kstats

Luigi Rizzo, lrizzo@google.com

Code: https://lwn.net/Articles/813303/

Summary

- Goals, motivation, design strategy
- kstats in a nutshell
- Internals, performance, accuracy...
- Related work
- Conclusions

Goals and motivation

```
-- RUNTIME MEASUREMENT --
t0 = clock();
work();
runtime = clock() - t0;

-- LATENCY MEASUREMENT --
ON SENDER:
x.t0 = clock()

ON RECEIVER:
latency = clock() - x.t0;
```

Goal measure runtime or latency for kernel code

Targets O(30ns) accuracy (Δ from real value)

O(30ns) resolution (δ between distinct values)

O(1M) samples per second per CPU

Motivation

study caching effect, lock contention, synchronization latencies

Applications

design, performance evaluation, optimizations, configuration, troubleshooting

Design strategy

Tool designer: **make the tool easy to use and non intrusive**

- complexity discourages adoption
- intrusivity alters system behaviour making measurements meaningless

Users: **be aware of the limitations of your tools**

- key to understanding output data (accuracy, confidence intervals...)
- time-related measurements are especially fragile
- be wary of additional abstraction layers (they may add noise)

kstats in a nutshell

```
    Change-Id: I4befd3df4400d4cca2a3343a8d69431d56668bbd

include/linux/kstats.h |  34 ++++++++++++++++
kernel/Makefile        |   1 +
kernel/kstats.c        | 303 ++++++++++++++++++++++++++++++++++...
```

```
#include <linux/kstats.h>
struct kstats *key = kstats_new("foo", 3 /* frac_bits */);

u64 t0 = ktime_get_ns();                  /* about 20ns on x86 */
do_something()
kstats_record(key, ktime_get_ns() - t0);  /* 30ns hot cache, 300ns cold */
```

```
$ cat /sys/kernel/debug/kstats/foo  # export values
...
slot 55  CPU  1    count       18 avg     480 p 0.002572
slot 55  CPU  2    count       25 avg     480 p 0.003325
...
slot 55  CPU  224  count        1 avg     480 p 0.001310
slot 55  CPUS 256  count      814 avg     480 p 0.002474
...
slot 97  CPU  254  count     1150 avg   20130 p 0.447442
slot 97  CPU  255  count       26 avg   20294 p 0.275555
slot 97  CPUS 256  count   152585 avg   19809 p 0.651747
...
```

kstats internals

```
start = clock();
work();
runtime = clock() - start;
```

kstats addresses several orthogonal problems

- **P1: instrument the code**
  - manual
  - dynamic probes
- **P2: acquire samples**
  - read a clock, understand its cost and accuracy/resolution
- **P3: aggregate values**
  - tradeoff between accuracy/resolution, time, space
- **P4: export data**
  - choose a useful format, robust to future extensions
- **P5: presentation: NO!**
  - plot, histogram... there are external tools for that!

These are addressed in the rest of the presentation.

General caveats

```
start = clock();
work();
runtime = clock() - start;
```

Obvious, caveats, but always good to remember:

- clock sources need proper serialization and possibly synchronization
- interrupts, preemption, contention can alter samples
  - don't call them errors or outliers: those are what affects our tails
  - this is why we collect distributions not just averages
- clock choice affects accuracy, pick the right one for your task
- collection cost affects max sampling rate (unfortunately, highly variable)

P1: Instrumentation

```
start = clock();
work();
runtime = clock() - start;
```

- manual
  - easy and fast at runtime, but intrusive
- dynamic probes (~breakpoints)
  - supported by linux (kprobe, kretprobe, tracepoints)
  - attach hooks around the code to be measured
  - naming the attach point may be non trivial
  - compiler optimizations and inlining gets in the way
  - significantly more expensive (trampolines, out of line code), affecting accuracy
- BPF
  - another layer on top of dynamic code modification
  - adds convenience, dependencies, runtime cost, measurement errors

P2: Acquire samples

```
start = clock();
work();
runtime = clock() - start;
```

- trivial if the sample is an already available value
  - block size, iteration count ...
- otherwise must read a clock twice, compute difference
  - need serialized and possibly synchronized clocks
  - accuracy is platform dependent
  - ktime_get_ns(), local_clock(), rdtscp(), rdtsc() have increasingly relaxed features
  - the clock read function may take cache or TLB misses (unlikely to be visible, because the first call will prime caches)
  - also possible that the clock read may spin a bit to synchronize with hw or timebase
- expect ~20ns accuracy for ktime_get_ns(), 10-15ns for local_clock()
  - the above for back-to-back reads. Tails are 40ns with interrupt disabled
  - platform dependent
- I don't have an estimate for the cost of serialization

