January 08, 2020, TURIN

A Comparison of ftrace and LTTng for Tracing Baremetal and Virtualized Workloads

Authors:

- Emilio Bruno
- Dario Faggioli
- Enrico Bini

SUSE

Who we are

Emilio Bruno <emilio.bruno@edu.unito.it>

- Student @ University of Turin
- Internship @ SUSE

Dario Faggioli <dfaggioli@suse.com>

- Software Engineer, Virtualization Specialist @ SUSE

Enrico Bini <enrico.bini@unito.it>

- Professor @ University of Turin

An Introduction to Tracing

What is Tracing?

Tracing allows to “record information about a program's execution […] used by programmers for debugging purposes, and additionally […] to diagnose common problems with software”

Tracing vs Logging:

- Tracing is usually much more detailed
- Tracing can be enabled and disabled at a fine-grained level
- Tracing can be very “noisy” and usually adds more overhead

What kind of software can be traced?

- System software (kernel/hypervisors)
- Applications (user level programs)

Tracing with LTTng

Linux Trace Toolkit next generation (LTTng): “a powerful, open
Source set of tools to trace the Linux kernel and user applications
at the same time.”
https://lttng.org/

Distinctive features:

- (combined) Kernel and userspace tracing
- Trace files follows the Common Trace Format (CTF, https://diamon.org/ctf/)
- Low latency and low overhead tracing
- Traces can be analyzed offline or in real-time

Tracing with ftrace

Ftrace: “ftrace is a Linux kernel feature that lets you trace Linux kernel function calls. Essentially, it lets you look into the Linux kernel and see what it’s doing”
https://blogs.vmware.com/opensource/2019/11/12/ftrace-linux-kernel/

Distinctive features:

- Integrated in the Linux kernel
- Multiple and different tracers (function graph, stack, io, wakeup, …)
- More than 1000 events
- Specific tracers for latency analysis
- Shares infrastructure with other kernel performance tools (e.g., perf)
- Controlled from special filesystem (or specific tools)

Comparing LTTng and ftrace

Installation

LTTng:

- Kernel infrastructure
  - LTTng-modules
  - Out-of-tree modules. Not integrated inside Linux kernel! :-(
  - Build from source or use distro packages

- User space components
  - e.g., lttng program
  - Build from sources or use distro packages

Ftrace:

- Kernel infrastructure
  - Already integrated inside the kernel
  - Nothing to do! :-)

- User space components
  - e.g., trace-cmd program
  - Build from sources of use distro packages

(Simple) Usage Example

So, let's:

```
# perf bench sched pipe
```

And trace it's execution with both the frameworks!

Usage: Kernel Tracing

LTTng

```
# lttng create session-name \
    --output=/your/path
# lttng enable-event --kernel sched_'*'
# lttng enable-event --kernel \
    --syscall read,write
# lttng start
# perf bench sched pipe
# lttng destroy session-name
```

Ftrace

```
# trace-cmd record -p nop -e sched \
    -e syscalls:sys_enter_write \
    -e syscalls:sys_enter_read \
    -e syscalls:sys_exit_write \
    -e syscalls:sys_exit_read -- \
    perf bench sched pipe
```

Usage: Userspace Tracing

LTTng

liblttng-ust library:

- Provides tracef & tracelog APIs (similar to printf) or write your own “Tracepoint Provider” (https://lttng.org/docs/#doc-c-application)
- … out of the scope of this work!

Prebuilt user space tracing helpers for:

- liblttng-ust-libc-wrapper.so, for tracing: malloc(), calloc(), realloc(), free(), …
- liblttng-ust-pthread-wrapper.so, for tracing pthread_mutex_lock() (request and acquire time), pthread_mutex_unlock(), pthread_mutex_trylock()
- liblttng-ust-cyg-profile.so, for tracing (potentially) every function

Used as pre-loadable shared objects:

- $ LD_PRELOAD=liblttng-ust-pthread-wrapper.so <your_command>

Ftrace

(Next to) Nothing to be done! :-(

Only option is trace_marker (https://lwn.net/Articles/366796/)

But it is very limited

SUSE

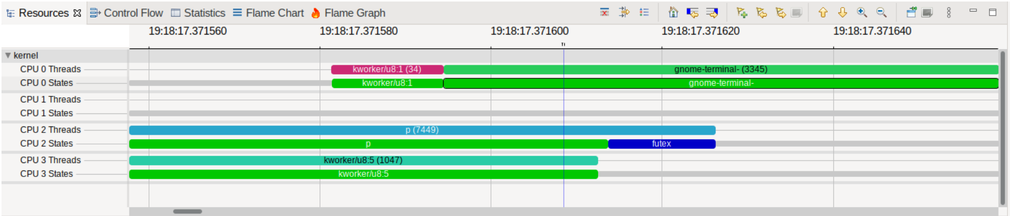
Usage example: Userspace + Kernel Tracing

The code

```c
int main(int argc, const char *argv[])
{
	pthread_t tid[N_THREADS];

	printf("Started process: %d\n", getpid());
	for (int i=0; i < N_THREADS; i++)
		pthread_create(&tid[i], NULL, &thread_body, NULL);

	for (int i=0; i < N_THREADS; i++)
		pthread_join(tid[i], NULL);

	exit(EXIT_SUCCESS);
}
```

```c
void* thread_body(void *arg)
{
	struct timespec time;
	pid_t tid;

#ifdef SYS_gettid
	tid = syscall(SYS_gettid);
#else
#error "SYS_gettid unavailable on this system"
#endif

	clock_gettime(CLOCK_MONOTONIC, &time);
	printf("%lu.%llu: Thread %d started (PID: %d)\n", time.tv_sec, time.tv_nsec, tid, getpid());
	sleep(1);
	printf("%lu.%llu: Thread %d requesting mutex=%x\n", time.tv_sec, time.tv_nsec, tid, &lock);
	pthread_mutex_lock(&lock);
	printf("%lu.%llu: Thread %d acquired mutex=%x\n", time.tv_sec, time.tv_nsec, tid, &lock);
	sleep(10);
	printf("%lu.%llu: Thread %d unlocking mutex=%x\n", time.tv_sec, time.tv_nsec, tid, &lock);
	pthread_mutex_unlock(&lock);
	sleep(1);
	printf("%lu.%llu: Thread %d ended\n", time.tv_sec, time.tv_nsec, tid);
	pthread_exit(NULL);
}
```

