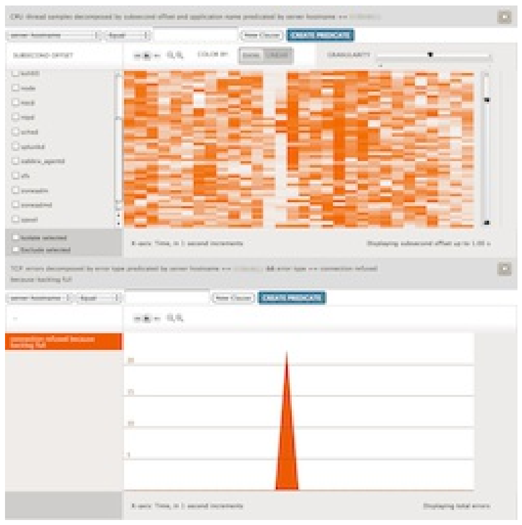
Subsecond Offset Heat Maps

“Wow, that’s weird!”. My subsecond offset visualization type looked great, but others found it weird and unfamiliar. I developed it for inclusion in Joyent’s Cloud Analytics tool for the purposes of *workload characterization*. Given that it was so unfamiliar, I had some explaining to do.

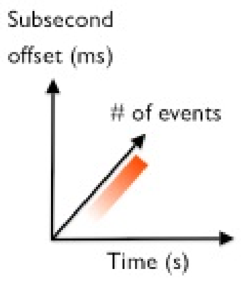


Voxer, a company that makes a walkie-talkie application for smart phones, had been seeing a performance issue with their Riak database. The issue appeared to be related to TCP listen drops – when SYNs are dropped as the application can’t keep up with the accept() queue. Voxer has millions of users whose numbers are growing fast, so I expected to see Riak hit 100% CPU usage when these drops occurred. The subsecond offset heat map (top on the right) painted a different story, which led to an operating system kernel fix.

Weird but wonderful, this heat map helped solve a hard problem, and I was left with some interesting screenshots to help explain this visualization type.

In this post, I’ll explain subsecond offset heat maps using the Voxer issue as a case study, then show various other interesting examples from a production cloud environment. This environment is a single datacenter that includes 200 physical servers and thousands of OS instances. The heat maps are all generated by Joyent Cloud Analytics, which uses DTrace to fetch the data.

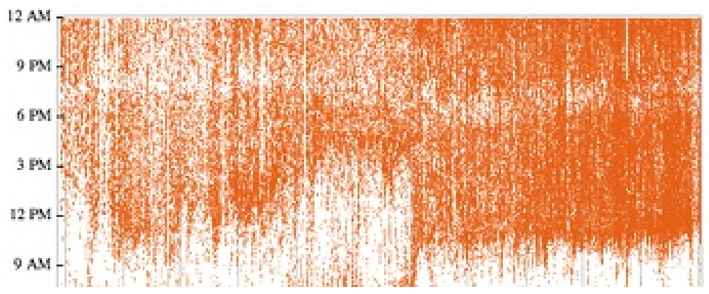
Description



The subsecond offset heat map puts time on two axes. The x-axis shows the passage of time, with each column representing one second. The y-axis shows the *time within a second*, spanning from 0.0s to 1.0s (time offsets). The z-axis (color) show the count of samples or events, quantized into x- and y-axis ranges (“buckets”), with the color darkness reflecting the event count (darker == more). This relationship is shown to the right.

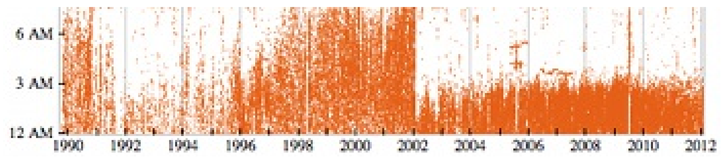
I previously explained the use of quantized heat maps in section 11 of Visualizing Device Utilization. I use them to show event latency as well.

Time on Two Axes



Heat maps aren’t the weird part. What’s weird is putting time on more than one axis. Stephen Wolfram recently posted The Personal Analytics of My Life, which included an amazing scatter plot (on the left). This has time on both x- and y- axes. I’ve included it as it may be a much easier example to grasp at first glance, before the subsecond offset heat maps.

His is at a much longer time scale: the x-axis shows days, and the y-axis shows offset within a day. Using similar terminology, this could be called a “subday-offset” or “24hr-offset” scatter plot. Each point on his plot shows when Wolfram sent an email, revealing his sleeping habits as the white gap in the morning.