P3: Aggregate values

- our target sample rate (1M/s per cpu) is too high to export a trace
  - per-CPU aggregation becomes mandatory
- split the range into (logarithmic) buckets. For each sample *x* do

```
index = log(x)/log(bucket_range)
bucket[cpu][index].count++
bucket[cpu][index].total += x
```

- most tools use bucket_range=2 which reduces to index = fls64(x)
  - this reduces resolution to 1 bit i.e. all values between N and 2N are merged
  - not enough for our purposes!
  - smaller bucket_range (e.g. 1.1) increases significant bits
- kstats makes resolution configurable (up to 5 bits) and approximates logarithm with shift and mask

P3: Aggregate values: actual code

- values have (1 + frac_bits) significant bits
- the sum is scaled to guarantee ~20 significant bits also for large values
- cost is ~30ns with hot cache, 300ns with cold cache

```
void kstats_record(struct kstats *ks, u64 val)
{
    /* Leftmost 1 selects the bucket, subsequent frac_bits select the slot within the
     * bucket. fls returns 0 when the argument is 0. frac_mask = (1 << frac_bits) - 1
     */
    u64 bucket = fls64(val >> ks->frac_bits);
    u64 slot = bucket == 0 ? val : ((bucket << ks->frac_bits) |
                                    ((val >> (bucket - 1)) & ks->frac_mask));
    /* preempt_disable protects from migration, this_cpu_add() uses a non
     * interruptible add, safe against hw interrupts which may call kstats_record.
     */
    preempt_disable();
    this_cpu_add(ks->slots[slot].samples, 1);
    this_cpu_add(ks->slots[slot].sum,
                 bucket < SUM_SCALE ? val : (val >> (bucket - SUM_SCALE)));
    preempt_enable();
}
```

P4: Export data, control operation

- export and presentation are two different steps
  - but several tools merge them, producing histograms or other output
- it may make sense to export raw data
  - that still freezes the API and requires metadata (bits, #CPUs, ...)
  - it also requires some userspace tool to produce useful output (cdf, pdf)

After trying a few options, I went for text format and minimal kernel preprocessing

export: `cat /sys/kernel/debug/kstats/foo`

control: `echo {reset|start|stop} > /sys/kernel/debug/kstats/foo`

P4: export data format

- one line per-slot, per cpu
- each line is self contained to ease postprocessing
  - format is friendly to grep, awk, gnuplot, watch, ...
  - easy to filter by CPU or aggregate, plot cdf or pdf

```
$ cat /sys/kernel/debug/kstats/foo
...
slot 55  CPU  0    count      589 avg     480 p 0.027613
slot 55  CPU  1    count       18 avg     480 p 0.002572
slot 55  CPU  2    count       25 avg     480 p 0.003325
...
slot 55  CPU  224  count        1 avg     480 p 0.001310
slot 55  CPUS 256  count      814 avg     480 p 0.002474
...
slot 97  CPU  254  count     1150 avg   20130 p 0.447442
slot 97  CPU  255  count       26 avg   20294 p 0.275555
slot 97  CPUS 256  count   152585 avg   19809 p 0.651747
...
```

Performance numbers

```
volatile int a, b;
start = clock();
work();
runtime = clock() - start;
```

Times in ns, each cell has two values:

cold cache (10 calls/s) / hot cache (>100k calls/ns)

Clock source / operation	p10	p50	p90	p99	p99.9
rdtsc() # don't use this!	22 / 22	45 / 22	67 / 45	67 / 45	
local_clock() # !sync	29 / 10	30 / 10	40 / 20	40 / 20	
ktime_get_ns()	30 / 20	40 / 29	50 / 30	60 / 30	
a = b + 1; (1 thread)	20 / 20	40 / 30	80 / 30	100 / 30	
a = b + 1; (2 threads)	40 / 250	200 / 250	213 / 250	230 / 260	265 / 600
kstats_record()	149 / 30	250 / 30	260 / 40	490 / 40	650 / 60

kstats accuracy

```
volatile int a, b;
start = clock();
work();
runtime = clock() - start;
```

From the first line we see accuracy is 30ns hot, 60ns cold.

Hence the hot measurements in line 2 are only an upper bound.