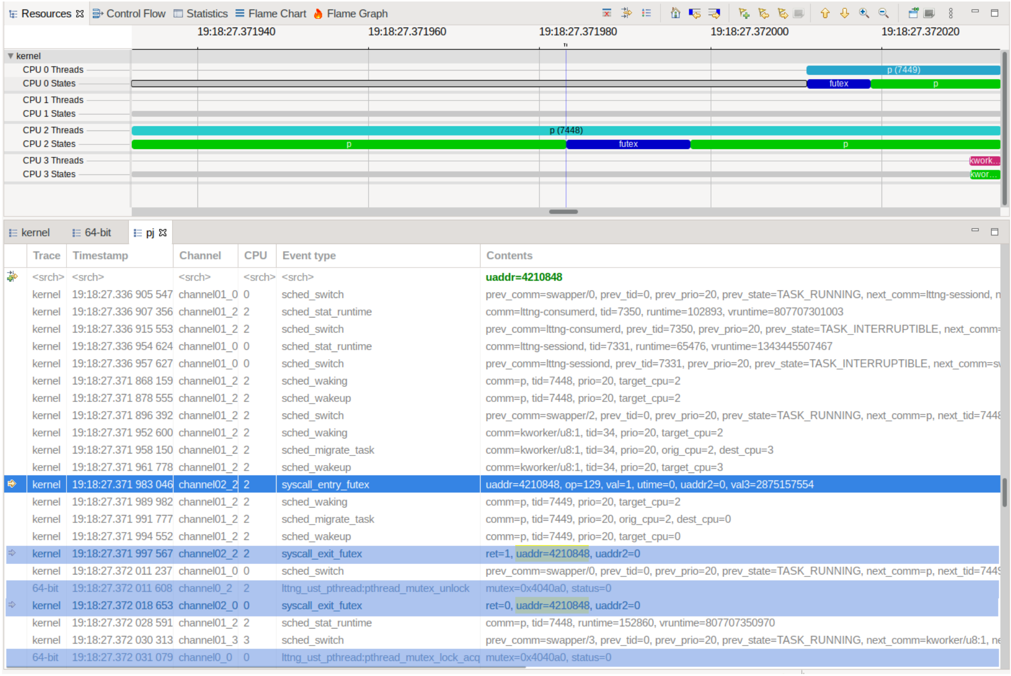
Trace Compass Resources view



Trace	Timestamp	Channel	CPU	Event type	Contents
<srch>	<srch>	<srch>	<srch>	<srch>	4040a0
kernel	19:18:17.371 561 760	channel01_3	3	sched_migrate_task	comm=gnome-terminal-, tid=3345, prio=20, orig_cpu=3, dest_cpu=0
kernel	19:18:17.371 562 405	channel01_2	2	sched_waking	comm=kworker/u8:1, tid=34, prio=20, target_cpu=2
kernel	19:18:17.371 566 447	channel01_2	2	sched_migrate_task	comm=kworker/u8:1, tid=34, prio=20, orig_cpu=2, dest_cpu=0
kernel	19:18:17.371 567 189	channel01_3	3	sched_wakeup	comm=gnome-terminal-, tid=3345, prio=20, target_cpu=0
kernel	19:18:17.371 570 582	channel01_2	2	sched_wakeup	comm=kworker/u8:1, tid=34, prio=20, target_cpu=0
kernel	19:18:17.371 578 987	channel01_2	2	sched_stat_runtime	comm=p, tid=7449, runtime=36012, vruntime=807668995974
kernel	19:18:17.371 579 727	channel01_3	3	sched_stat_runtime	comm=kworker/u8:5, tid=1047, runtime=103329, vruntime=1085298042134
kernel	19:18:17.371 581 350	channel01_0	0	sched_switch	prev_comm=swapper/0, prev_tid=0, prev_prio=20, prev_state=TASK_RUNNING, next_comm=kworker/u8:1, ne
kernel	19:18:17.371 589 329	channel01_0	0	sched_stat_runtime	comm=kworker/u8:1, tid=34, runtime=19959, vruntime=1343397431087
kernel	19:18:17.371 594 442	channel01_0	0	sched_switch	prev_comm=kworker/u8:1, prev_tid=34, prev_prio=20, prev_state=TASK_DEAD, next_comm=gnome-terminal-
kernel	19:18:17.371 601 002	channel01_3	3	sched_stat_runtime	comm=kworker/u8:5, tid=1047, runtime=21870, vruntime=1085298064004
64-bit	19:18:17.371 608 457	channel0_2	2	lttng_ust_pthread:pthread_mutex_lock_req	mutex=0x4040a0
kernel	19:18:17.371 612 439	channel01_3	3	sched_switch	prev_comm=kworker/u8:5, prev_tid=1047, prev_prio=20, prev_state=TASK_DEAD, next_comm=swapper/3, ne
kernel	19:18:17.371 613 634	channel02_2	2	syscall_entry_futex	uaddr=4210848, op=128, val=2, utime=0, uaddr2=0, val3=0
kernel	19:18:17.371 622 356	channel01_2	2	sched_stat_runtime	comm=p, tid=7449, runtime=43622, vruntime=807669039596
kernel	19:18:17.371 626 214	channel01_2	2	sched_switch	prev_comm=p, prev_tid=7449, prev_prio=20, prev_state=TASK_INTERRUPTIBLE, next_comm=swapper/2, ne
kernel	19:18:17.371 767 588	channel01_0	0	sched_stat_runtime	comm=gnome-terminal-, tid=3345, runtime=177302, vruntime=1343401898504
kernel	19:18:17.371 771 062	channel01_0	0	sched_switch	prev_comm=gnome-terminal-, prev_tid=3345, prev_prio=20, prev_state=TASK_INTERRUPTIBLE, next_comm
kernel	19:18:17.382 063 109	channel01_0	0	sched_waking	comm=gnome-terminal-, tid=3345, prio=20, target_cpu=0
kernel	19:18:17.382 072 369	channel01_0	0	sched_wakeup	comm=gnome-terminal-, tid=3345, prio=20, target_cpu=0
kernel	19:18:17.382 088 654	channel01_0	0	sched_switch	prev_comm=swapper/0, prev_tid=0, prev_prio=20, prev_state=TASK_RUNNING, next_comm=gnome-terminal-
kernel	19:18:17.382 751 944	channel01_0	0	sched_stat_runtime	comm=gnome-terminal-, tid=3345, runtime=680073, vruntime=1343402578577

SUSE

Trace Compass Resources view



SUSE

Traces Sizes

Traces size:

- LTTng: ~ 300 MB
- Ftrace: ~ 900 MB

Textual Analysis of the Traces

SUSE

Analyzing in a Terminal

LTTng

Babeltrace (version 2)

- APIs and command line tools
- Reference parser implementation for CTF
- https://babeltrace.org/

Ftrace

- In theory, no tool required at all:
  - All there already in the special filesystem
- More convenient:
  - trace-cmd
  - https://man7.org/linux/man-pages/man1/trace-cmd.1.html

A babeltrace2 Example

```
babeltrace2 /your/path/*
```

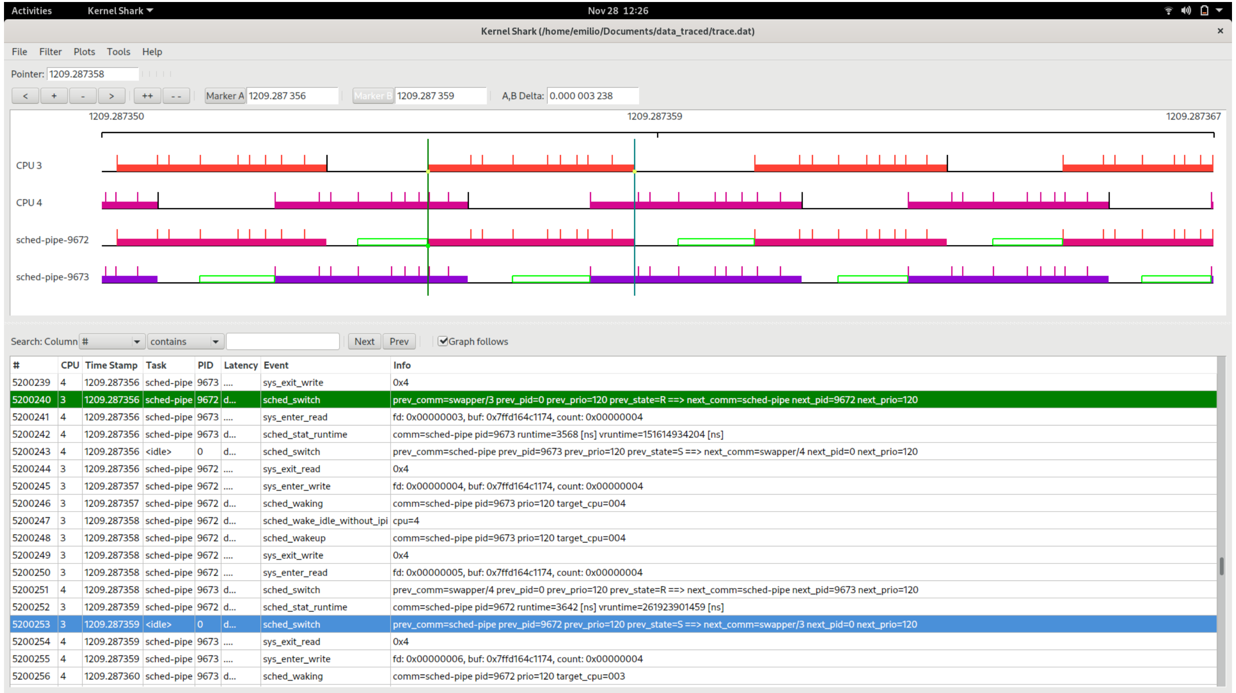
```
[12:52:41.040710281] (+0.000000034) DESKTOP-RCTBS9G.lan sched_switch: { cpu_id
↪   = 11 }, { prev_comm = "swapper/11", prev_tid = 0, prev_prio = 20,
↪   prev_state = ( "TASK_RUNNING" : container = 0 ), next_comm = "sched-pipe",
↪   next_tid = 11506, next_prio = 20 }
[12:52:41.040710517] (+0.000000236) DESKTOP-RCTBS9G.lan sched_stat_runtime: {
↪   cpu_id = 2 }, { comm = "sched-pipe", tid = 11505, runtime = 3364, vruntime
↪   = 356554408889 }
[12:52:41.040710748] (+0.000000231) DESKTOP-RCTBS9G.lan sched_switch: { cpu_id
↪   = 2 }, { prev_comm = "sched-pipe", prev_tid = 11505, prev_prio = 20,
↪   prev_state = ( "TASK_INTERRUPTIBLE" : container = 1 ), next_comm =
↪   "swapper/2", next_tid = 0, next_prio = 20 }
[12:52:41.040711063] (+0.000000315) DESKTOP-RCTBS9G.lan syscall_exit_read: {
↪   cpu_id = 11 }, { ret = 4, buf = 140734060990340 }
```

