# DTrace and Python

Jesús Cea Avión
jcea@jcea.es
@jcea
httpS://www.jcea.es/
httpS://blog.jcea.es/

# Jesús Cea Avión

- Programming in Python since 1996 (Python 1.4).
- Core Developer since 2008 (Python 2.6 and 3.0).
- Founder of Python Madrid, Python Vigo and Python España association.
- Solaris user since 1990, SysOP since 1996.
- Consultant and freelance always searching for new and interesting challenges. Hire me! :-)

## Overview of this talk

- What is DTrace?
  - Quick & dirty overview.
- Relevance for Python.
  - Probes in the interpreter.
- Examples tracing a Python program.
- Examples tracing the entire stack, including OS.
- Future. Help!
  - More probes.
  - Porting to other DTrace supported platforms.

## Python

- You already know about this…

## DTrace

- Comprehensive full system dynamic tracing framework developed by Sun Microsystems for Solaris.
- Virtually zero performance impact when not in use.
- Safe to use in production.
- Available on Solaris and derivatives, FreeBSD, NetBSD, Mac OS X, Oracle Linux.

# DTrace

- Operating system, libraries and programs can define “probes”:

```
# dtrace -l | wc -l
259438
```

- Can fire at machine language function call/return.
- Can fire at arbitrary machine language instruction.
- SAFE to use in production.
- (Almost) zero performance impact when not in use.

## DTrace

- DTrace language is safe, read-only. **(*)**
- Probes everywhere:
  - Syscall, virtual memory, CPU scheduler, network, locks, disk...
  - High level probes.
  - Dedicated providers. For instance, Python.
  - Dynamic providers. For instance, sampling profile.
  - Synthetic providers: process defined probes.

# DTrace

- Simple language to activate arbitrary probes and execute code when the event “fires”.
- Speculative tracing.
- It doesn’t require process collaboration, but helpful.
- Native aggregation functions.
- Associative arrays.
- Excellent documentation.
- DevOps paradise.

## DTrace examples

Show me the processes doing “fsync()” calls:

```
# dtrace -l -P syscall | wc -l
471    ← Include entry/return + heading

# dtrace -n 'syscall::fdsync:entry {printf("%s",
execname);}'
dtrace: description 'syscall::fdsync:entry ' matched
1 probe
CPU     ID                    FUNCTION:NAME
  0  58858                     fdsync:entry lmtp
  [...]
  4  58858                     fdsync:entry lmtp
  7  58858                     fdsync:entry cleanup
```

# DTrace examples

Show me “fsync()” duration stats:

```
# dtrace -n 'syscall::fdsync:entry {self->t =
timestamp;} syscall::fdsync:return {@t =
quantize(timestamp-self->t);}'
dtrace: description 'syscall::fdsync:entry ' matched
2 probes
^C
           value  ------------- Distribution ------------- count
          262144 |                                         0
          524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      7
         1048576 |                                         0
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         0
        16777216 |@@@@@                                    1
        33554432 |                                         0
```

# DTrace examples

Peek inside a process:

```
# dtrace -l -n pid25590:::entry | wc -l
21204
# dtrace -l -n pid25590:libssl.so.1.0.0::entry | wc -l
649
# dtrace -n \
pid9498:libssl.so.1.0.0:ssl_verify_cert_chain:entry
dtrace: description
'pid9498:libssl.so.1.0.0:ssl_verify_cert_chain:entry
' matched 1 probe
```

# DTrace examples

- Sampling profiler:
```
# dtrace -n 'profile-997 /pid == 9912/ {jstack();}'
```

- Show me CPU use of a particular process:

```
# dtrace -n 'BEGIN {oncpu=0;totalcpu=0;} profile-
997 /pid == 10354/ {oncpu+=1;} profile-997
{totalcpu+=1;} END {printf("%d %d", totalcpu,
oncpu);}'
dtrace: description 'BEGIN ' matched 4 probes
^C
CPU     ID                    FUNCTION:NAME
  3      2                         :END 78972 9871
```

# DTrace probes in Python

- Instrumented interpreter for better information:

```
# dtrace -l -P python9134
   ID   PROVIDER            MODULE                          FUNCTION NAME
59421 python9134 libpython3.5m.so.1.0                PyEval_EvalFrameEx function-entry
59422 python9134 libpython3.5m.so.1.0                PyEval_EvalFrameEx function-return
59423 python9134 libpython3.5m.so.1.0               _PyGC_CollectNoFail gc-done
59424 python9134 libpython3.5m.so.1.0                      PyGC_Collect gc-done
59425 python9134 libpython3.5m.so.1.0                        gc_collect gc-done
59426 python9134 libpython3.5m.so.1.0             collect_with_callback gc-done
59427 python9134 libpython3.5m.so.1.0               _PyGC_CollectNoFail gc-start
59428 python9134 libpython3.5m.so.1.0                      PyGC_Collect gc-start
59429 python9134 libpython3.5m.so.1.0                        gc_collect gc-start
59430 python9134 libpython3.5m.so.1.0             collect_with_callback gc-start
59431 python9134 libpython3.5m.so.1.0                   subtype_dealloc instance-delete-done
59432 python9134 libpython3.5m.so.1.0                   subtype_dealloc instance-delete-start
59433 python9134 libpython3.5m.so.1.0               PyType_GenericAlloc instance-new-done
59434 python9134 libpython3.5m.so.1.0               PyType_GenericAlloc instance-new-start
59435 python9134 libpython3.5m.so.1.0                PyEval_EvalFrameEx line
```