For a better estimate of kstats_record() we must measure a longer interval

Clock source / operation	p10	p50	p90	p99	p99.9
ktime_get_ns()	30 / 20	40 / 29	50 / 30	60 / 30	
kstats_record()	149 / 30	250 / 30	260 / 40	490 / 40	650 / 60
kstats_record() 100 times no interrupts	543	543	543	675	993

P5: Presentation (external tools)

- data format is well suited for post processing

```
...
slot 97  CPU  254  count     1150 avg    20130 p 0.447442
slot 97  CPU  255  count       26 avg    20294 p 0.275555
slot 97  CPUS 256  count   152585 avg    19809 p 0.651747
...
```

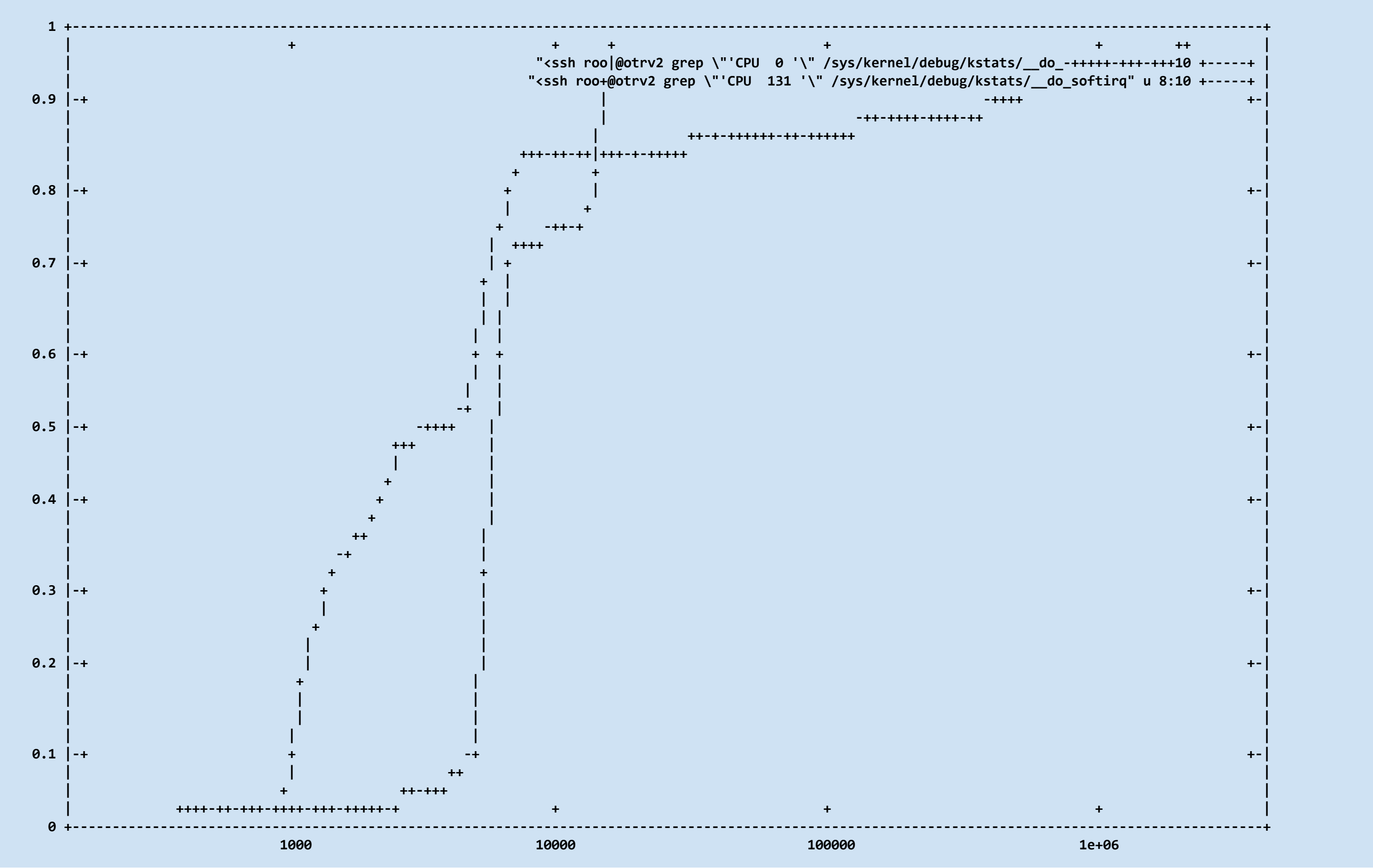
- gnuplot is one option for live monitoring

```
$ cd /sys/kernel/debug/kstats

$ echo reset > foo; watch grep "'CPU  6 '" foo    #  show one cpu
$ echo reset > foo; watch grep "'CPUS'" foo       # show totals

$ gnuplot
> set terminal dumb size 200,80 ansi256; set logscale x
> # plot distribution from a live machine. Replace 8:6 with 8:10 for cdf
> plot  "<ssh root@otrv2 grep CPUS /sys/kernel/debug/kstats/foo" u 8:6 w l
> # refresh data
> while (1) { pause 1; reread; replot; }
```