A trace-cmd Example

```
trace-cmd report

    <idle>-0     [001]  1207.982219: sched_switch:          swapper/1:0 [120] R
    ↪  ==> sched-pipe:9673 [120]
sched-pipe-9672  [003]  1207.982220: sched_stat_runtime:    comm=sched-pipe
↪  pid=9672 runtime=3824 [ns] vruntime=260989436840 [ns]
sched-pipe-9673  [001]  1207.982220: sys_exit_read:         0x4
sched-pipe-9672  [003]  1207.982220: sched_switch:          sched-pipe:9672
↪  [120] S ==> swapper/3:0 [120]
sched-pipe-9673  [001]  1207.982220: sys_enter_write:       fd: 0x00000006,
↪  buf: 0x7ffd164c1174, count: 0x00000004
sched-pipe-9673  [001]  1207.982221: sched_waking:          comm=sched-pipe
↪  pid=9672 prio=120 target_cpu=003
sched-pipe-9673  [001]  1207.982221: sched_wake_idle_without_ipi: cpu=3
sched-pipe-9673  [001]  1207.982221: sched_wakeup:          sched-pipe:9672
↪  [120] success=1 CPU:003
sched-pipe-9673  [001]  1207.982222: sys_exit_write:        0x4
    <idle>-0     [003]  1207.982222: sched_switch:          swapper/3:0 [120] R
    ↪  ==> sched-pipe:9672 [120]
```

Graphical Analysis of the Traces

KernelShark (v1.2)

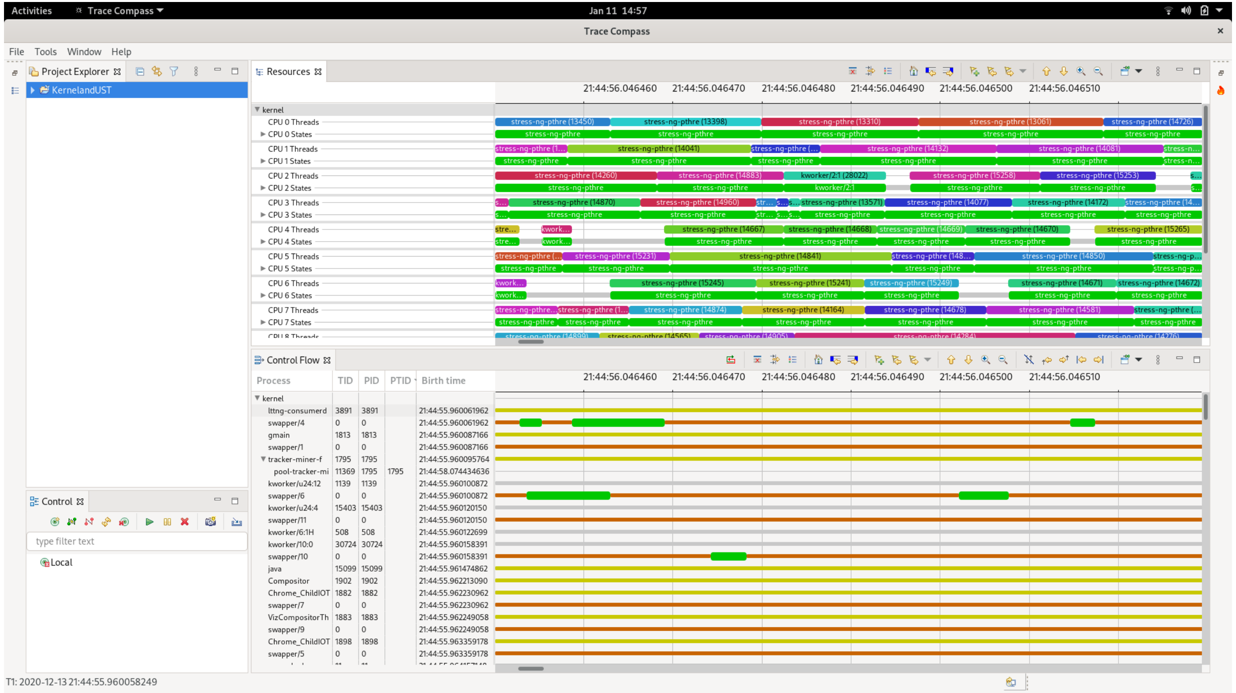


A graphical front end for trace-cmd
- Per-CPU activity plot
- Per-task activity plot
- https://kernelshark.org/

Some remarks:
- Provided views are very clear and useful
- It’s in active development
- Lags and delays, even with not so big traces!

SUSE

Trace Compass



Versatile traces and logs analyzer

- Supports multiple tracing formats:
- Both ftrace and LTTng!
- Provides multiple views
- https://www.eclipse.org/tracecompass/

Some remarks:

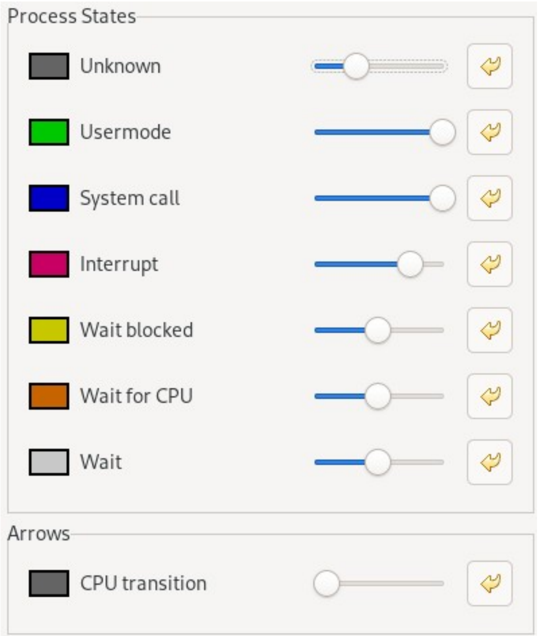
- Part of the Eclipse Framework
- Versatile and powerful
- No lags, UX stays smooth and fluid