## DTrace probes in Python

- Current probes:
  - line
  - function-entry
  - function-return
  - gc-start
  - gc-done
  - instance-new-start
  - instance-new-done
  - instance-delete-start
  - instance-delete-done

## Examples of DTrace in Python

- Tell me where a particular library call is done:

```
 # dtrace -n 'python12042:::function-entry
 /copyinstr(arg0)=="/usr/local/lib/python3.5/ssl.py
 " && copyinstr(a1)=="getpeercert"/ {jstack(100,
 10000);}' | grep '\['
dtrace: description 'python12042:::function-entry '
matched 1 probe
 [ python3.5/ssl.py:805 (getpeercert) ]
 [ urllib3/connection.py:259 (connect) ]
 […]
 [ requests/adapters.py:376 (send) ]
 […]
 [ requests/api.py:53 (request) ]
 […]
```

## Examples of DTrace in Python

- Tell me how long are garbage collections:

```
# dtrace -n 'python12042:::gc-start {self->t =
timestamp;} python12042:::gc-done {printf("%uus",
timestamp-self->t);}'
dtrace: description 'python12042:::gc-start '
matched 8 probes
CPU     ID                    FUNCTION:NAME
  5 101930    collect_with_callback:gc-done 8480us
  5 101930    collect_with_callback:gc-done 3062us
  5 101930    collect_with_callback:gc-done 1891us
```

- What Python function fires most GC?
- How frequent are GC?

## Examples of DTrace in Python

- Poor man memory “Leak” detector:

```
# dtrace -n 'python12042:::instance-new-start
{@[copyinstr(arg0)] = sum(1);}
python12042:::instance-delete-done
{@[copyinstr(arg0)] = sum(-1);}'
dtrace: description 'python12042:::instance-new-
start ' matched 2 probes
^C
  […]
  _GeneratorContextManager                            0
  socket                                              0
  BufferedSubFile                                     2
  FeedParser                                          2
  HTTPMessage                                         2
```

## Examples of DTrace in Python

- Trace a Apache MOD_WSGI process:

```
# dtrace -n 'python25589:::function-entry
/copyinstr(arg1)=="application"/ {self->f=1;}
python25589:::function-return
/copyinstr(arg1)=="application"/ {self->f=0;}
python25589:::function-entry /self->f/ {printf("%s",
copyinstr(arg1));}'
dtrace: matched 3 probes
CPU     ID                    FUNCTION:NAME
  3   4350 PyEval_EvalFrameEx:function-entry application
  3   4350 PyEval_EvalFrameEx:function-entry salida
```

- What operating system calls are being slow?
- Where are we being preempt by the OS? For how long? Why?

## Examples DTrace: Python + OS

- Show me where I am being blocked (synchronization object):

```
# dtrace -n 'sched:::sleep /pid==14857/
{printf("[BLOCKED] %d\n", tid); jstack();}' | grep
"\["
dtrace: matched 7 probes
  2   5025                   cv_block:sleep [START] 2
 [ python3.5/threading.py:293 (wait) ]
 [ python3.5/queue.py:164 (get) ]
 [ python3.5/concurrent/futures/thread.py:64 (_worker) ]
 [...]
```

- CPU accounting per Python Thread.
- What processes are stealing my CPU?
- Examine lock contention, even GIL.