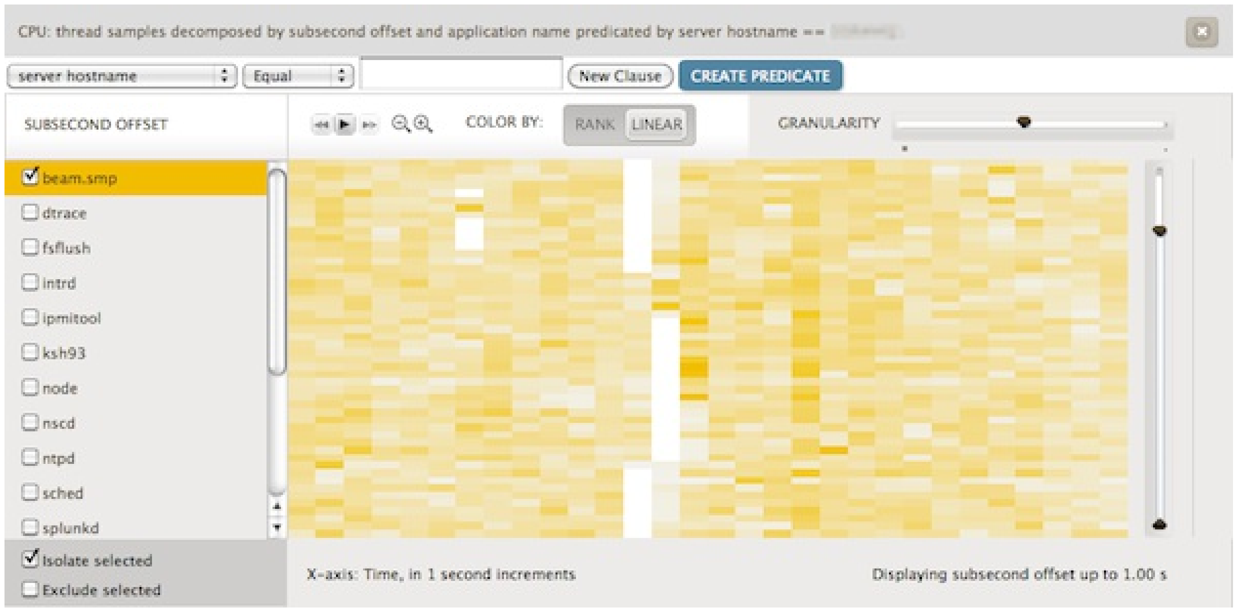
Scatter plots are limited in the density of the points they can display, and don’t compress the data set (x & y coordinates are kept for each event). Heat maps solve both issues, allowing them to scale, which is especially important for the cloud computing uses that follow. These use the subsecond offset scale, but other ranges are possible as well (minute-offset, hour-offset, day-offset).

That’s No Artifact

The screenshot at the top of this page (click any for full-res) used a subsecond offset heat map for CPU thread samples – showing when applications were on-CPU during the second. The sampling was at 99 Hertz across all CPUs, to minimize overhead (instead of, say, 1000 Hz), and to avoid lockstep (with any power-of-10 Hz task). These CPU samples are then quantized into the buckets seen as pixels.

The heat map revealed that CPU usage dropped at the same time as the TCP listen drops. I was expecting the opposite.

By selecting Riak (as “beam.smp”, the Erlang VM it uses) and “Isolate selected”, only Riak is shown:


CPU: thread samples decomposed by subsecond offset and application name predicated by server hostname ==

server hostname | Equal | New Clause | CREATE PREDICATE

SUBSECOND OFFSET

COLOR BY: RANK LINEAR

GRANULARITY

☑ beam.smp
☐ dtrace
☐ fsflush
☐ intrd
☐ ipmitool
☐ ksh93
☐ node
☐ nscd
☐ ntpd
☐ sched
☐ splunkd