SUSE

Trace Compass Views

SUSE

Flow Control View



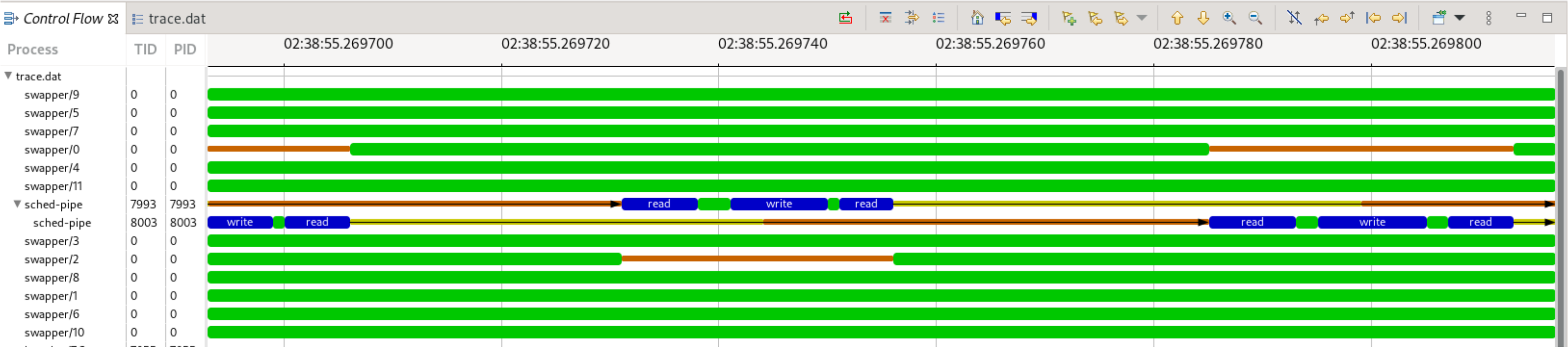
<<What does a task do?>>

<<What's going on inside all the tasks at a given time?>>

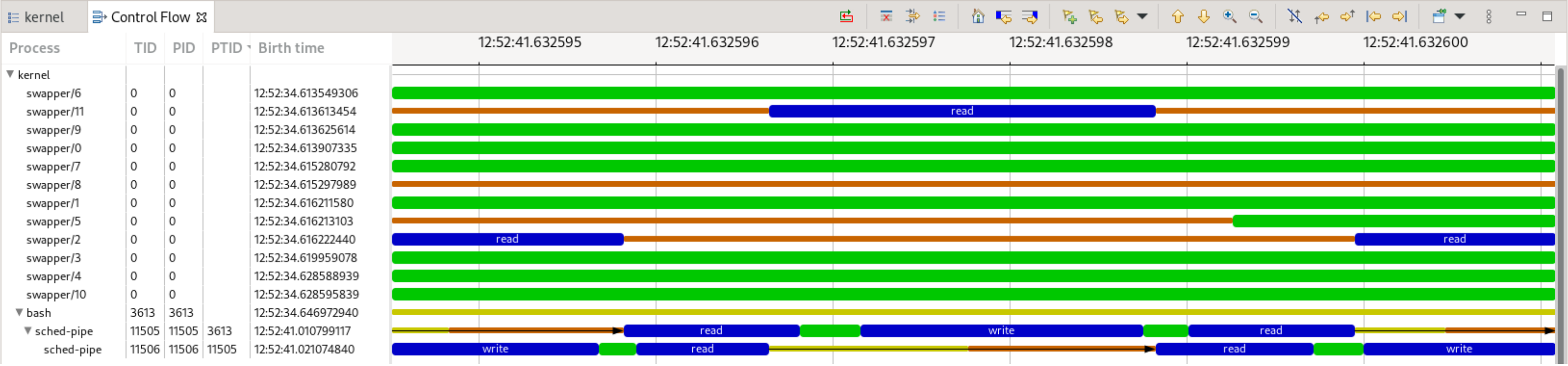
Tasks are on "rows"

Colors correspond to different states/actions

Flow Control View



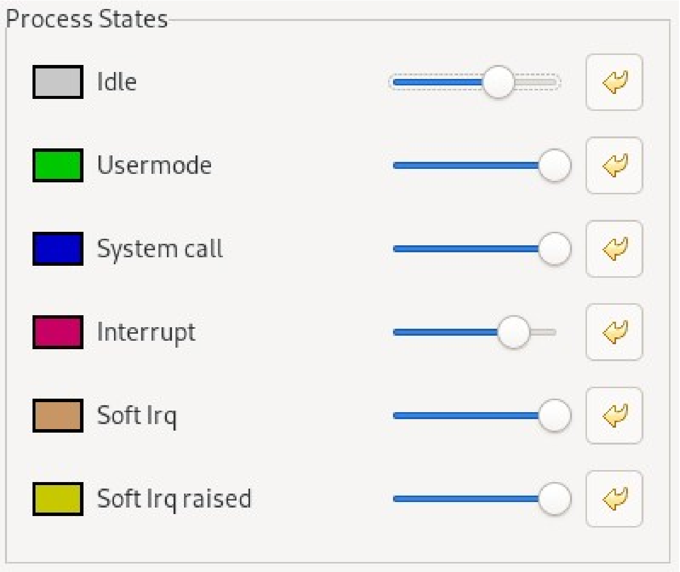
With an ftrace trace



With an LTTng trace

SUSE

Resources View



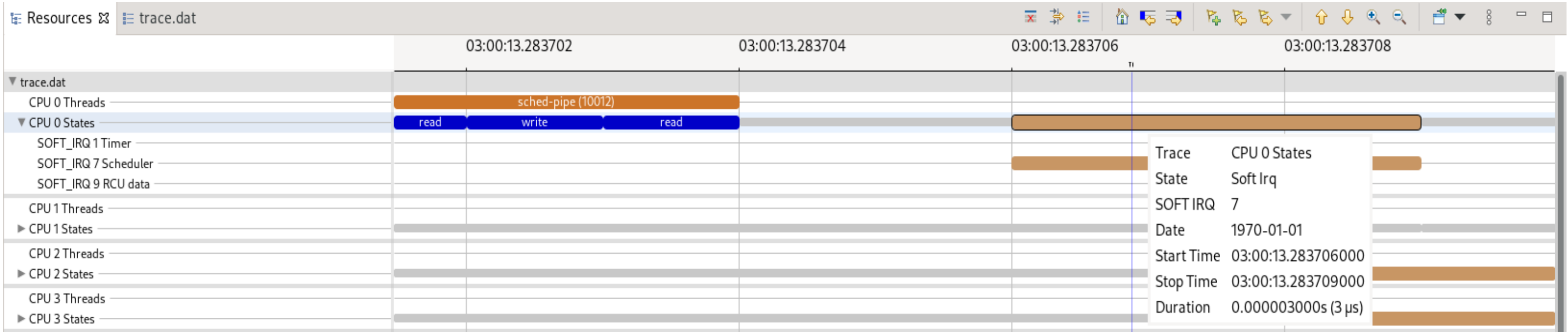
<<What is a CPU doing?>>

<<What is the state of all the CPUs at a given time?>>

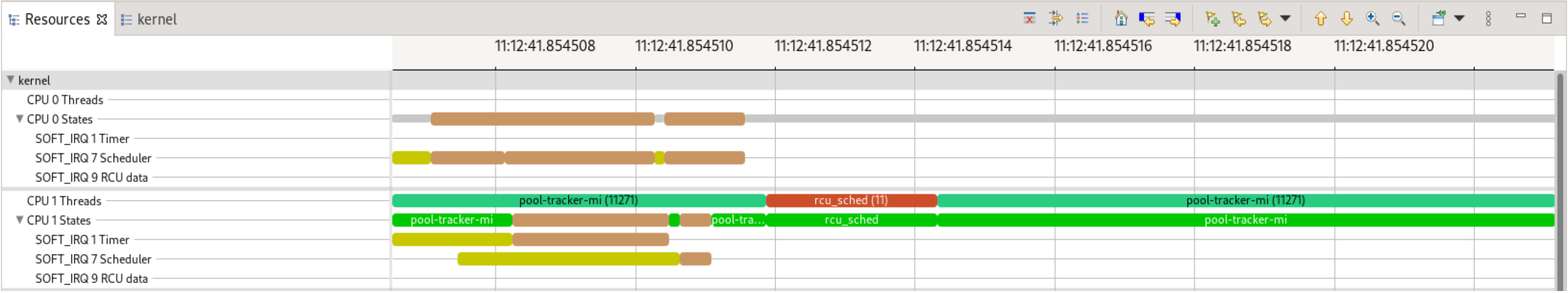
Each CPU plot (on “rows”) has:

- A thread “row”: shows the task running there
- A states “row”: colors corresponds to different states
- Dedicated “rows” for IRQ events