## Examples DTrace: Python + OS

- What code is actually accessing the disk, not getting data from cache?

```
# dtrace -n 'io:::start /pid==14857/ {jstack();}'
dtrace: description 'io:::start ' matched 6 probes
CPU     ID                    FUNCTION:NAME
  6   5049              bdev_strategy:start
    libc.so.1`_read+0x15
    libpython3.5m.so.1.0`_Py_read+0x4b
    libpython3.5m.so.1.0`_io_FileIO_readall_impl.isra.8+0xeb
    libpython3.5m.so.1.0`PyCFunction_Call+0xca
    libpython3.5m.so.1.0`PyObject_Call+0x68
    libpython3.5m.so.1.0`PyObject_CallMethodObjArgs+0xa2
    libpython3.5m.so.1.0`_io__Buffered_read+0x47f
    libpython3.5m.so.1.0`PyCFunction_Call+0xd9
    libpython3.5m.so.1.0`PyEval_EvalFrameEx+0xa051
    [ <stdin>:1 (<module>) ] ← open("file", "rb").read()
    libpython3.5m.so.1.0`_PyEval_EvalCodeWithName+0xb31
    libpython3.5m.so.1.0`PyEval_EvalCode+0x30
    libpython3.5m.so.1.0`PyRun_InteractiveOneObject+0x1a5
    libpython3.5m.so.1.0`PyRun_InteractiveLoopFlags+0x7d
    libpython3.5m.so.1.0`PyRun_AnyFileExFlags+0x40
    libpython3.5m.so.1.0`Py_Main+0xe21
    python3.5`main+0x170
    python3.5`_start+0x80
```

## Notice:

- You don’t modify the source code. You don’t even need source code access. No collaboration.
  - If you have OS source code, you are GOD!.
- You enable the tracing surgically, when you need it and for the time you need it, from a separate terminal.
- The process continues unaltered, in production.
- Exploratory tracing: hypothesis and fast validation.
- Full system visibility.

## More: Python USDT

*(Userland Statically Defined Tracing)*

- Your python code can define high level probes:
  - client connect, request start, job enqueued, download completed, ...
- Activate logging surgically, on demand, with the daemon running undisturbed.
- You can create individual entry/return probes per function/method with “@fbt” decorator.
- BAD: Stale? code, no documentation. Partial 3.x.

# More: Python USDT

*(Userland Statically Defined Tracing)*

```
Python 2.7.11 (dtrace-issue13405_2.7:8c5948409bbe,
Mar  3 2016, 04:49:13)
[GCC 5.3.0] on sunos5
Type "help", "copyright", "credits" or "license" for
more information.
>>> import os
>>> from usdt.tracer import fbt
>>> @fbt
... def example(v) :
...   pass
...
>>> os.getpid()
24793
>>> example("hello world!")
```

## More: Python USDT

*(Userland Statically Defined Tracing)*

```
# dtrace -l -P python-fbt24793
   ID   PROVIDER            MODULE    FUNCTION NAME
59327 python-fbt24793    fbt         example entry
59328 python-fbt24793    fbt         example return

# dtrace -n 'python-fbt24793::example:* {}'
dtrace: description 'python-fbt24793::example:* '
matched 2 probes
CPU     ID                    FUNCTION:NAME
  5  59327                    example:entry
  5  59328                   example:return
```

## Future:

- Support all DTrace platforms. Sprint tomorrow!
- Add more Python probes in the interpreter and C módules:
  - GIL, Threading module, import machinery...
- Python programs should be able to create personalized dynamic probes. **DONE: Python-USDT.**
- Challenge: integrate with mainstream CPython.

## Performance

- When not enabled, performance hit is **VERY** low:

```
                           DISABLED              ENABLED
0xfee9f79a <+2954>:  jne    0xfee9ede3
0xfee9f7a0 <+2960>:  xor    %eax,%eax
0xfee9f7a2 <+2962>:  nop                          int3
0xfee9f7a3 <+2963>:  nop
0xfee9f7a4 <+2964>:  nop
0xfee9f7a5 <+2965>:  test   %eax,%eax
0xfee9f7a7 <+2967>:  mov    -0x60(%ebp),%edx
0xfee9f7aa <+2970>:  jne    0xfeea840e
```

- Current Python USDT implemented in Python, performance hit even when probes are not enabled. Python 2.7, function call+return: x143.

# DTrace Sprint

**tomorrow**

**March, 13th**

Main target:

Correctly support FreeBSD, NetBSD and Mac OS X.

## Additional References

- Python documentation and code:
    https://www.jcea.es/artic/python_dtrace.htm
- General documentation:
  https://en.wikipedia.org/wiki/DTrace
  http://dtrace.org/guide/preface.html
  http://dtrace.org/blogs/
  https://wiki.freebsd.org/DTrace/One-Liners
  http://dtracebook.com/index.php/Main_Page
- Python USDT:
  https://pypi.python.org/pypi/usdt/
  https://github.com/nshalman/python-usdt/
  https://github.com/chrisa/libusdt
- Linux:
  https://github.com/dtrace4linux/linux
  https://docs.oracle.com/cd/E37670_01/E37355/html/ol_dtrace.html

## Questions?

- What is “Speculative Tracing”?
- Stack traces and Mac OS X.

  https://www.mail-archive.com/dtrace-discuss@opensolaris.org/msg04668.html
- What is needed to integrate with mainline CPython?.
  - Other interpreters?
- SystemTap synergies.
- DTrace in Linux?

# ¡Thank you!

Jesús Cea Avión
jcea@jcea.es
@jcea
httpS://www.jcea.es/
httpS://blog.jcea.es/