Blazing Performance with Flame Graphs

Brendan Gregg
An Interactive Visualization for Stack Traces

Flame Graph

Function: `mysqld` filesort (108,672 samples, 31.19%)
My Previous Visualizations Include

- Latency Heat Maps (and other heat map types), including:

  ![Latency Heat Map Example](image)

- Quotes from LISA'13 yesterday:
  - "Heat maps are a wonderful thing, use them" – Caskey Dickson
  - "If you do distributed systems, you need this" – Theo Schlossnagle
- I did heat maps and visualizations in my LISA'10 talk
Audience

- This is for developers, sysadmins, support staff, and performance engineers
  - This is a skill-up for everyone: beginners to experts
- This helps analyze all software: kernels and applications
G’Day, I’m Brendan

Recipient of the LISA 2013 Award for Outstanding Achievement in System Administration! (Thank you!)

Work/Research: tools, methodologies, visualizations

Author of Systems Performance, primary author of DTrace (Prentice Hall, 2011)

Lead Performance Engineer @joyent; also teach classes: Cloud Perf coming up: http://www.joyent.com/developers/training-services
Joyent

• High-Performance Cloud Infrastructure
  • Public/private cloud provider
• OS-Virtualization for bare metal performance
• KVM for Linux guests
• Core developers of SmartOS and node.js
• Office walls decorated with Flame Graphs:
Agenda: Two Talks in One

1. CPU Flame Graphs
   - Example
   - Background
   - Flame Graphs
   - Generation
   - Types: CPU

2. Advanced Flame Graphs
   - Types: Memory, I/O, Off-CPU, Hot/Cold, Wakeup
   - Developments

SVG demos: https://github.com/brendangregg/FlameGraph/demos
CPU Flame Graphs
Example
Example

• As a short example, I’ll describe the real world performance issue that led me to create flame graphs
• Then I’ll explain them in detail
Example: The Problem

- A production MySQL database had poor performance
- It was a heavy CPU consumer, so I used a CPU profiler to see why. It sampled stack traces at timed intervals
- The profiler condensed its output by only printing unique stacks along with their occurrence counts, sorted by count
- The following shows the profiler command and the two most frequently sampled stacks...
### Example: CPU Profiling

```bash
# dtrace -x ustackframes=100 -n 'profile-997 /execname == "mysqld"/ {
    @[ustack()] = count(); } tick-60s { exit(0); }

```

dtrace: description 'profile-997' matched 2 probes

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75195</td>
<td>:tick-60s</td>
</tr>
</tbody>
</table>

```
[...] libc.so.1`__priocnt1set+0xa libc.so.1`getparam+0x83 libc.so.1`pthread_getschedparam+0x3c libc.so.1`pthread_setschedprio+0x1f mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x9ab mysqld`_Z10do_commandP3THD+0x198 mysqld`handle_one_connection+0x1a6 libc.so.1`_thrp_setup+0x8d libc.so.1`_lwp_start 4884

mysqld`_Z13add_to_statusP17system_status_varS0_+0x47 mysqld`_Z22calc_sum_of_all_statusP17system_status_var+0x67 mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x1222 mysqld`_Z10do_commandP3THD+0x198 mysqld`handle_one_connection+0x1a6 libc.so.1`_thrp_connection+0x8d libc.so.1`_lwp_start 5530
```
```
Example: CPU Profiling

```bash
# dtrace -x ustackframes=100 -n 'profile-997 /execname == "mysqld"/ {
    @[ustack()] = count(); } tick-60s { exit(0); }'
dtrace: description 'profile-997 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
    1  75195                        :tick-60s
[...]
lib.so.1`__priocntlset+0xa
lib.so.1`getparam+0x83
lib.so.1`pthread_getschedparam+0x3c
lib.so.1`pthread_setschedprio+0x1f
mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x9ab
mysqld`_Z10do_commandP3THD+0x198
mysqld`handle_one_connection+0x1a6
lib.so.1`_ thrp_setup+0x8d
lib.so.1`_ lwp_start
4884
mysqld`_Z13add_to_statusP17system_status_varS0_ +0x47
mysqld`_Z22calc_sum_of_all_statusP17system_status_var+0x67
mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x1222
mysqld`_Z10do_commandP3THD+0x198
mysqld`handle_one_connection+0x1a6
lib.so.1`_ thrp_setup+0x8d
```

Profiling Command (DTrace)

Stack Trace

# of occurrences
Example: Profile Data

• Over 500,000 lines were elided from that output ("[…]"

• Full output looks like this...
Example: Profile Data
Example: Profile Data

First Stack

Size of One Stack

27,053 Unique Stacks

Last Stack
Example: Profile Data

• The most frequent stack, printed last, shows CPU usage in `add_to_status()`, which is from the “show status” command. Is that to blame?

• Hard to tell – it only accounts for < 2% of the samples

• I wanted a way to quickly understand stack trace profile data, without browsing 500,000+ lines of output
To understand this profile data quickly, I created visualization that worked very well, named “Flame Graph” for its resemblance to fire (also as it was showing a “hot” CPU issue).
Example: Flame Graph

Same profile data
Example: Flame Graph

Same profile data

The "show status" Stack

Flame Graph

Where CPU is really consumed

One Stack Sample

All Stack Samples
Example: Flame Graph

- All data in one picture
- Interactive using JavaScript and a browser: mouse overs
- Stack elements that are frequent can be seen, read, and compared visually. Frame width is relative to sample count
- CPU usage was now understood properly and quickly, leading to a 40% performance win
Background
Background: Stack Frame

- A stack frame shows a location in code
- Profilers usually show them on a single line. Eg:

```
libc.so.1`mutex_trylock_adaptive+0x112
```

module function instruction offset
Background: Stack Trace

- A stack trace is a list of frames. Their index is the *stack depth*:

<table>
<thead>
<tr>
<th>Frame</th>
<th>Stack Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>24</td>
</tr>
<tr>
<td>parent</td>
<td>23</td>
</tr>
<tr>
<td>parent</td>
<td>22</td>
</tr>
<tr>
<td>grand parent</td>
<td>[...]</td>
</tr>
<tr>
<td>grand parent</td>
<td>0</td>
</tr>
</tbody>
</table>

```
• libc.so.1`mutex_trylock_adaptive+0x112
• libc.so.1`mutex_lock_impl+0x165
• libc.so.1`mutex_lock+0xc
• libc.so.1`_lwp_start
```
Background: Stack Trace

- One full stack:

```
libc.so.1`mutex_trylock_adaptive+0x112
libc.so.1`mutex_lock_impl+0x165
libc.so.1`mutex_lock+0xc
mysqld`key_cache_read+0x741
mysqld`_mi_fetch_keypage+0x48
mysqld`w_search+0x84
mysqld`_mi_ck_write_btree+0xa5
mysqld`mi_write+0x344
mysqld`ha_myisam::write_row+0x43
mysqld`handler::ha_write_row+0x8d
mysqld`end_write+0x1a3
mysqld`evaluate_join_record+0x11e
mysqld`sub_select+0x86
mysqld`do_select+0xad9
mysqld`JOIN::exec+0x482
mysqld`mysql_select+0x30e
mysqld`handle_select+0x17d
mysqld`execute_sqlcom_select+0xa6
mysqld`mysql_execute_command+0x124b
mysqld`mysql_parse+0x3e1
mysqld`dispatch_command+0x1619
mysqld`do_handle_one_connection+0x1e5
mysqld`handle_one_connection+0x4c
libc.so.1`_thrp_setup+0xbc
libc.so.1`_lwp_start
```
Background: Stack Trace

- Read top-down or bottom-up, and look for key functions

```
libc.so.1`mutex_trylock_adaptive+0x112
libc.so.1`mutex_lock_impl+0x165
libc.so.1`mutex_lock+0xc
mysql\`key_cache_read+0x741
mysql\`_mi_fetch_keypage+0x48
mysql\`_w_search+0x84
mysql\`_mi_ck_write_btree+0xa5
mysql\`_mi_write+0x344
mysql\`ha_myisam::write_row+0x43
mysql\`handler::ha_write_row+0x8d
mysql\`end_write+0x1a3
mysql\`evaluate_join_record+0x11e
mysql\`sub_select+0x86
mysql\`do_select+0xd9
mysql\`JOIN::exec+0x482
mysql\`mysql_select+0x30e
mysql\`handle_select+0x17d
mysql\`execute_sqlcom_select+0xa6
mysql\`mysql_execute_command+0x124b
mysql\`mysql_parse+0x3e1
mysql\`dispatch_command+0x1619
mysql\`do_handle_one_connection+0x1e5
mysql\`handle_one_connection+0x4c
libc.so.1`_thrp_setup+0xbc
libc.so.1`_lwp_start
```
Two types of stacks can be profiled:

- user-level for applications (user mode)
- kernel-level for the kernel (kernel mode)

During a system call, an application may have both
You don’t need to be a programmer to understand stacks.

Some function names are self explanatory, others require source code browsing (if available). Not as bad as it sounds:

- MySQL has ~15,000 functions in > 0.5 million lines of code
- The earlier stack has 20 MySQL functions. To understand them, you may need to browse only 0.13% (20 / 15000) of the code. Might take hours, but it is doable.
- If you have C++ signatures, you can use a demangler first:

  mysql\_\_ZN4JOIN4execEv+0x482

  gc++filt, demangler.com

  mysql\_JOIN\:_exec() +0x482
Stack frames can be visualized as rectangles (boxes)

Function names can be truncated to fit

In this case, color is chosen randomly (from a warm palette) to differentiate adjacent frames

A stack trace becomes a column of colored rectangles
Background: Time Series Stacks

- Time series ordering allows time-based pattern identification
- However, stacks can change thousands of times per second
Background: Time Series Stacks

- Time series ordering allows time-based pattern identification
- However, stacks can change thousands of times per second
Background: Frame Merging

- When zoomed out, stacks appear as narrow stripes
- Adjacent identical functions can be merged to improve readability, eg:

  When zoomed out, stacks appear as narrow stripes
  Adjacent identical functions can be merged to improve readability, eg:

  This sometimes works: eg, a repetitive single threaded app
  Often does not (previous slide already did this), due to code execution between samples or parallel thread execution
Background: Frame Merging

- Time-series ordering isn’t necessary for the primary use case: identify the most common (“hottest”) code path or paths
- By using a different x-axis sort order, frame merging can be greatly improved...
Flame Graphs

- Flame Graphs sort stacks alphabetically. This sort is applied from the bottom frame upwards. This increases merging and visualizes code paths.
Flame Graphs: Definition

- Each box represents a function (a merged stack frame)
- y-axis shows stack depth
  - top function led directly to the profiling event
  - everything beneath it is ancestry (explains why)
- x-axis spans the sample population, sorted alphabetically
- Box width is proportional to the total time a function was profiled directly or its children were profiled
- All threads can be shown in the same Flame Graph (the default), or as separate per-thread Flame Graphs
- Flame Graphs can be interactive: mouse over for details
Flame Graphs: Variations

- Profile data can be anything: CPU, I/O, memory, ...
  - Naming suggestion: [event] [units] Flame Graph
  - Eg: "FS Latency Flame Graph"
- By default, Flame Graphs == CPU Sample Flame Graphs
- Colors can be used for another dimension
  - by default, random colors are used to differentiate boxes
  - --hash for hash-based on function name
- Distribution applications can be shown in the same Flame Graph (merge samples from multiple systems)
Flame Graphs: A Simple Example

• A CPU Sample Flame Graph:

  ![Flame Graph Example](image)

  - I’ll illustrate how these are read by posing various questions
Flame Graphs: How to Read

- A CPU Sample Flame Graph:

  ![Flame Graph Diagram](diagram)

  a()  b()  c()  d()  e()  f()  g()  h()

- Q: which function is on-CPU the most?
Flame Graphs: How to Read

- A CPU Sample Flame Graph:

- Q: which function is on-CPU the most?
  - A: \( f() \)

\( e() \) is on-CPU a little, but its runtime is mostly spent in \( f() \), which is on-CPU directly.
Flame Graphs: How to Read

• A CPU Sample Flame Graph:

Q: why is $f()$ on-CPU?
Flame Graphs: How to Read

- A CPU Sample Flame Graph:

  ![Flame Graph Diagram](image)

  - \( f() \) was called by \( e() \)
  - \( e() \) was called by \( c() \)
  - ... 

- Q: why is \( f() \) on-CPU?

- A: \( a() \rightarrow b() \rightarrow c() \rightarrow e() \rightarrow f() \)
Flame Graphs: How to Read

- A CPU Sample Flame Graph:

- Q: how does \( b() \) compare to \( g() \)?
Flame Graphs: How to Read

- A CPU Sample Flame Graph:

  ![Flame Graph Diagram]

  - Q: how does \texttt{b()} compare to \texttt{g()}?
  - A: \texttt{b()} looks like it is running (present) about 10 times more often than \texttt{g()}

  \textit{visually compare lengths}
Flame Graphs: How to Read

• A CPU Sample Flame Graph: ... or mouse over

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f()</td>
<td>d()</td>
<td>e()</td>
<td>h()</td>
</tr>
<tr>
<td>c()</td>
<td>b()</td>
<td></td>
<td>g()</td>
</tr>
<tr>
<td>a()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

• Q: how does \( b() \) compare to \( g() \)?

• A: for interactive Flame Graphs, mouse over shows \( b() \) is 90%, \( g() \) is 10%
Flame Graphs: How to Read

- A CPU Sample Flame Graph: ... or mouse over

- Q: how does $b()$ compare to $g()$?

- A: for interactive Flame Graphs, mouse over shows $b()$ is 90%, $g()$ is 10%
Flame Graphs: How to Read

- A CPU Sample Flame Graph:

```
<table>
<thead>
<tr>
<th>a()</th>
<th>b()</th>
<th>c()</th>
<th>d()</th>
<th>e()</th>
<th>f()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Q: why are we running $f()$?
Flame Graphs: How to Read

- A CPU Sample Flame Graph:

  ![Flame Graph Diagram](image)

  - Q: why are we running \( f() \)?
  - A: code path branches can reveal key functions:
    - \( a() \) choose the \( b() \) path
    - \( c() \) choose the \( e() \) path

  look for branches
Flame Graphs: Example 1

- Customer alerting software periodically checks a log, however, it is taking too long (minutes).
- It includes `grep(1)` of an ~18 Mbyte log file, which takes around 10 minutes!
- `grep(1)` appears to be on-CPU for this time. Why?
Flame Graphs: Example 1

• CPU Sample Flame Graph for grep(1) user-level stacks:
Flame Graphs: Example 1

- CPU Sample Flame Graph for grep(1) user-level stacks:

- 82% of samples are in check_multibyte_string() or its children. This seems odd as the log file is plain ASCII.

- And why is UTF8 on the scene? ... Oh, LANG=en_US.UTF-8
Flame Graphs: Example 1

- CPU Sample Flame Graph for grep(1) user-level stacks:

- Switching to LANG=C improved performance by 2000x

- A simple example, but I did spot this from the raw profiler text before the Flame Graph. You really need Flame Graphs when the text gets too long and unwieldy.
Flame Graphs: Example 2

• A potential customer benchmarks disk I/O on a cloud instance. The performance is not as fast as hoped.
• The host has new hardware and software. Issues with the new type of disks is suspected.
Flame Graphs: Example 2

• A potential customer benchmarks disk I/O on a cloud instance. The performance is not as fast as hoped.
• The host has new hardware and software. Issues with the new type of disks is suspected.
• I take a look, and notice CPU time in the kernel is modest.
• I’d normally assume this was I/O overheads and not profile it yet, instead beginning with I/O latency analysis.
• But Flame Graphs make it easy, and it may be useful to see what code paths (illumos kernel) are on the table.
Flame Graphs: Example 2
Flame Graphs: Example 2

- 24% in tsc_read()? Time Stamp Counter? Checking ancestry...
• 62% in zfs_zone_io_throttle? Oh, we had forgotten that this new platform had ZFS I/O throttles turned on by default!
Flame Graphs: Example 3

- Application performance is about half that of a competitor
- Everything is believed identical (H/W, application, config, workload) except for the OS and kernel
- Application is CPU busy, nearly 100% in user-mode. How can the kernel cause a 2x delta when the app isn't in kernel-mode?
- Flame graphs on both platforms for user-mode were created:
  - Linux, using perf
  - SmartOS, using DTrace
- Added flamegraph.pl --hash option for consistent function colors (not random), aiding comparisons
Flame Graphs: Example 3

- Function label formats are different, but that's just due to different profilers/stackcollapse.pl's (should fix this)
- Widths slightly different, but we already know perf differs
- Extra function? This is executing different application software!

Extra Function:

```cpp
SphDocID_t UnzipDocid () { return UnzipOffset(); }
```
Flame Graphs: More Examples

- Flame Graphs are typically more detailed, like the earlier MySQL example
- Next, how to generate them, then more examples
Generation
Generation

• I’ll describe the original Perl version I wrote and shared on github:
  • https://github.com/brendangregg/FlameGraph
• There are other great Flame Graph implementations with different features and usage, which I’ll cover in the last section
Generation: Steps

- 1. Profile event of interest
- 2. stackcollapse.pl
- 3. flamegraph.pl
Generation: Overview

• Full command line example. This uses DTrace for CPU profiling of the kernel:

```
# dtrace -x stackframes=100 -n 'profile-997 /arg0/ {
    @[stack()] = count(); } tick-60s { exit(0); }' -o out.stacks

# stackcollapse.pl < out.stacks > out.folded

# flamegraph.pl < out.folded > out.svg
```

• Then, open out.svg in a browser

• Intermediate files could be avoided (piping), but they can be handy for some manual processing if needed (eg, using vi)
The profile data, at a minimum, is a series of stack traces. These can also include stack trace counts. Eg:

```
mysqld`_Z13add_to_statusP17system_status_varS0_+0x47
mysqld`_Z22calc_sum_of_all_statusP17system_status_var+0x67
mysqld`_Z16dispatch_command19enum_server_commandP3THDPcj+0x1222
mysqld`_Z10do_commandP3THD+0x198
mysqld`handle_one_connection+0x1a6
libc.so.1`_thrp_setup+0x8d
libc.so.1`_lwp_start
```

# of occurrences for this stack

This example is from DTrace, which prints a series of these. The format of each group is: stack, count, newline.