Resources View



With an ftrace trace



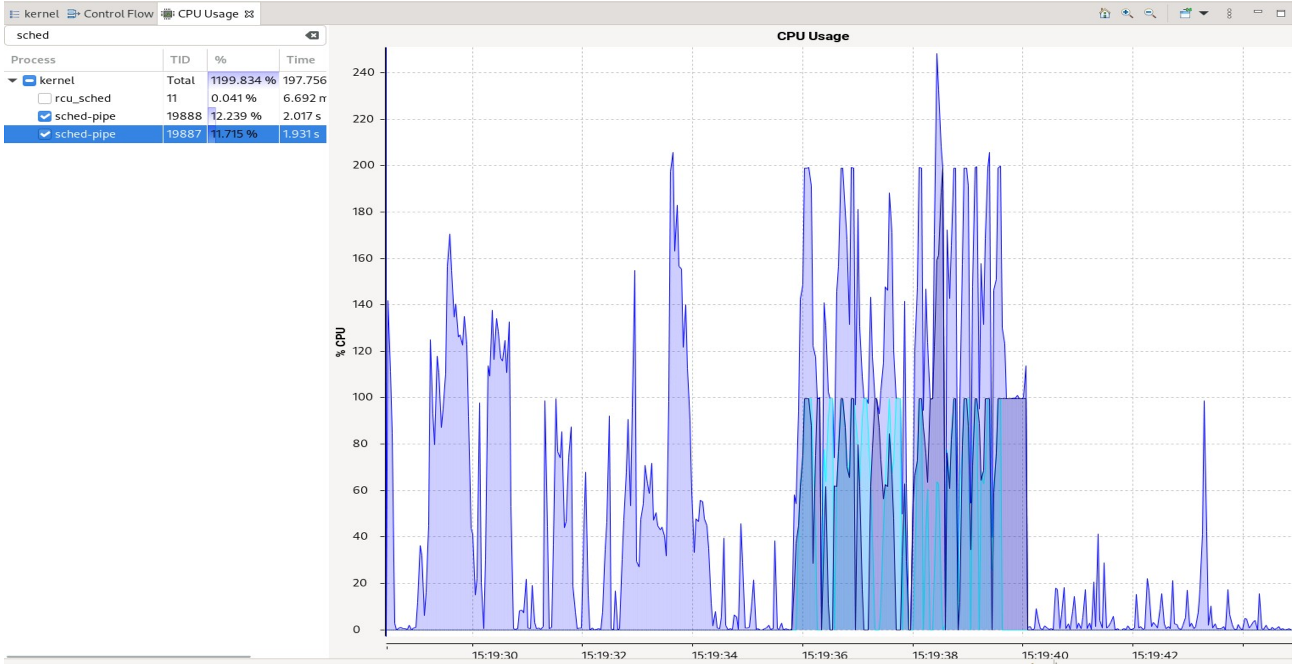
With an LTTng trace

SUSE

Other views

SUSE

CPU usage view



How much CPU each task used (with graphs)

SUSE

System call Latency view

Duration and latency of each syscall.

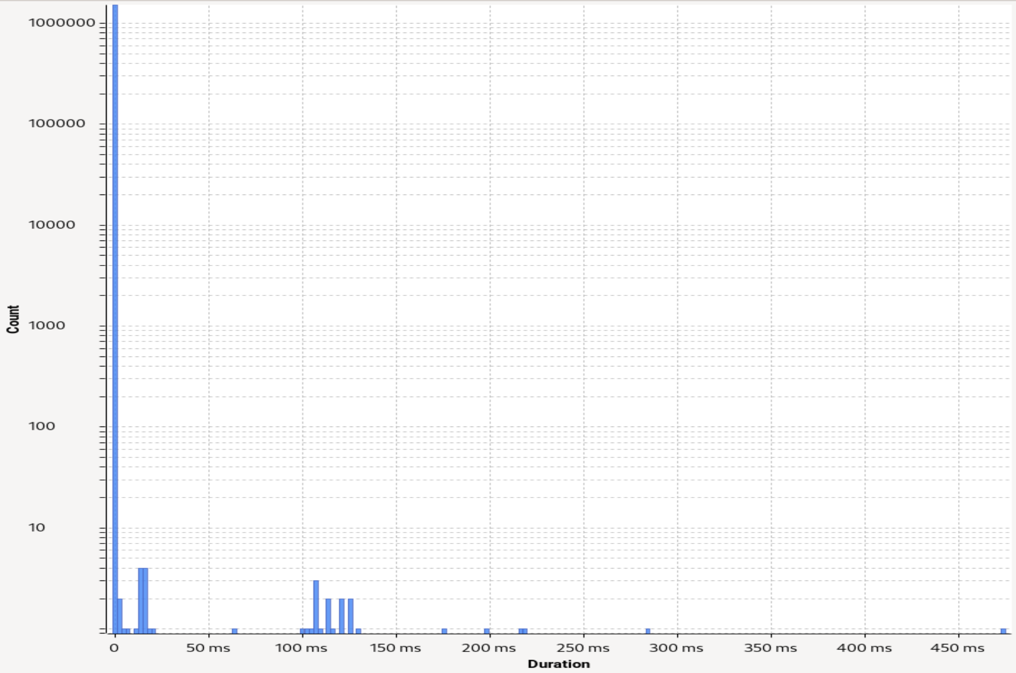
View divided in the following subviews:

- System call density
- System call latencies
- System call statistics
- System call latency vs time

System call Latency view

kernel | Control Flow | System Call Density

Duration	System Call	TID	Return value	Start Time	End Time
127	read	1447	-11	15:19:30.644 364 517	15:19:30.644 364 644
132	read	1447	-11	15:19:33.732 307 548	15:19:33.732 307 680
137	read	1447	-11	15:19:30.604 348 510	15:19:30.604 348 647
137	read	1447	-11	15:19:30.668 349 439	15:19:30.668 349 576
138	read	1447	-11	15:19:30.468 357 108	15:19:30.468 357 246
138	read	1447	-11	15:19:30.436 356 832	15:19:30.436 356 970
139	read	1447	-11	15:19:29.792 352 790	15:19:29.792 352 929
140	read	1447	-11	15:19:29.840 369 971	15:19:29.840 370 111
140	read	2449	-11	15:19:37.468 971 505	15:19:37.468 971 645
141	read	1447	-11	15:19:33.712 316 452	15:19:33.712 316 593
141	read	1447	-11	15:19:30.648 371 446	15:19:30.648 371 587
142	read	1447	-11	15:19:30.432 358 624	15:19:30.432 358 766
143	read	2449	-11	15:19:37.468 967 343	15:19:37.468 967 486
144	read	1447	-11	15:19:30.452 347 995	15:19:30.452 348 139
144	read	1447	-11	15:19:30.592 362 264	15:19:30.592 362 408
144	read	1447	-11	15:19:29.734 368 030	15:19:29.734 368 174
145	read	1447	-11	15:19:29.904 354 599	15:19:29.904 354 744
146	read	1447	-11	15:19:30.576 352 624	15:19:30.576 352 770
146	read	1447	-11	15:19:30.616 355 541	15:19:30.616 355 687
147	read	1447	-11	15:19:30.428 350 745	15:19:30.428 350 892
149	read	1447	-11	15:19:33.704 312 857	15:19:33.704 313 006
149	write	1988	8	15:19:28.509 086 148	15:19:28.509 086 297
149	write	1988	8	15:19:28.509 762 037	15:19:28.509 762 186
150	read	1447	-11	15:19:30.524 353 912	15:19:30.524 354 062
150	write	1988	8	15:19:28.515 808 598	15:19:28.515 808 748
151	read	1447	-11	15:19:29.768 359 158	15:19:29.768 359 309
151	write	1988	8	15:19:28.509 536 966	15:19:28.509 537 117
151	write	1988	8	15:19:28.509 812 713	15:19:28.509 812 864
151	write	1988	8	15:19:28.514 807 298	15:19:28.514 807 449
152	write	1988	8	15:19:28.513 430 470	15:19:28.513 430 622
152	write	1988	8	15:19:28.516 830 578	15:19:28.516 830 730
153	read	2449	-11	15:19:37.468 774 972	15:19:37.468 775 125
153	write	1988	8	15:19:28.515 147 012	15:19:28.515 147 165
153	write	1988	8	15:19:28.516 082 078	15:19:28.516 082 231
153	write	1988	8	15:19:28.517 236 622	15:19:28.517 236 775