☑ Isolate selected
☐ Exclude selected

X-axis: Time, in 1 second increments

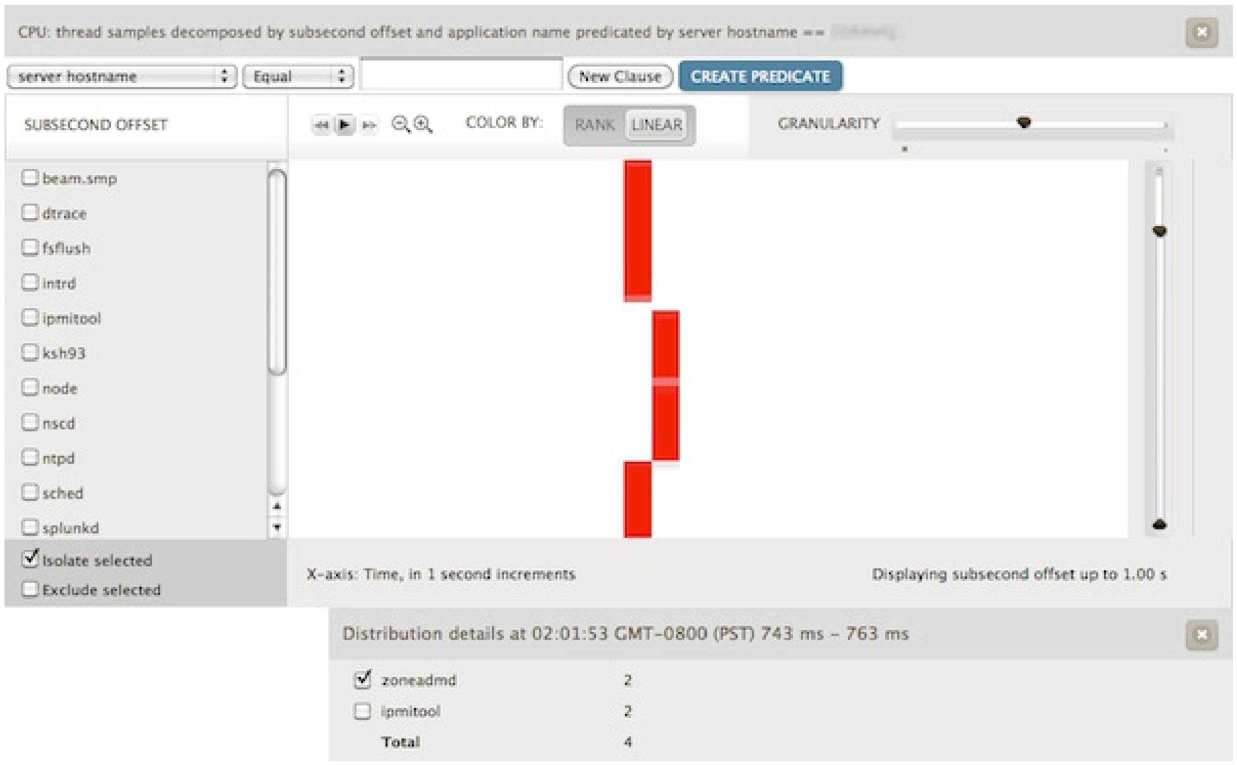
Displaying subsecond offset up to 1.00 s

Left of center shows two columns, each with about 40% of the offsets colored white. Assuming no sampling issue, it means that the Riak database was entirely off-CPU for hundreds of consecutive milliseconds. This is similar to the white gaps showing when Wolfram was asleep — except that we aren’t expecting the Riak database to take naps! This was so bizarre that I first thought that something was wrong with the instrumentation, and that the white gaps were an artifact.

Application threads normally spend time off-CPU when blocked on I/O or waiting for work. What’s odd here is that for so long the number of running Riak threads is zero, when normally it varies more quickly. And this event coincided with TCP listen drops.

The Shoe That Fits

In Cloud Analytics, heat maps can be clicked to reveal details at that point. I clicked inside the white gap, which revealed that a process called “zoneadmd” was running; isolating it:



Distribution details at 02:01:53 GMT-0800 (PST) 743 ms - 763 ms

✓ zoneadmd	2
ipmitool	2
Total	4

This fits the white gap closely, and a similar relationship was observed at other times as well. This pointed suspicion to zoneadmd, which other observability tools had missed. Some tools sampled the running processes every few seconds or minutes, and usually missed the short-lived zoneadmd completely. Even watching every second was difficult: Riak’s CPU usage dropped for two seconds, at a different rate to what zoneadmd consumed (Riak is multi-threaded, so it can consume more CPU in the same interval than the single-threaded zoneadmd). The subsecond offset heat map showed the clearest correlation: the duration of these events was similar, and the starting and ending points were nearby.

If zoneadmd was somehow blocking Riak from executing, it would explain the off-CPU gap and also the TCP listen drops – as Riak wouldn’t be running to accept the connections.

Kernel Fix

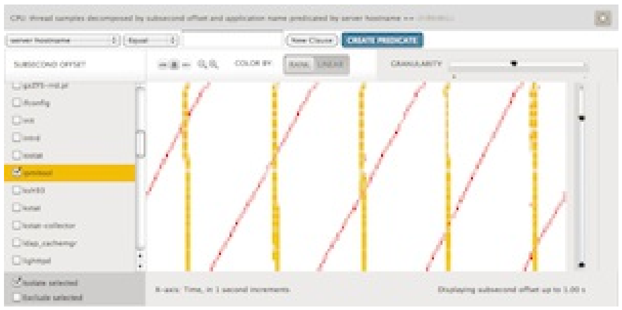
Investigation on the server using DTrace quickly found that Riak was getting blocked as it waited for an address space lock (as_lock) during mmap()/munmap() calls from its bitcask storage engine. That lock was being held by zoneadmd for hundreds of milliseconds (see the Artifacts section later for a longer description). zoneadmd enforces multi-tenant memory limits, and every couple of minutes checked the size of Riak. It did this via kernel calls which scan memory pages while holding as_lock. This scan took time, as Riak was tens of Gbytes in size.