Your profiler needs to print full (not truncated) stacks, with symbols. This may be step 0: get the profiler to work!
Generation: Profiling Tools

- Solaris/FreeBSD/SmartOS/…:
  - DTrace
- Linux:
  - perf, SystemTap
- OS X:
  - Instruments
- Windows:
  - Xperf.exe
Generation: Profiling Examples: DTrace

• CPU profile kernel stacks at 997 Hertz, for 60 secs:

```bash
# dtrace -x stackframes=100 -n 'profile-997 /arg0/ {
    @[stack()] = count(); } tick-60s { exit(0); }' -o out.kern_stacks
```

• CPU profile user-level stacks for PID 12345 at 99 Hertz, 60s:

```bash
# dtrace -x ustackframes=100 -n 'profile-97 /PID == 12345 && arg1/ {
    @[ustack()] = count(); } tick-60s { exit(0); }' -o out.user_stacks
```

• Should also work on Mac OS X, but is pending some fixes preventing stack walking (use Instruments instead)

• Should work for Linux one day with the DTrace ports
**Generation: Profiling Examples: perf**

- CPU profile full stacks at 97 Hertz, for 60 secs:
  ```bash
  # perf record -a -g -F 97 sleep 60
  # perf script > out.stacks
  ```

- Need debug symbol packages installed (*dbgsym), otherwise stack frames may show as hexadecimal

- May need compilers to cooperate (-fno-omit-frame-pointer)

- Has both user and kernel stacks, and the kernel idle thread. Can filter the idle thread after `stackcollapse-perf.pl` using:
  ```bash
  # stackcollapse-perf.pl < out.stacks | grep -v cpu_idle | ...
  ```
Generation: Profiling Examples: SystemTap

- CPU profile kernel stacks at 100 Hertz, for 60 secs:

```
# stap -s 32 -D MAXTRACE=100 -D MAXSTRINGLEN=4096 -D MAXMAPENTRIES=10240 \
-D MAXACTION=10000 -D STP_OVERLOAD_THRESHOLD=5000000000 --all-modules \
-ve 'global s; probe timer.profile { s[backtrace()] <<< 1; } 
probe end { foreach (i in s+) { print_stack(i); printf("\t%d\n", @count(s[i])); } } probe timer.s(60) { exit(); }' \\n> out.kern_stacks
```

- Need debug symbol packages installed (*dbgsym), otherwise stack frames may show as hexadecimal

- May need compilers to cooperate (-fno-omit-frame-pointer)
Generation: Dynamic Languages

- C or C++ are usually easy to profile, runtime environments (JVM, node.js, ...) are usually not, typically a way to show program stacks and not just runtime internals.

- Eg, DTrace’s ustack helper for node.js:

```
0xfc618bc0
0xfc61bd62
0xfe870841
0xfc61c1f3
0xfc617685
0xfe870841
0xfc6154d7
0xfe870e1a
[...]
```

```
libc.so.1`gettimeofday+0x7
Date at position
<< adaptor >>
<< constructor >>
(anon) as exports.active at timers.js position 7590
(anonymous) as Socket._write at net.js position 21336
(anonymous) as Socket.write at net.js position 19714
<< adaptor >>
(anonymous) as OutgoingMessage._writeRaw at http.js p...
(anonymous) as OutgoingMessage._send at http.js posit...
<< adaptor >>
  (anonymous) as OutgoingMessage.end at http.js pos...
[...]
```

http://dtrace.org/blogs/dap/2012/01/05/where-does-your-node-program-spend-its-time/
Generation: stackcollapse.pl

- Converts profile data into a single line records
- Variants exist for DTrace, perf, SystemTap, Instruments, Xperf
- Eg, DTrace:

```plaintext
unix`i86_mwait+0xd
unix`cpu_idle_mwait+0xf1
unix`idle+0x114
unix`thread_start+0x8
19486
```

```bash
# stackcollapse.pl < out.stacks > out.folded
```

```
unix`thread_start;unix`idle;unix`cpu_idle_mwait;unix`i86_mwait 19486
```
Generation: stackcollapse.pl

- Converts profile data into a single line records
- Variants exist for DTrace, perf, SystemTap, Instruments, Xperf
- Eg, DTrace:
  
  ```
  unix`i86_mwait+0xd
  unix`cpu_idle_mwait+0xf1
  unix`idle+0x114
  unix`thread_start+0x8
  19486
  ```

  # stackcollapse.pl < out.stacks > out.folded

  ```
  unix`thread_start;unix`idle;unix`cpu_idle_mwait;unix`i86_mwait 19486
  ```

  stack trace, frames are ‘;’ delimited
  count
Generation: stackcollapse.pl

- Full output is many lines, one line per stack
- Bonus: can be grepped

```bash
# ./stackcollapse-stap.pl out.stacks | grep ext4fs_dirhash
system_call_fastpath;sys_getdents;vfs_readdir;ext4_readdir;ext4_htree_fill_tree;htree_dirblock_to_tree;ext4fs_dirhash 100
system_call_fastpath;sys_getdents;vfs_readdir;ext4_readdir;ext4_htree_fill_tree;htree_dirblock_to_tree;ext4fs_dirhash;half_md4_transform 505
system_call_fastpath;sys_getdents;vfs_readdir;ext4_readdir;ext4_htree_fill_tree;htree_dirblock_to_tree;ext4fs_dirhash;str2hashbuf_signed 353
[...]
```

- That shows all stacks containing ext4fs_dirhash(); useful debug aid by itself
- grep can also be used to filter stacks before Flame Graphs
  - eg: grep -v cpu_idle
Generation: Final Output

• Desires:
  • Full control of output
  • High density detail
  • Portable: easily viewable
  • Interactive
Generation: Final Output

• Desires:
  • Full control of output
  • High density detail
  • Portable: easily viewable
  • Interactive

• SVG+JS: Scalable Vector Graphics with embedded JavaScript
  • Common standards, and supported by web browsers
  • Can print poster size (scalable); but loses interactivity!
  • Can be emitted by a simple Perl program...
Generation: flamegraph.pl

- Converts folded stacks into an interactive SVG. Eg:

```bash
# flamegraph.pl --titletext="Flame Graph: MySQL" out.folded > graph.svg
```