SUSE

System call Latency view

kernel | System Call Latencies

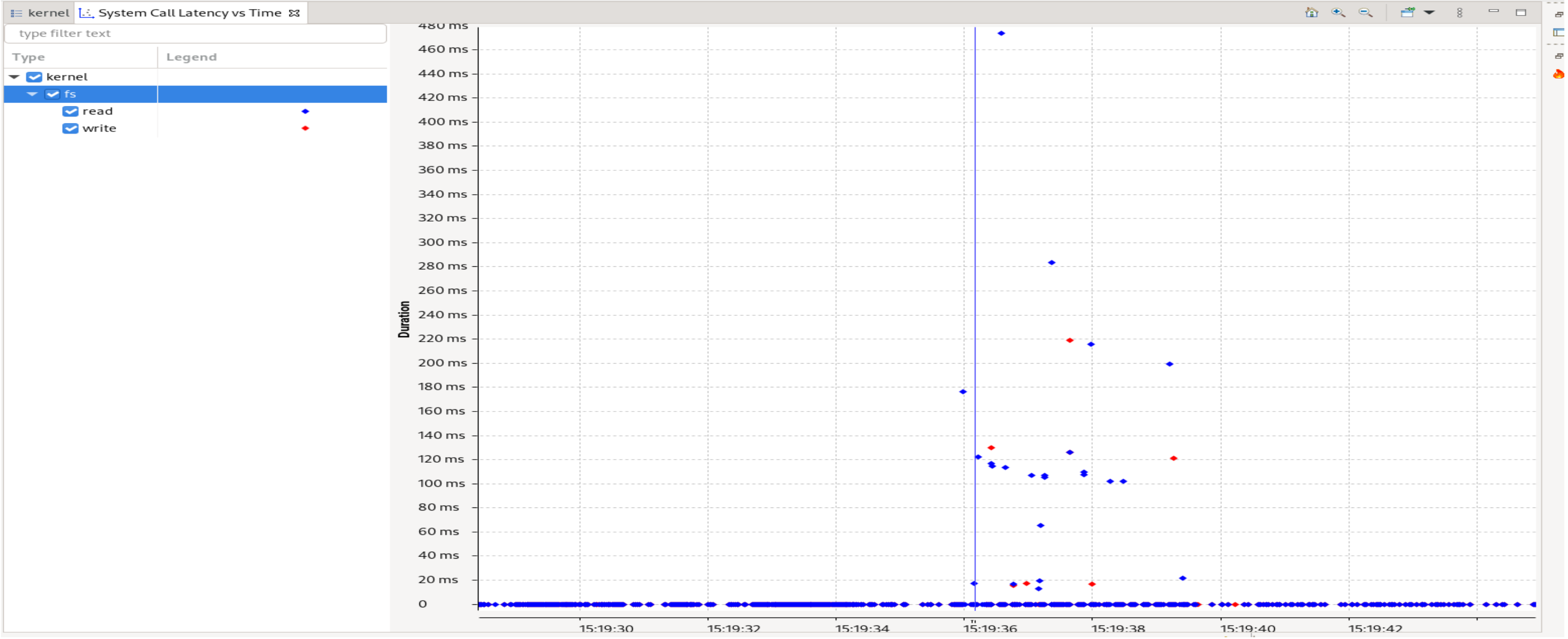
Start Time	End Time	Duration	System Call	TID	Return value	Component	File
15:19:36.159 556 440	15:19:36.159 558 827	2,387	read	19887	4	fs	fs/read_write.c
15:19:36.994 796 527	15:19:36.994 799 161	2,634	read	19888	4	fs	fs/read_write.c
15:19:36.566 572 294	15:19:37.042 773 027	476,200,733	read	0	4	fs	fs/read_write.c
15:19:37.182 804 802	15:19:37.248 869 012	66,064,210	read	0	4	fs	fs/read_write.c
15:19:37.355 000 463	15:19:37.640 327 921	285,327,458	read	0	4	fs	fs/read_write.c
15:19:37.640 328 148	15:19:37.860 589 118	220,260,970	write	0	4	fs	fs/read_write.c
15:19:39.010 930 887	15:19:39.010 935 262	4,375	read	19887	4	fs	fs/read_write.c
15:19:39.199 294 439	15:19:39.399 530 657	200,236,218	read	0	4	fs	fs/read_write.c
15:19:35.889 421 703	15:19:35.889 425 688	3,985	read	19888	4	fs	fs/read_write.c
15:19:35.892 196 725	15:19:35.892 201 139	4,414	write	1447	1	fs	fs/read_write.c
15:19:35.896 895 339	15:19:35.896 898 219	2,880	read	19888	4	fs	fs/read_write.c
15:19:35.929 323 713	15:19:35.929 326 272	2,559	read	19887	4	fs	fs/read_write.c
15:19:35.931 494 773	15:19:35.931 497 304	2,531	read	19888	4	fs	fs/read_write.c
15:19:35.979 847 465	15:19:36.156 980 906	177,133,441	read	19888	4	fs	fs/read_write.c

System call Latency view

kernel | System Call Latency Statistics

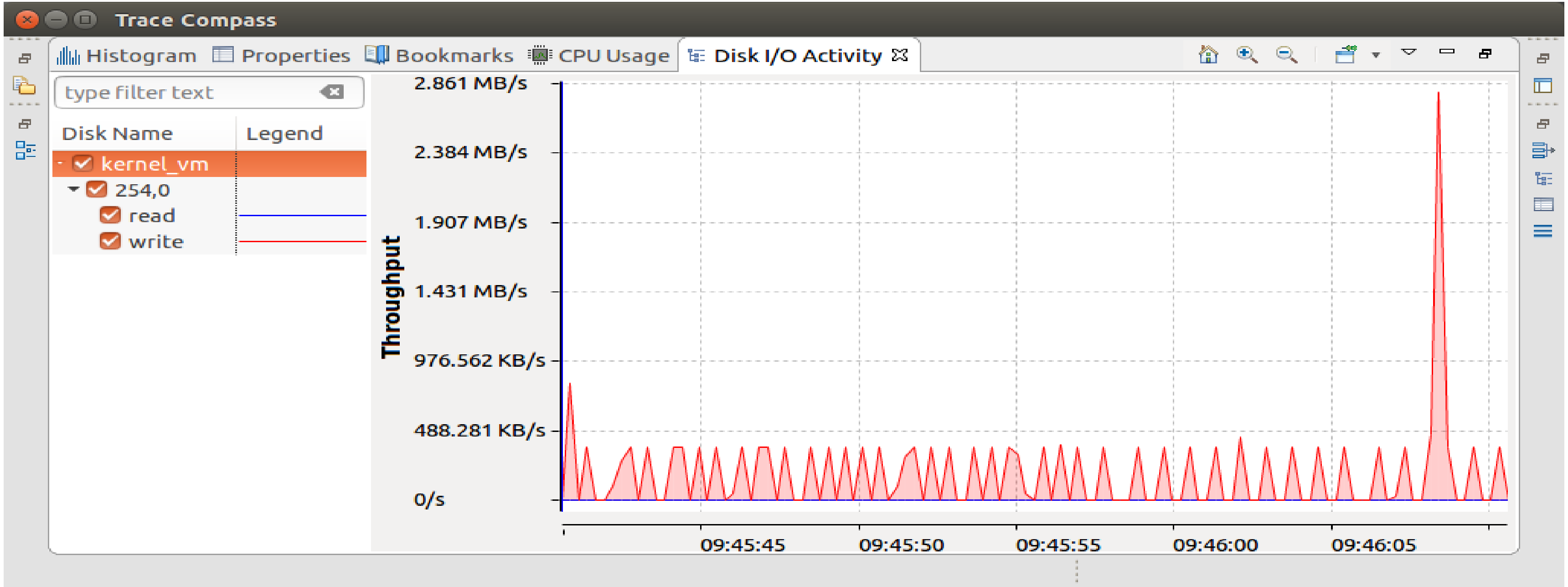
Level	Minimum	Maximum	Average	Standard Deviation	Count	Total
kernel						
Total	127 ns	476.201 ms	4.117 µs	668.464 µs	1524740	6.278 s
read	127 ns	476.201 ms	6.353 µs	874.887 µs	762648	4.845 s
write	149 ns	220.261 ms	1.88 µs	357.798 µs	762092	1.433 s
Selection	865 ns	2.582 µs	2.098 µs	826 ns	4	8.391 µs
read	2.387 µs	2.582 µs	2.509 µs	106 ns	3	7.526 µs
write	865 ns	865 ns	865 ns	---	1	865 ns