P5: Presentation example



P1: Instrumentation: dynamic probes

- linux supports dynamic code modification (think of breakpoints)
  - can attach a call to a kernel function in any point
  - through BPF, can make the kernel call user-supplied code

We can replace code changes with probes attached dynamically around the code under observation!

- beware:
  - the desired attach point may not exist in the binary due to optimization, inlining, etc.
  - likewise, variables of interest may have disappeared
  - invoking the callbacks relies on adapters, trampolines and multiple out-of-line data access

Dynamic probes add 100..1500ns to each sample

  - not systematic: the actual value depends on cache state
  - even larger impact in case of concurrency
  - this defeats some use cases, and may be very misleading in others

P1: Instrumentation: dynamic probes (2)

- Despite their limitations, dynamic probes ***are useful***, and come in 3+ forms:
- **kprobe** (attach callback to a function, or in principle anywhere)
    - a trampoline invokes the user callback with a copy of the registers
    - we need two of them, plus storage, to do the timing. **Cost: 100..500ns**
- **kretprobe** (attach callbacks on entry and return of a function)
    - a trampoline allocates temp storage, invokes the user callbacks on entry and on return.
    - Simpler to use, but more expensive. **Cost: 100..1500ns**
    - Current version serializes all entry and exit points. A fix for upstream is under review
- **tracepoints** (placeholder functions to attach callbacks)
    - these are manually added annotations, but many exist already
    - more convenient than k*probes, because arguments are passed explicitly to the user function

(There are also dedicated bpf hooks...)

P1: kstats and dynamic probes

kstats has builtin support to wrap a block of code with dynamic probes

- use kretprobe by default, also possible to track time between two places
- we added percpu support to kretprobes to remove a serialization point

```
$ cd /sys/kernel/debug/kstats

$ echo trace __do_softirq bits 4 > _control
$ echo reset > __do_softirq; watch grep "'CPUS'" __do_softirq # show totals
$ echo remove __do_softirq > _control

$ echo trace pcpu:__do_softirq > _control # pcpu avoids a lock on enter
$ watch grep "'CPUS'" __do_softirq

$ echo trace foo start __tracepoint_x end __tracepoint_y > _control
$ watch grep "'CPUS'" foo
```

Cost of dynamic probes

```
t0 = ktime_get_ns();
work(); // empty function
kstats_record(foo, ktime_get_ns() - t0);
```

Times in ns, each cell has two values:
cold cache (10 calls/s) / hot cache (>100k calls/ns)

kstats entry	p10	p50	p90	p99
foo # work untraced	30 / 20	50 / 30	60 / 30	60 / 30
foo # work traced	1500 / 230	1580 / 241	2100 / 241	4900 / 265
work	500 / 90	540 / 90	790 / 99	1050 / 120

Hot accuracy goes from **30ns (manual)** to **250ns with dynamic probes**.
Cold accuracy goes from **60ns (manual)** to **1050ns with dynamic probes**.
The traced function takes a large hit (200..4900us; it is 30..300ns for manual)