We found other applications exhibiting the same behavior, including Riak’s “memsup” memory monitor. All of these were blocking Riak, and with Riak off-CPU unable to accept() connections, the TCP backlog queue often hit its limit resulting in TCP listen drops (tcpListenDrop). Jerry Jelinek of Joyent has been fixing these codepaths via kernel changes.

Patterns

The previous heat map included a “Distribution details” box at the bottom, summarizing the quantized bucket that I clicked on. It shows that “zoneadmd” and “ipmitool” were running, each sampled twice in the range 743 – 763 ms (consistent with them being single threaded and sampled at 99 Hertz).

ipmitool and zabbix_agentd



To check whether ipmitool was an issue, I isolated its on-CPU usage and found that it often did not coincide with Riak off-CPU time. While checking this, I found a interesting pattern caused by zabbix_agentd. These are shown on the right: ipmitool is highlighted in yellow, and zabbix_agentd in red.

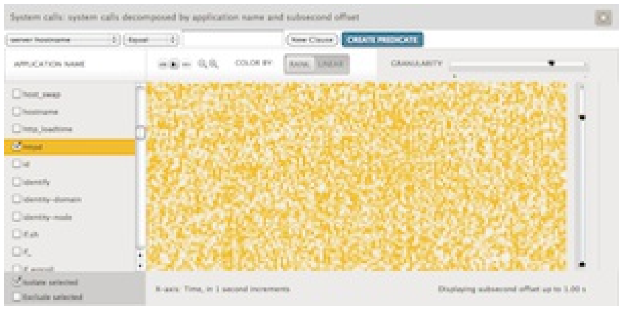
Just based on the heat map, it would appear that zabbix_agentd is a single thread (or process) that wakes up every second to perform a small amount of work. It then sleeps for an entire second. Repeat. This causes the diagonal rising line, the slope of which is relative to time zabbix_agentd worked before sleeping for the next full second: with greater slopes (approaching 90 degrees) reflecting more work was performed before the next sleep.

zabbix_agentd is part of the Zabbix monitoring software. If it is supposed to perform work roughly every second, then it should be ok. But if it is supposed to perform work *exactly* once a second, such as reading system counters to calculate the statistics it is monitoring, then there could be problems.

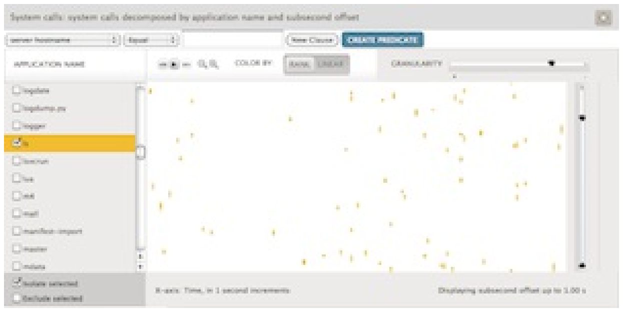
Cloud Scale

The previous examples showed CPU thread samples on a single server (each title included “predicated by server hostname == …”). Cloud Analytics can show these for the entire cloud – which may be hundreds of systems (virtualized operating system instances). I’ll show this with a different heat map type: instead of CPU thread samples, which shows the CPU usage of applications, I’ll show subsecond offset of system calls (syscalls), which paints a different picture – one better reflecting the I/O behavior. Tracing syscalls can reveal more processes than by sampling, which can miss short-lived events.

The two images that follow show subsecond offsets for syscalls across an entire datacenter (200 physical servers, thousands of OS instances). On the left are syscalls by “httpd” (Apache web server), and the right are those by the “ls” command:



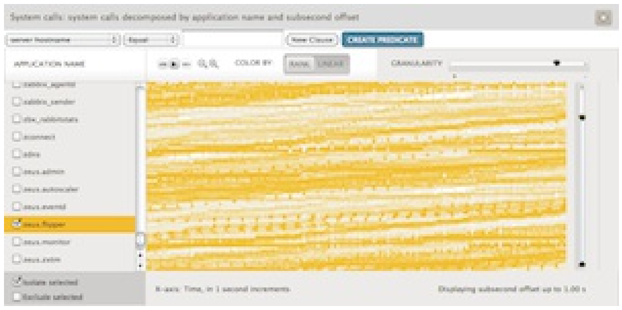
httpd

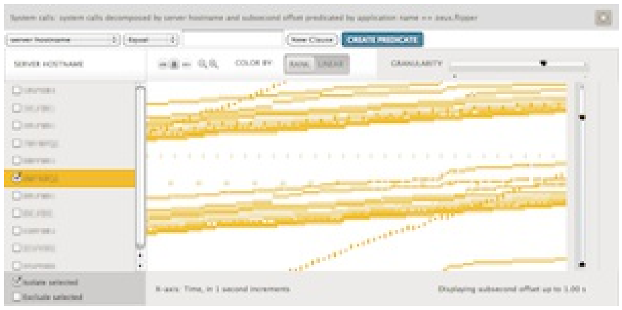


ls

Neither of these may be very surprising. The httpd syscalls will arrive at random times based on the client workload, and combining them for dozens of busy web servers results in a heat map with random color intensities (which have been enhanced due to the rank-based default color map).