System call Latency view



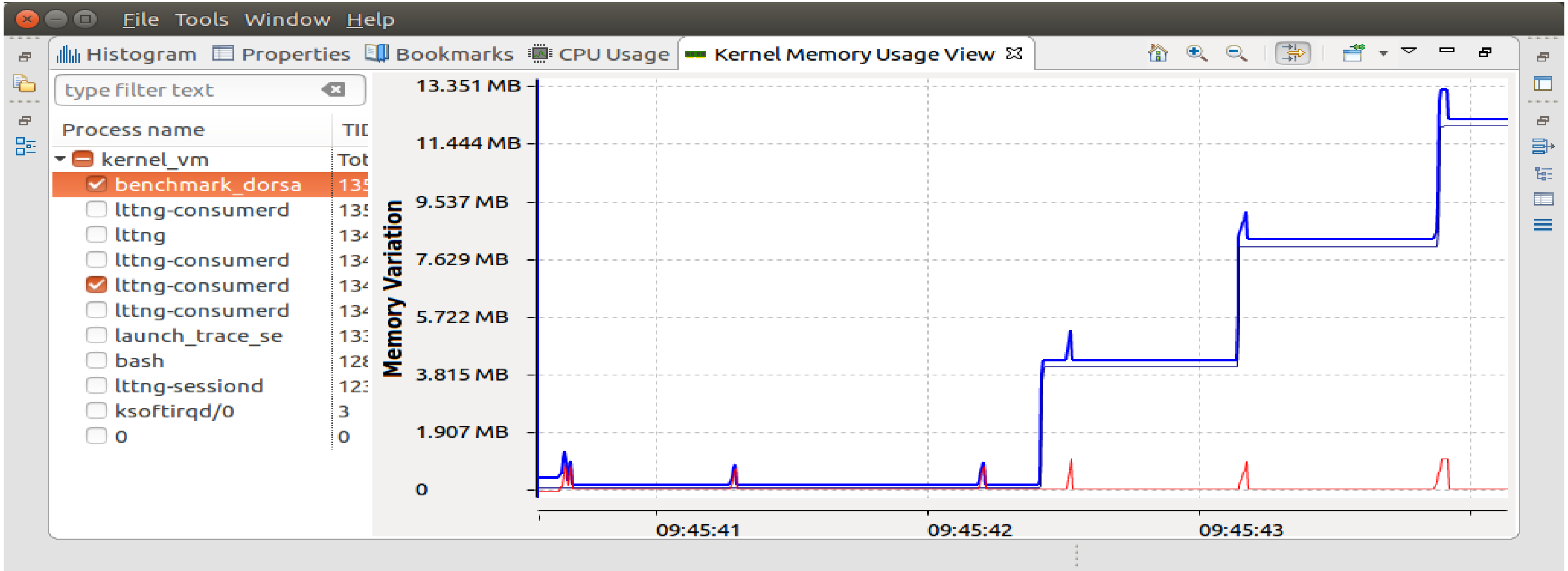
SUSE

Disk IO view



Read & Write throughput

Memory usage view



Tasks contribution to system memory usage

SUSE

Tracing in Virtualized Environments

Tracing & Virtualization

Virtualization
- 1 Host
- 1 or more Virtual Machines (Guests)

- Typically the interesting workload runs in the guest(s)

- KVM Model
  - Guests have virtual CPUs (vcpus)
  - Each vcpu is a process

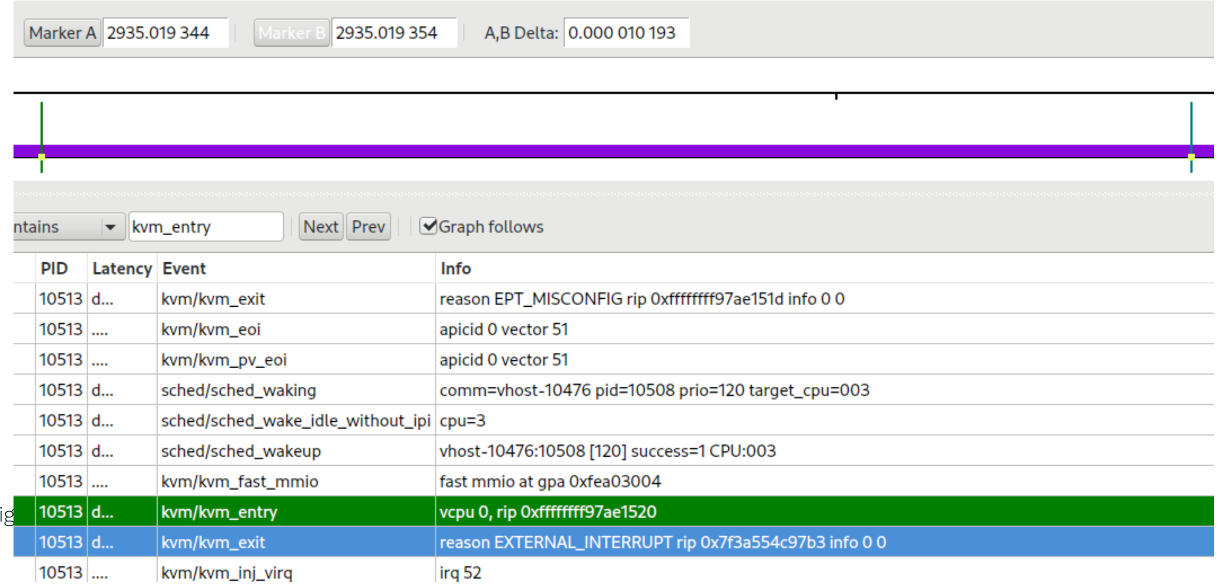
- Tracing

  - On the host:
    - Events generated by what is running on the host
    - Including the guests' vcpus