- Options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--titletext</td>
<td>change the title text (default is “Flame Graph”)</td>
</tr>
<tr>
<td>--width</td>
<td>width of image (default is 1200)</td>
</tr>
<tr>
<td>--height</td>
<td>height of each frame (default is 16)</td>
</tr>
<tr>
<td>--minwidth</td>
<td>omit functions smaller than this width (default is 0.1 pixels)</td>
</tr>
<tr>
<td>--fonttype</td>
<td>font type (default “Verdana”)</td>
</tr>
<tr>
<td>--fontsize</td>
<td>font size (default 12)</td>
</tr>
<tr>
<td>--countname</td>
<td>count type label (default “samples”)</td>
</tr>
<tr>
<td>--nametype</td>
<td>name type label (default “Function:”)</td>
</tr>
<tr>
<td>--colors</td>
<td>color palette: &quot;hot&quot;, &quot;mem&quot;, &quot;io&quot;</td>
</tr>
<tr>
<td>--hash</td>
<td>colors are keyed by function name hash</td>
</tr>
</tbody>
</table>
Types
Types

- CPU
- Memory
- Off-CPU
- More
CPU
CPU

• Measure code paths that consume CPU
• Helps us understand and optimize CPU usage, improving performance and scalability
• Commonly performed by sampling CPU stack traces at a timed interval (e.g., 100 Hertz for every 10 ms), on all CPUs
  • DTrace/perf/SystemTap examples shown earlier
• Can also be performed by tracing function execution
CPU: Sampling

CPU stack sampling:

A A A A B - - - - B A A A A

A( syscall X Off-CPU

A( On-CPU

user-level
kernel

block . . . . . . . . . . interrupt

syscall
CPU: Tracing

CPU function tracing:

A(..............)

B(..............)

A(..............)

B(..............)

--- syscall ---

On-CPU

X

Off-CPU

interrupt

user-level

kernel
CPU: Profiling

• Sampling:
  • Coarse but usually effective
  • Can also be low overhead, depending on the stack type and sample rate, which is fixed (e.g., 100 Hz x CPU count)

• Tracing:
  • Overheads can be too high, distorting results and hurting the target (e.g., millions of trace events per second)

• Most Flame Graphs are generated using stack sampling
CPU: Profiling Results

• Example results. Could you do this?

As an experiment to investigate the performance of the resulting TCP/IP implementation ... the [REDACTED] is CPU saturated, but the [REDACTED] has about 30% idle time. The time spent in the system processing the data is spread out among handling for the Ethernet (20%), IP packet processing (10%), TCP processing (30%), checksumming (25%), and user system call handling (15%), with no single part of the handling dominating the time in the system.
CPU: Profiling Results

- Example results. Could you do this?

As an experiment to investigate the performance of the resulting TCP/IP implementation ... the 11/750 is CPU saturated, but the 11/780 has about 30% idle time. The time spent in the system processing the data is spread out among handling for the Ethernet (20%), IP packet processing (10%), TCP processing (30%), checksumming (25%), and user system call handling (15%), with no single part of the handling dominating the time in the system.

  – Bill Joy, 1981, TCP-IP Digest, Vol 1 #6

- An impressive report, that even today would be difficult to do

- Flame Graphs make this a lot easier
CPU: Another Example

- A file system is archived using tar(1).
- The files and directories are cached, and the run time is mostly on-CPU in the kernel (Linux). Where exactly?
CPU: Another Example
CPU: Another Example

- 20% for reading directories
CPU: Another Example

- 54% for file statistics
CPU: Another Example

Also good for learning kernel internals: browse the active code
Once you start profiling a target, you begin to recognize the common stacks and patterns.

Linux getdents() ext4 path:

The next slides show similar example kernel-mode CPU Sample Flame Graphs.
CPU: Recognition: illumos localhost TCP

• From a TCP localhost latency issue (illumos kernel):
CPU: Recognition: illumos IP DCE issue
CPU: Recognition: Linux TCP send

- Profiled from a KVM guest:
CPU: Recognition: Syscall Towers
CPU: Recognition: Syscall Towers
CPU: Both Stacks

- Apart from showing either user- or kernel-level stacks, both can be included by stacking kernel on top of user
  - Linux perf does this by default
  - DTrace can by aggregating @[stack(), ustack()]
- The different stacks can be highlighted in different ways:
  - different colors or hues
  - separator: flamegraph.pl will color gray any functions called ",-", which can be inserted as stack separators
- Kernel stacks are only present during syscalls or interrupts
CPU: Both Stacks Example: KVM/qemu
Advanced Flame Graphs
Other Targets

- Apart from CPU samples, stack traces can be collected for any event; eg:
  - disk, network, or FS I/O
  - CPU events, including cache misses
  - lock contention and holds
  - memory allocation
- Other values, instead of sample counts, can also be used:
  - latency
  - bytes
- The next sections demonstrate memory allocation, I/O tracing, and then all blocking types via off-CPU tracing
Memory
Memory

• Analyze memory growth or leaks by tracing one of the following memory events:
  
  1. Allocator functions: malloc(), free()
  2. brk() syscall
  3. mmap() syscall
  4. Page faults

• Instead of stacks and sample counts, measure stacks with byte counts

• Merging shows show total bytes by code path
Memory: Four Targets

1. `malloc()`
2. `brk()`
3. `mmap()`
4. `munmap()`

- Allocator (libc)
- Heap
- Mappings
- Virtual Memory
- MMU
- Physical Memory
- Process Address Space
Memory: Allocator

• Trace malloc(), free(), realloc(), calloc(), ...

• These operate on virtual memory

• *alloc() stacks show why memory was first allocated (as opposed to populated): Memory Allocation Flame Graphs

• With free()/realloc()/..., suspected memory leaks during tracing can be identified: Memory Leak Flame Graphs!

• Down side: allocator functions are frequent, so tracing can slow the target somewhat (eg, 25%)

• For comparison: Valgrind memcheck is more thorough, but its CPU simulation can slow the target 20 - 30x
Memory: Allocator: malloc()

- As a simple example, just tracing malloc() calls with user-level stacks and bytes requested, using DTrace:

```bash
# dtrace -x ustackframes=100 -n 'pid$target::malloc:entry {
  @[ustack()] = sum(arg0); } tick-60s { exit(0); }' -p 529 -o out.malloc
```

- malloc() Bytes Flame Graph:

```bash
# stackcollapse.pl out.malloc | flamegraph.pl --title="malloc() bytes" \
  --countname="bytes" --colors=mem > out.malloc.svg
```

- The options customize the title, countname, and color palette
Memory: Allocator: malloc()
Yichun Zhang developed Memory Leak Flame Graphs using SystemTap to trace allocator functions, and applied them to leaks in Nginx (web server):
Memory: brk()