Some use cases

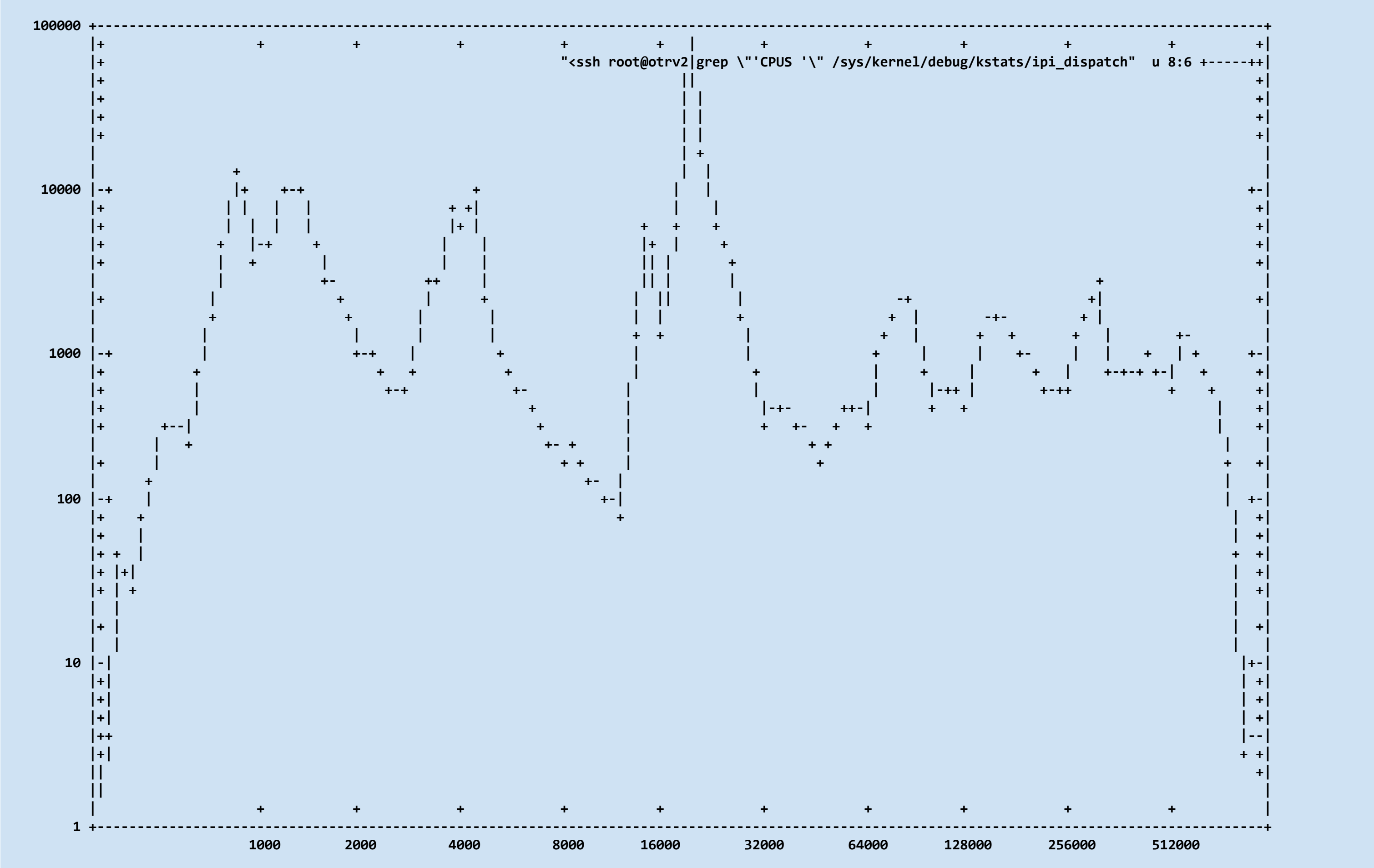
- IPI dispatch latency

```
include/linux/smp.h |  3 +++
kernel/smp.c        | 23 +++++++++++++++++++++++
```

- network rx latency (nic to socket to application)

- tx latency (tcp_sendmsg to xmit_one)

IPI latency (ns)



Related tools

- kstats' main feature is collect samples and compute distributions with **well defined accuracy and resolution**
- the same can be done attaching user-code to kretprobes via BPF.
  The following tools (and probably many others) have code for that:
  perf, bpftrace, systemtap, ext4dist, ...

Key differences:

- dynamic probes accuracy and overhead are inherently 10x worse than inline calls. May be improved with custom hooks (fenter, fexit) + kernel changes.
- kstats has programmable resolution (default 4 bits).
  Most other tools have 1 bit, hardwired (buckets are power of 2). This is fixable (but may take work depending on collection and presentation are entangled)

Conclusions

- kstats is very small and cheap at runtime (5ns hot, 300ns cold)
- accuracy and resolution may significantly limit usefulness
  - dynamic probes problematic/misleading for sub-microsecond measurements
  - use > 1bit resolution for fine-grained performance analysis

Code: https://lwn.net/Articles/813303/