Sometimes the heat maps are surprising. The next two show zeus.flipper (web load balancing software), on the left for the entire cloud, and on the right for a single server:

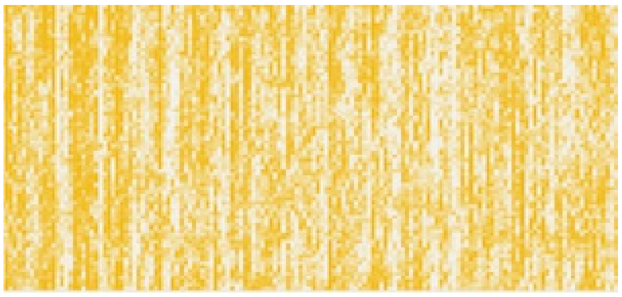

zeus.flipper

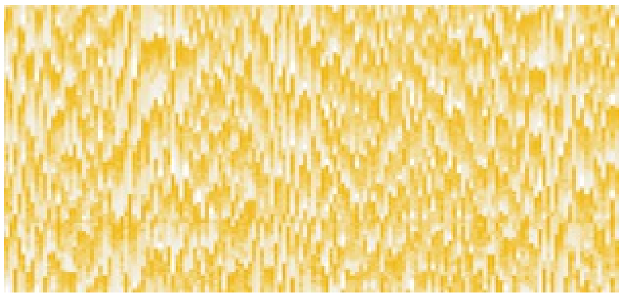

zeus.flipper (single)

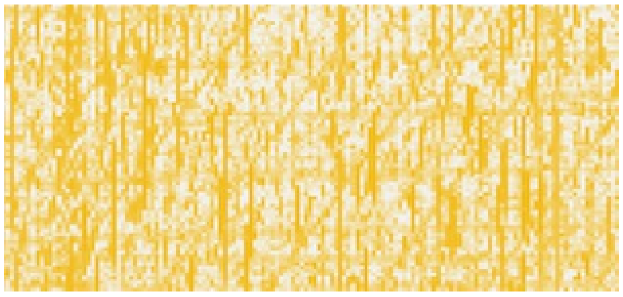
The cloud-wide heat map does show that there is a pattern present, which has been isolated for a single server on the right. It appears that multiple threads are present: many waking up more than once a second (the two large bands), and others waking up every two (), five () and ten seconds ().

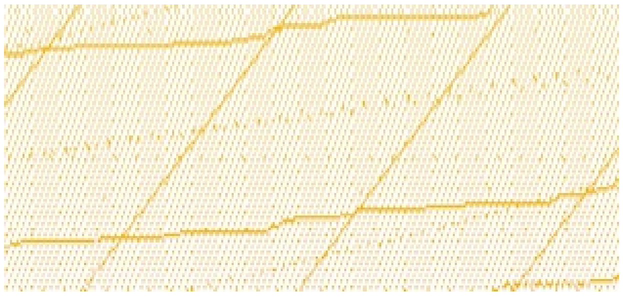
Cloud Wide vs Single Server

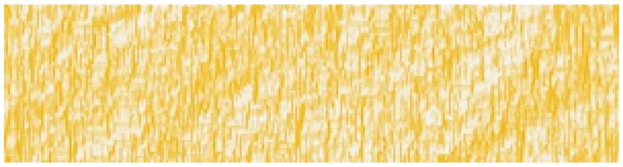
Here are some other examples comparing an entire cloud vs a single server (click for full screenshot). These are also syscall subsecond offsets:

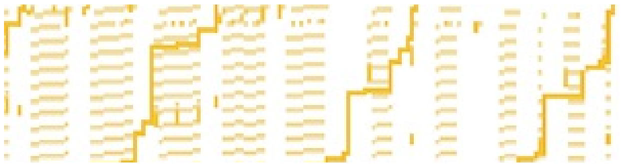

node.js

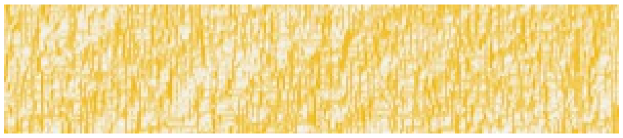

node.js (single)

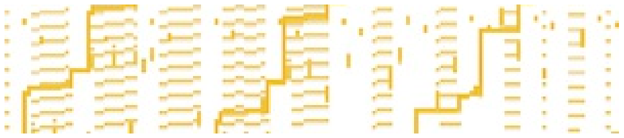

Java


Java (single)