• Many apps grow their virtual memory size using brk(), which sets the heap pointer
• A stack trace on brk() shows what triggered growth
• Eg, this script (brkbytes.d) traces brk() growth for “mysqld”:

```bash
#!/usr/sbin/dtrace -s

inline string target = "mysqld";
uint brk[int];

syscall::brk:entry /execname == target/ { self->p = arg0; }
system::brk:return /arg0 == 0 && self->p && brk[pid]/ {
    @[ustack()] = sum(self->p - brk[pid]);
}
system::brk:return /arg0 == 0 && self->p/ { brk[pid] = self->p; }
system::brk:return /self->p/ { self->p = 0; }
```
Memory: brk(): Heap Expansion

# ./brkbytes.d -n 'tick-60s { exit(0); }' > out.brk

# stackcollapse.pl out.brk | flamegraph.pl --countname="bytes" \
   --title="Heap Expansion Flame Graph" --colors=mem > out.brk.svg
Memory: brk()

- brk() tracing has low overhead: these calls are typically infrequent

- Reasons for brk():
  - A memory growth code path
  - A memory leak code path
  - An innocent application code path, that happened to spill-over the current heap size
  - Asynchronous allocator code path, that grew the application in response to diminishing free space
Memory: mmap() 

- mmap() may be used by the application or its user-level allocator to map in large regions of virtual memory.
- It may be followed by munmap() to free the area, which can also be traced.
- Eg, mmap() tracing, similar to brk tracing, to show bytes and the stacks responsible:

```
# xxd -p /var/log/mmap.out
```

- This should be low overhead – depends on the frequency.
Memory: Page Faults

- `brk()` and `mmap()` expand virtual memory
- Page faults expand physical memory (RSS). This is demand-based allocation, deferring mapping to the actual write
- Tracing page faults show the stack responsible for consuming (writing to) memory:

```
# dtrace -x ustackframes=100 -n 'vminfo:::as_fault /execname == "mysqld"/ { @[ustack()] = count(); } tick-60s { exit(0); }' > out.fault

# stackcollapse.pl out.mysqld_fault01 | flamegraph.pl --countname=pages --title="Page Fault Flame Graph" --colors=mem > mysqld_fault.svg
```
Memory: Page Faults

Page Fault Flame Graph

Function: all (30,826 pages, 100%)
I/O

- Show time spent in I/O, e.g., storage I/O
- Measure I/O completion events with stacks and their latency; merging to show total time waiting by code path

Logical I/O:
Measure here for user stacks, and real application latency

Physical I/O:
Measure here for kernel stacks, and disk I/O latency
I/O: Logical I/O Latency

- For example, ZFS call latency using DTrace (zfsustack.d):

```c
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option ustackframes=100

fbt::zfs_read:entry, fbt::zfs_write:entry,
fbt::zfs_readdir:entry, fbt::zfs_getattr:entry,
fbt::zfs_setattr:entry
{
    self->start = timestamp;
}

fbt::zfs_read:return, fbt::zfs_write:return,
fbt::zfs_readdir:return, fbt::zfs_getattr:return,
fbt::zfs_setattr:return
/self->start/
{
    this->time = timestamp - self->start;
    @[ustack(), execname] = sum(this->time);
    self->start = 0;
}

dtrace:::END
{
    printa("%k%s\n%@d\n", @);
}
```

Timestamp from function start (entry) ... to function end (return)
I/O: Logical I/O Laency

• Making an I/O Time Flame Graph:

```
# ./zfsustacks.d -n 'tick-10s { exit(0); }' -o out.iostacks
# stackcollapse.pl out.iostacks | awk '{ print $1, $2 / 1000000 }' | \ flamegraph.pl --title="FS I/O Time Flame Graph" --color=io \ --countname=ms --width=500 > out.iostacks.svg
```

• DTrace script measures all processes, for 10 seconds

• awk to covert ns to ms
gzip(1) waits more time in write()s than read()s

Function: gzip `flush_block` (226 ms, 66.10%)
I/O: Time Flame Graph: MySQL

Function: mysqld shared library function: 255 ms, 26.25%
I/O: Flame Graphs

- I/O latency tracing: hugely useful
- But once you pick an I/O type, there usually isn't that many different code paths calling it
- Flame Graphs are nice, but often not necessary
Off-CPU
Off-CPU tracing:

Off-CPU — on-CPU

On-CPU syscall

user-level kernel

block . . . . . . . . . interrupt
Off-CPU: Performance Analysis

- Generic approach for all blocking events, including I/O
- An advanced performance analysis methodology:
- Counterpart to (on-)CPU profiling
- Measure time a thread spent off-CPU, along with stacks
- Off-CPU reasons:
  - Waiting (sleeping) on I/O, locks, timers
  - Runnable waiting for CPU
  - Runnable waiting for page/swap-ins
- The stack trace will explain which
Off-CPU: Time Flame Graphs

• Off-CPU profiling data (durations and stacks) can be rendered as **Off-CPU Time Flame Graphs**

• As this involves many more code paths, Flame Graphs are usually really useful

• Yichun Zhang created these, and has been using them on Linux with SystemTap to collect the profile data. See:
  

• Which describes their uses for Nginx performance analysis
Off-CPU: Profiling

- Example of off-CPU profiling for the bash shell:

```bash
# dtrace -x ustackframes=100 -n ' 
    sched:::off-cpu /execname == "bash"/ { self->ts = timestamp; } 
    sched:::on-cpu /self->ts/ { 
        @[ustack()] = sum(timestamp - self->ts); self->ts = 0; } 
    tick-30s { exit(0); }' -o out.offcpu
```

- Traces time from when a thread switches off-CPU to when it returns on-CPU, with user-level stacks. i.e., time blocked or sleeping

- Off-CPU Time Flame Graph:

```bash
# stackcollapse.pl < out.offcpu | awk '{ print $1, $2 / 1000000 }' | \ 
    flamegraph.pl --title="Off-CPU Time Flame Graph" --color=io \ 
    --countname=ms --width=600 > out.offcpu.svg
```

