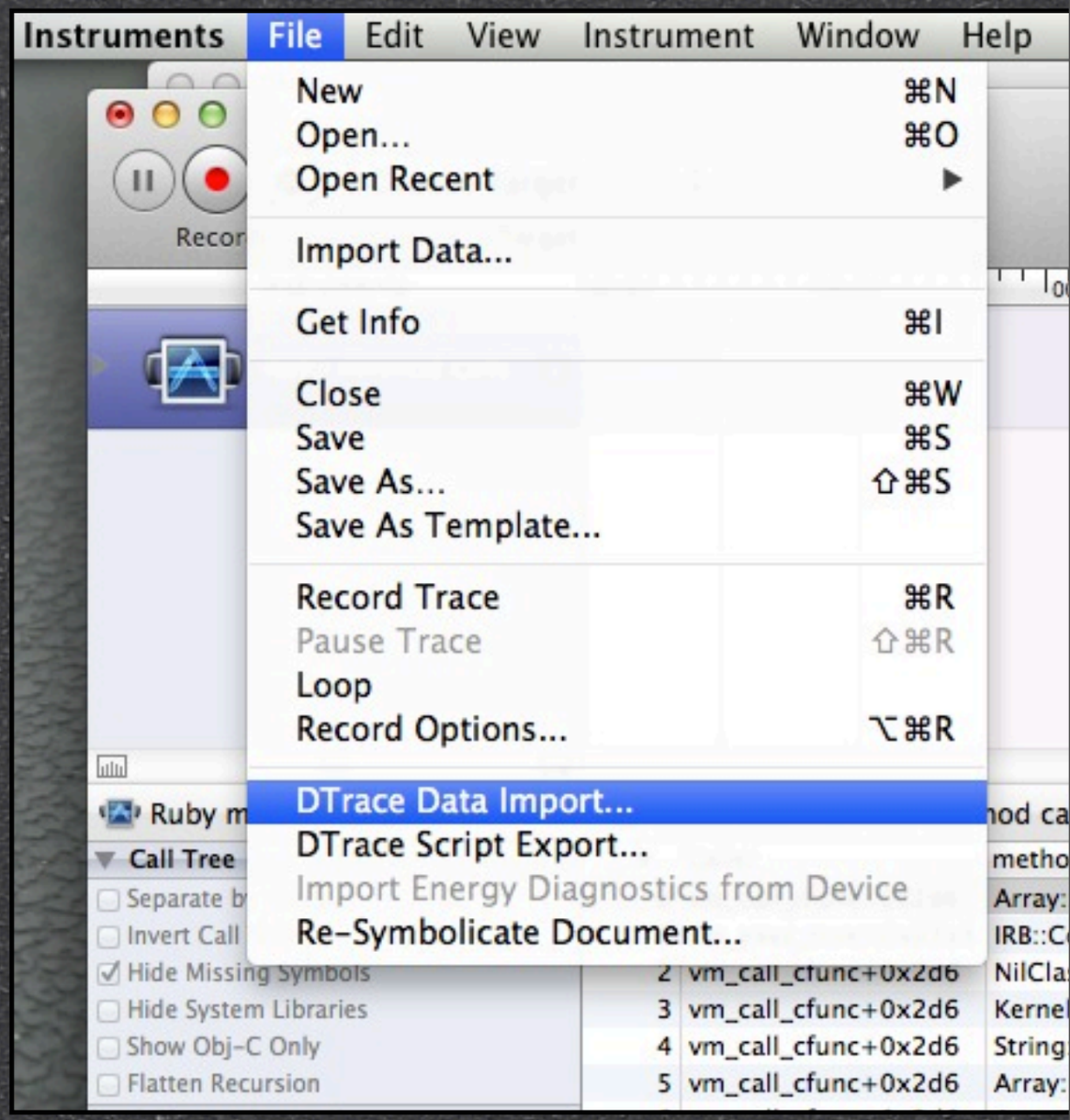
- Let's look at some ruby and system calls together...
- "rb_syscalls.d"
  "rb_syscolors.d"




```
demo: `dtrace -s rb_syscalls.d -c 'ruby ../examples/example2.rb'
```

DTrace:::instruments

- Xcode's Instruments tool uses DTrace.
- It can assist drinking from the fire-hydrant.
- Record directly, or export scripts to run remotely, capture data, analyse in Instruments later.



demo: `example3.trace` in Instruments

DTrace::custom:probes

- Custom probes can be written. Require
  - “C” code to invoke them.
  - Simple C library, with FFI wrapped class
  - Could use it to turn tracing on/off at specific times.

demo: custom.c, custom.rb, custom_trace.d

References

- http://dtrace.org/guide/preface.html
- https://wikis.oracle.com/display/DTrace/Documentation
- http://www.tablespace.net/quicksheet/dtrace-quickstart.html
- http://tenderlovemaking.com/2011/06/29/i-want-dtrace-probes-in-ruby.html
- https://github.com/ruby/ruby/commit/4c740bae97f31e16cc6d15753868b6bd49e330d0
- http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/CreatingCustomInstruments/CreatingCustomInstruments.html#//apple_ref/doc/uid/TP40004652-CH8-SW1