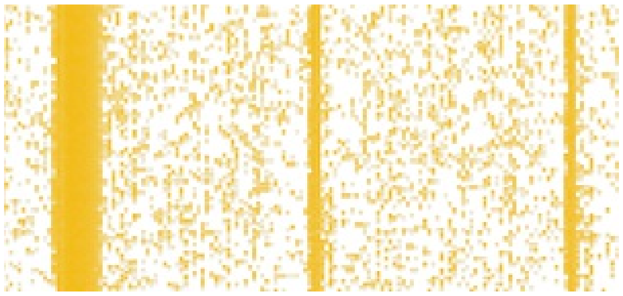

Python

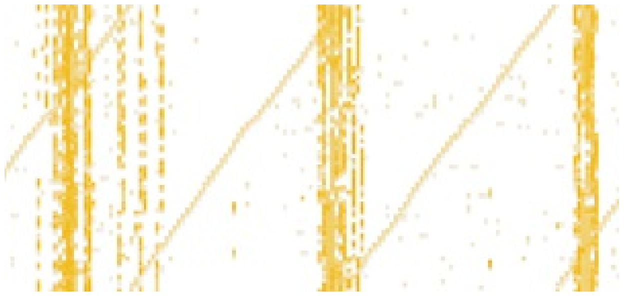

Python (single)

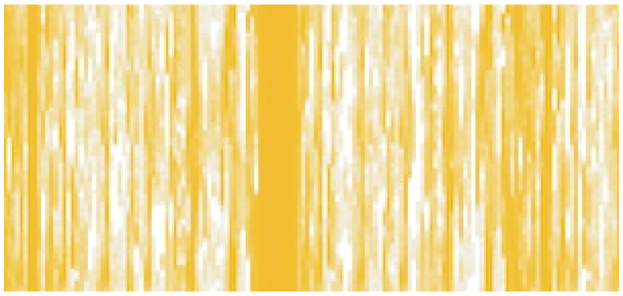
I've just shown one single server example for node.js, Java, and Python, however each server can look quite different based on its use and workload. Applications such as zeus.flipper are more likely to look similar as they serve the same function on every server.

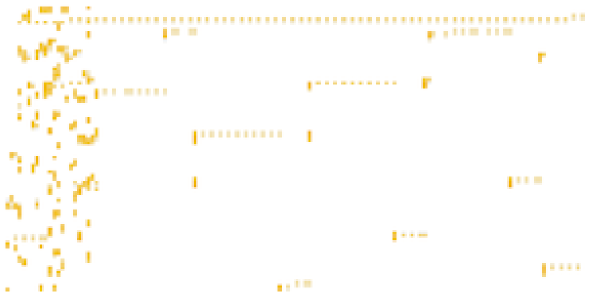
Cloud Identification Chart

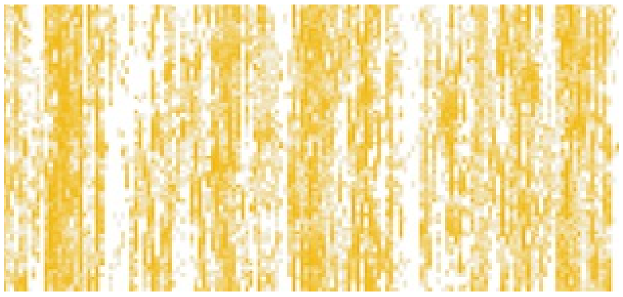
Some other cloud-wide examples, using syscall subsecond offsets:

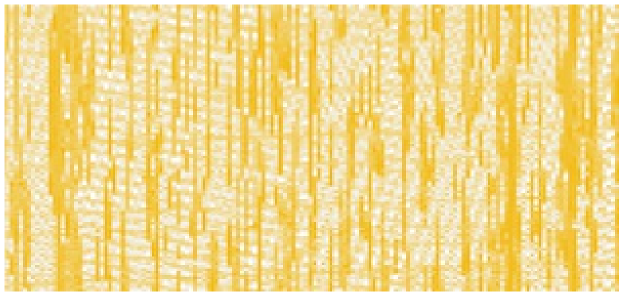

awk


bash


kstat


munin-node


Perl

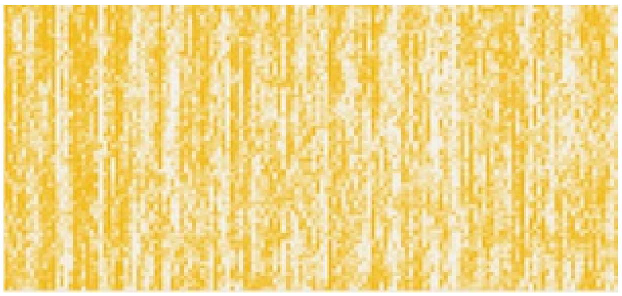

php-fpm

The munin-node heat map has several lines of dots , each dot two seconds apart. Can you guess what those might be?