- This uses awk to convert nanoseconds into milliseconds
# Off-CPU: Bash Shell

## Off-CPU Time Flame Graph

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc.so.1`__read</td>
</tr>
<tr>
<td>bash`rl_getc</td>
</tr>
<tr>
<td>bash`rl_read_key</td>
</tr>
<tr>
<td>bash`readline_internal_char</td>
</tr>
<tr>
<td>bash`readline</td>
</tr>
<tr>
<td>libc... bash`yy_readline_get</td>
</tr>
<tr>
<td>libc... bash`shell_getc</td>
</tr>
<tr>
<td>bash<code>.. bash</code>read_token</td>
</tr>
<tr>
<td>bash<code>.. bash</code>yyparse</td>
</tr>
<tr>
<td>bash<code>.. bash</code>parse_command</td>
</tr>
<tr>
<td>bash<code>.. bash</code>read_command</td>
</tr>
<tr>
<td>bash`reader_loop</td>
</tr>
<tr>
<td>bash`main</td>
</tr>
<tr>
<td>bash`_start</td>
</tr>
</tbody>
</table>

Function: libc.so.1`waitpid (1,193 ms, 8.65%)
Off-CPU: Bash Shell

Waiting for:
- Keystrokes
- Child processes

Off-CPU Time Flame Graph:

Function: libc.so.1\`waitpid (1,193 ms, 8.65%)
Off-CPU: Bash Shell

• For that simple example, the trace data was so short it could have just been read (54 lines, 4 unique stacks):

• For multithreaded applications, idle thread time can dominate

• For example, an idle MySQL server...
Off-CPU: MySQL Idle
Columns from _thrp_setup are threads or thread groups

MySQL gives thread routines descriptive names (thanks!)
Mouse over each to identify

Function: mysql`buf_flush_page_cleaner_thread (29,001 ms, 5.52%) (profiling time was 30s)
Off-CPU: MySQL Idle

Off-CPU Time Flame Graph

buf_flush_page_cleaner_thread
dict_stats_thread
fts_optimize_thread
io_handler_thread
lock_wait_timeout_thread

mysqld Main
srv_monitor_thread
srv_master_thread
srv_error_monitor_thread
pfs_spawn_thread
Off-CPU: MySQL Idle

- Some thread columns are wider than the measurement time: evidence of multiple threads
- This can be shown a number of ways. Eg, adding process name, PID, and TID to the top of each user stack:

```bash
#!/usr/sbin/dtrace -s

#pragma D option ustackframes=100

sched:::off-cpu /execname == "mysqld"/ { self->ts = timestamp; }

sched:::on-cpu /self->ts/
{  
    @[execname, pid, curlwpsinfo->pr_lwpid, ustack()] = 
        sum(timestamp - self->ts);
    self->ts = 0;
}

dtrace:::END { printa("\n%s-%d/%d%k%@d\n", @); }
```
Off-CPU: MySQL Idle

1 thread
2 threads
many threads
4 threads doing work (less idle)
Off-CPU: Challenges

- Including multiple threads in one Flame Graph might still be confusing. Separate Flame Graphs for each can be created.
- Off-CPU stacks often don't explain themselves:
  
  ```
  libc.so.1`__lwp_park
  libc.so.1`cond_wait_queue
  libc.so.1`__cond_wait
  libc.so.1`cond_wait
  libc.so.1`pthread_cond_wait
  mysql`one_thread_per_connection_end
  ```

- This is blocked on a conditional variable. The real reason it is blocked and taking time isn't visible here.

- Now let's look at a busy MySQL server, which presents another challenge...
Off-CPU: MySQL Busy

idle threads

net_read_packet() -> pollsys()

Function: libc.so.1`__pollsys (289,499 ms, 48.09%)
random narrow stacks during work, with no reason to sleep?
Off-CPU: MySQL Busy

- Those were user-level stacks only. The kernel-level stack, which can be included, will usually explain what happened
  - eg, involuntary context switch due to time slice expired
- Those paths are likely hot in the CPU Sample Flame Graph
Hot/Cold
Hot/Cold: Profiling

Thread State Transition Diagram

- Runnable
  - wakeup
  - acquire
  - work arrives
  - Sleep
    - I/O wait
    - block
  - Lock
    - acquire
    - work arrives
  - Idle
    - wait for work

- Executing
  - schedule
  - preempted or time quantum expired

- Anon. Paging
  - anon. major fault
  - page in

On-CPU Profiling
Off-CPU Profiling (everything else)
Hot/Cold: Profiling

- Profiling both on-CPU and off-CPU stacks shows everything.
- In my LISA'12 talk I called this the *Stack Profile Method*: profile all stacks.
- Both on-CPU ("hot") and off-CPU ("cold") stacks can be included in the same Flame Graph, colored differently: **Hot Cold Flame Graphs**!
- Merging multiple threads gets even weirder. Creating a separate graph per-thread makes much more sense, as comparisons to see how a thread's time is divided between on- and off-CPU activity.
- For example, a single web server thread with kernel stacks...
Hot/Cold: Flame Graphs

Function: `sockfs.so_accept` (43413 ms, 88.26%)
Hot/Cold: Flame Graphs

On-CPU (!?)

Off-CPU

Function: sockets`so_accept (43413 ms, 88.26%)
Hot/Cold: Challenges

• Sadly, this often doesn't work well for two reasons:

  1. On-CPU time columns get compressed by off-CPU time
     • Previous example dominated by the idle path – waiting for a new connection – which is not very interesting!
     • Works better with zoomable Flame Graphs, but then we've lost the ability to see key details on first glance
     • Pairs of on-CPU and off-CPU Flame Graphs may be the best approach, giving both the full width

  2. Has the same challenge from off-CPU Flame Graphs: real reason for blocking may not be visible
State of the Art

• That was the end of Flame Graphs, but I can't stop here – we're so close
  • On + Off-CPU Flame Graphs can attack any issue
• 1. The compressed problem is solvable via one or more of:
  • zoomable Flame Graphs
  • separate on- and off-CPU Flame Graphs
  • per-thread Flame Graphs
• 2. How do we show the real reason for blocking?
Wakeup Tracing

Wakeup tracing:

sleep \rightarrow \text{wakeup}

\text{On-CPU}

X \rightarrow \text{Off-CPU}

\text{block} \rightarrow \text{wakeup}

user-level

kernel

B(}
Tracing Wakeups

- The systems knows who woke up who
- Tracing who performed the wakeup – and their stack – can show the real reason for waiting
- **Wakeup Latency Flame Graph**
- Advanced activity
- Consider overheads – might trace too much
- Eg, consider ssh, starting with the Off CPU Time Flame Graph
Waiting on a conditional variable
But *why* did we wait this long?