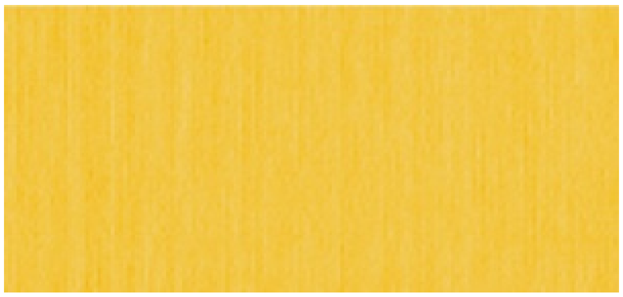
Color Maps

The colors chosen for the heat map can either be rank-based or linear-based, which select color saturation differently. The selected type for the previous heat maps can be seen after "COLOR BY:" in the full screenshots (click images)

This shows node.js processes across the entire cloud, to compare the color maps side-by-side:



node.js (rank)

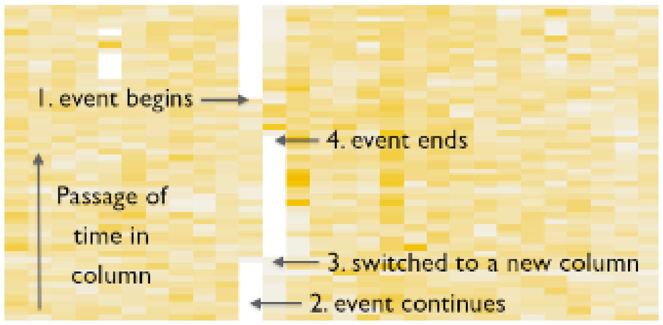


node.js (linear)

The rank-based heat map highlights subtle variation well. The linear colored heat map reflects reality. This is an extreme example; often the heat maps look much more similar. For a longer description of rank vs linear, see Dave Pacheco's heat map coloring post, and the Saturation section in my Visualizing Device Utilization post.

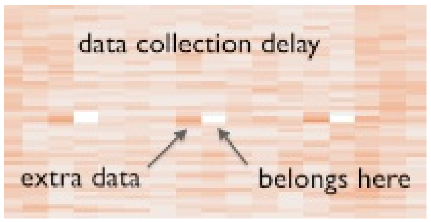
Artifacts

The first example I showed featured the Riak database being blocked by zoneadmd. The blocking event was continuous, and lasted for almost a full second. It was shown twice in the first subsecond offset column due to the way the data is currently collected by Joyent Cloud Analytics – resulting in an "artifact".



This is shown on the right. The time that a column of data is collected from the server does not occur at the 0.0 offset, but rather some other offset during the second. This means that an activity that is in-flight will suddenly jump to the next column, as has happened here (at the "3″ mark). It also means that an activity at the top of the column can wrap and continue at the bottom of the same column (at the "2″ mark), before the column switch occurs. I think this is fixable by recalculating offsets relative to the data collection time, so the switch happens at offset 0.0. (It hasn't been fixed already since it usually isn't annoying, and didn't noticeably interfere with the many other examples I've shown.)

On the left is a different type of artifact, one caused when data collection is delayed. To minimize overhead, data is aggregated in-kernel, then read at a gentle rate (usually once per second) by a user-land process. This problem occurs when the user-land process is delayed slightly for some reason, and the kernel aggregations include extra data (overlapping offsets) by the time they are read. Those offsets are then missing from the next column, on the right.



Thread Scheduling

I intended to include a “checkerboard” heat map of CPU samples, like those Robert Mustacchi showed in his Visualizing KVM post. This involves running two threads (or processes) that share one CPU, each performing the same CPU-bound workload. When each is highlighted in different colors it should look like a checkerboard, as the kernel scheduler evenly switches between running them.

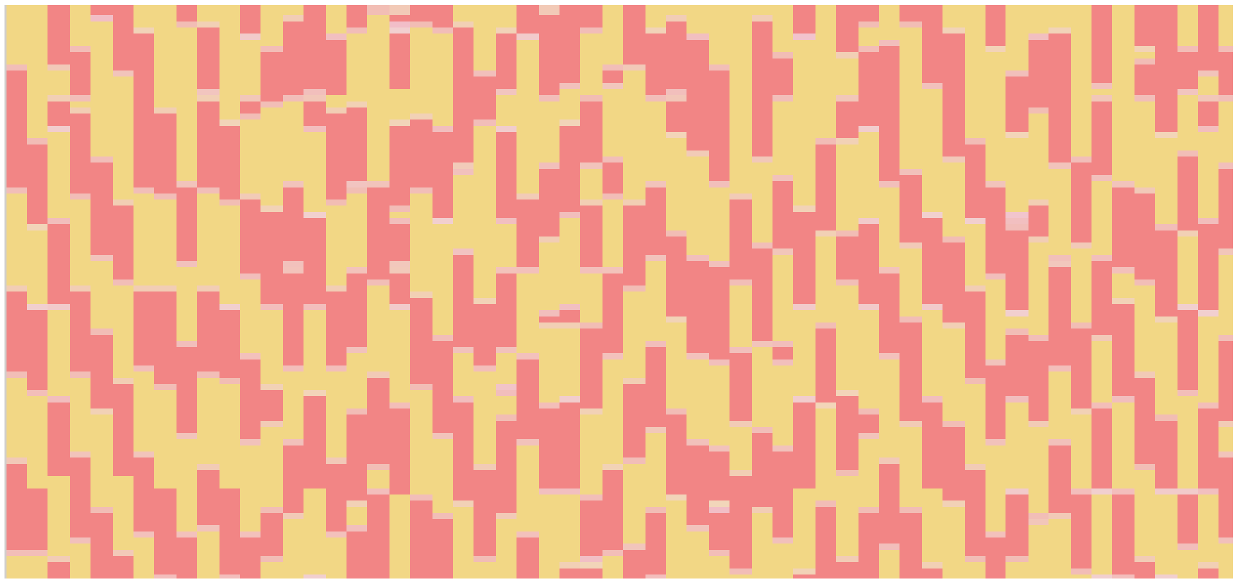
Robert was testing on the Linux kernel under KVM, and used DTrace to inspect running threads from the SmartOS host (by checking the VM MMU context). I performed the experiment on SmartOS directly, which resulted in the following heat map:

This breaks my head. Instead of a neat checkerboard, this is messy – showing uneven runtimes for the identical threads. One column in particular is entirely red, which if true (not a sampling or instrumentation error) meant that the scheduler left the same thread running for an entire second, while another was in the ready-to-run state on the CPU dispatcher queue. This is much longer than the maximum runtime quantum set by the scheduler (110 ms for the FSS class). I confirmed this behavior using two different means (DTrace, and thread microstate accounting), and saw even worse instances – threads blocked for many seconds when they should have been running.

Jerry Jelinek has been wading into the scheduler code, finding evidence that this is a kernel bug (in code that hasn’t changed since Solaris) and developing the fix. Fortunately, not many of our customers have hit this since it requires CPUs running at saturation (which isn’t normal for us).

UPDATE (April 2nd)

Jerry has fixed the code, which was a bug with how thread priorities were updated in the scheduler. The following screenshot shows the same workload post-fix:



This looks much better. There are no longer any full seconds where one thread hogs the CPU, with the other thread waiting. Looking more closely, there appear to be cases where the thread has switched early – which is much better than switching late.

We also found that the bug was indeed hurting a customer along with a confluence of other factors.

Conclusion

The subsecond offset heat map provides new visibility for software execution time, which can be used for workload characterization and performance analysis. These are currently available in Joyent Cloud Analytics, from which I included screenshots of these heat maps for production environments.

Using these heat maps I identified two kernel scheduling issues, one of which was causing dropped TCP connections for a large scale cloud-based service. Kernel fixes are being developed for both. I also showed various applications running on single servers and the cloud, which produced fascinating patterns – providing a new way of understanding application runtime.

The examples I included here were based on sampled thread runtime, and traced system call execution times. Other event sources can be visualized in this way, and these could also be produced on other time frames: sub-minute, sub-hour, etc.

Acknowledgements

- Dave Pacheco leads the Joyent Cloud Analytics project. All of the heat maps shown here (except Wolfram's) are from Cloud Analytics.
- The Cloud Analytics team with whom I discussed this visualization.
- Bryan Cantrill wrote the prototype Cloud Analytics tool which I hacked to test out this idea in production, and he came up with the name "subsecond offset". He also fathered DTrace, which is used to fetch all the data shown by the heat maps.
- Robert Mustacchi developed the Meta-D language for Cloud Analytics instrumentations, which made implementing subsecond offset heat maps trivial, and put them to use for KVM.
- Jerry Jelinek, for fixing the tricky kernel bugs that these heat maps have recently been unearthing.
- Stephen Wolfram's recent post was great timing, as it provided an intuitive example of a graph having time on both axes.
- Edward Tufte for the idea of high definition images in text, and for inspiration to try harder in general (see any of his texts).
- Deirdré Straughan for edits and suggestions.

Thanks to the folk at Voxer for realizing (earlier than I did) that something more than just normal bursts of load was causing the tcpListenDrops, and pushing for the real answer.

Posted on March 26, 2012 at 11:16 am by Brendan Gregg · Permalink In: Performance · Tagged with: cloud analytics, dtrace, subsecond, visualizations

« Previous post
Next post »