Object sleeping on
Wakeup Latency Flame Graph: ssh
Wakeup Latency Flame Graph: ssh

These code paths, ...

... woke up these objects
This example targets sshd (previous example also matched vmstat, after discovering that sshd was blocked on vmstat, which it was: "vmstat 1")

Time from sleep to wakeup

Stack traces of who is doing the waking

Aggregate if possible instead of dumping, to minimize overheads
Following Stack Chains

- 1st level of wakeups often not enough
- Would like to programmatically follow multiple chains of wakeup stacks, and visualize them
- I've discussed this with others before – it's a hard problem
- The following is in development!: Chain Graph
Chain Graph
Chain Graph

...  
Wakeup Stack
why I waited  
Wakeup Stacks  
I wokeup
Wakeup Thread 2  
I wokeup  
Wakeup Thread 1
Off CPU Stacks:  
why I blocked
Chain Graph Visualization

- New, experimental; check for later improvements
- Stacks associated based on sleeping object address
- Retains the value of relative widths equals latency
- Wakeup stacks frames can be listed in reverse (may be less confusing when following towers bottom-up)
- Towers can get very tall, tracing wakeups through different software threads, back to metal
Following Wakeup Chains, Example (DTrace)

```c
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option ustackframes=100
#pragma D option stackframes=100

int related[uint64_t];

sched:::sleep
/execname == "sshd" || related[curlwpsinfo->pr_addr]/
{
    ts[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/ts[args[0]->pr_addr]/
{
    this->d = timestamp - ts[args[0]->pr_addr];
    @[args[1]->pr_fname, args[1]->pr_pid, args[0]->pr_lwpid, args[0]->pr_wchan,
       stack(), usystem(), execname, pid, curlwpsinfo->pr_lwpid] = sum(this->d);
    ts[args[0]->pr_addr] = 0;
    related[curlwpsinfo->pr_addr] = 1;
}

dtrace:::END
{
    printa("\n%s-%d/%d-%x%k-%k%s-%d/%d\n", @);
}
```

Also follow who wakes up the waker
Developments
Developments

• There have been many other great developments in the world of Flame Graphs. The following is a short tour.
node.js Flame Graphs

- Dave Pacheco developed the DTrace ustack helper for v8, and created Flame Graphs with node.js functions

http://dtrace.org/blogs/dap/2012/01/05/where-does-your-node-program-spend-its-time/
OS X Instruments Flame Graphs

- Mark Probst developed a way to produce Flame Graphs from Instruments

  1. Use the Time Profile instrument
  2. Instrument -> Export Track
  3. stackcollapse-instruments.pl
  4. flamegraphs.pl

http://schani.wordpress.com/2012/11/16/flame-graphs-for-instruments/
Ruby Flame Graphs

• Sam Saffron developed Flame Graphs with the Ruby MiniProfiler

• These stacks are very deep (many frames), so the function names have been dropped and only the rectangles are drawn

• This preserves the value of seeing the big picture at first glance!

http://samsaffron.com/archive/2013/03/19/flame-graphs-in-ruby-miniprofiler
Windows Xperf Flame Graphs

- Bruce Dawson developed Flame Graphs from Xperf data, and an xperf_to_collapsedstacks.py script

http://randomascii.wordpress.com/2013/03/26/summarizing-xperf-cpu-usage-with-flame-graphs/
WebKit Web Inspector Flame Charts

- Available in Google Chrome developer tools, these show JavaScript CPU stacks as colored rectangles
- Inspired by Flame Graphs but not the same: they show the passage of time on the x-axis!
- This generally works here as:
  - the target is single threaded apps often with repetitive code paths
  - ability to zoom
- Can a "Flame Graph" mode be provided for the same data?

https://bugs.webkit.org/show_bug.cgi?id=111162
Perl Devel::NYTProf Flame Graphs

• Tim Bunce has been adding Flame Graph features, and included them in the Perl profiler: Devel::NYTProf

http://blog.timbunce.org/2013/04/08/nytprof-v5-flaming-precision/
Leak and Off-CPU Time Flame Graphs

- Yichun Zhang (agentzh) has created Memory Leak and Off-CPU Time Flame Graphs, and has given good talks to explain how Flame Graphs work.

http://agentzh.org/#Presentations
http://www.youtube.com/watch?v=rxn7HoNrv9A
http://agentzh.org/misc/flamegraph/nginx-leaks-2013-10-08.svg
https://github.com/agentzh/nginx-systemtap-toolkit

... these also provide examples of using SystemTap on Linux
Color Schemes

- Colors can be used to convey data, instead of the default random color scheme. This example from Dave Pacheco colors each function by its degree of direct on-CPU execution.

- A Flame Graph tool could let you select different color schemes.

- Another can be: color by a hash on the function name, so colors are consistent.

https://npmjs.org/package/stackvis
Zoomable Flame Graphs

- Dave Pacheco has also used d3 to provide click to zoom!

https://npmjs.org/package/stackvis
Robert Mustacchi has been experimenting with showing the difference between two Flame Graphs, as a Flame Graph. Great potential for non-regression testing, and comparisons!
Flame Graphs as a Service

- Pedro Teixeira has a project for node.js Flame Graphs as a service: automatically generated for each github push!

http://www.youtube.com/watch?v=sMohaWP5YqA
References & Acknowledgements

- Neelakanth Nadgir (realneel): developed SVGs using Ruby and JavaScript of time-series function trace data with stack levels, inspired by Roch's work

- Roch Bourbonnais: developed Call Stack Analyzer, which produced similar time-series visualizations

- Edward Tufte: inspired me to explore visualizations that show all the data at once, as Flame Graphs do

- Thanks to all who have developed Flame Graphs further!

realneel's function_call_graph.rb visualization
Thank you!

• Questions?
• Homepage: http://www.brendangregg.com (links to everything)
• Resources and further reading:
  • http://dtrace.org/blogs/brendan/2011/12/16/flame-graphs/: see "Updates"
  • http://dtrace.org/blogs/brendan/2012/03/17/linux-kernel-performance-flame-graphs/
  • http://dtrace.org/blogs/brendan/2013/08/16/memory-leak-growth-flame-graphs/
  • http://dtrace.org/blogs/dap/2012/01/05/where-does-your-node-program-spend-its-time/