  - Inside the guests:
    - Events generated by what runs in the guest

Tracing On The Host

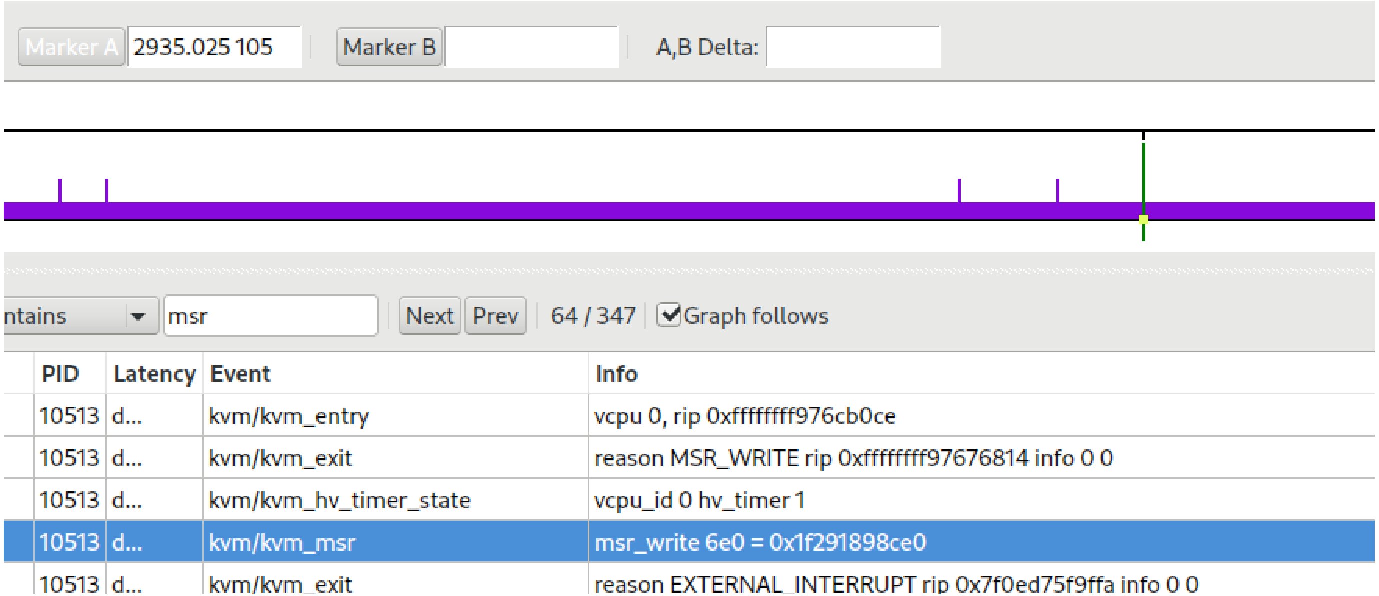
- We **can** see when the guests' vcpus run
- We **can** see hypervisor related events (e.g., kvm/kvm_entry)
- We **can't** see what triggered them, from inside the guests



PID	Latency	Event	Info
10513	d...	kvm/kvm_exit	reason EPT_MISCONFIG rip 0xffffffff97ae151d info 0 0
10513	....	kvm/kvm_eoi	apicid 0 vector 51
10513	....	kvm/kvm_pv_eoi	apicid 0 vector 51
10513	d...	sched/sched_waking	comm=vhost-10476 pid=10508 prio=120 target_cpu=003
10513	d...	sched/sched_wake_idle_without_ipi	cpu=3
10513	d...	sched/sched_wakeup	vhost-10476:10508 [120] success=1 CPU:003
10513	....	kvm/kvm_fast_mmio	fast mmio at gpa 0xfea03004
10513	d...	kvm/kvm_entry	vcpu 0, rip 0xffffffff97ae1520
10513	d...	kvm/kvm_exit	reason EXTERNAL_INTERRUPT rip 0x7f3a554c97b3 info 0 0
10513	....	kvm/kvm_inj_virq	irq 52

Tracing Inside The Guest

- We **can** see events generated by the workload running inside the guest
- We **can't** see what hypervisor/host events they cause
- We **can't** know now what happens when one of "our" vcpu is not running



SUSE

Combined Host + Guest(s) Tracing

Traces need to be collected in host and guests at the same time

Traces need to be synchronized

Tools need to support combined host CPUs and vCPUs views

- We know that both ftrace and LTTng can do something in that regard
  - (with addons and/or using experimental versions)
- We will experiment with these features, in the remainder of the internship

SUSE

Tracing Overhead Analysis

Is Tracing Causing Overhead?

Let’s run:
- hackbench
- stres-ng (CPU workload)

HW Platform:
- Intel core i7-10750H, 12 CPUs (6 Cores, 2 Threads)

Let’s measure:
- With no tracing
- While tracing with LTTng
- While tracing with ftrace

Let’s compare!
- Different tracing options
- 200 runs (of each benchmark)
- Check if tracing makes things slower (in %)

Hackbench

Hackbench slowdown (lower == better) when tracing different sets of events with ftrace or LTTng

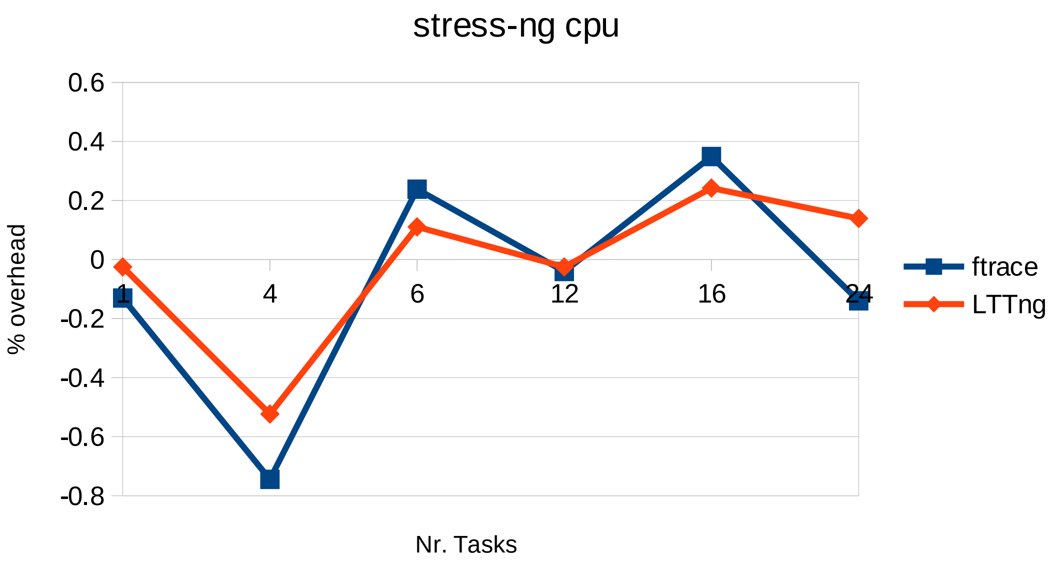
- Overhead is visible even when only tracing scheduling events
- LTTng overhead seems lower (but ftrace is probably tracing more events)

	ftrace	LTTng
Tracing only sched. events	3.00%	2.59%
Tracing all events	173.84%	106.79%

Stress-ng CPU

Stress-ng CPU workload slowdown (lower == better) when tracing **only scheduling events** with ftrace or LTTng

- Overhead is really small
- Overhead is the same for the two frameworks

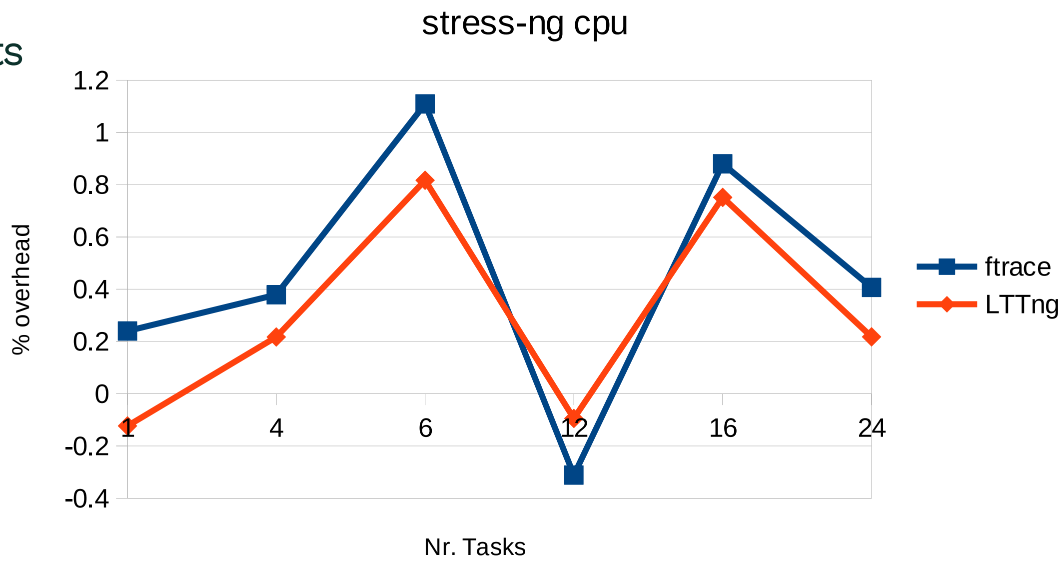


SUSE

Stress-ng CPU

Stress-ng CPU workload slowdown (lower == better) when tracing **all events** with ftrace or LTTng

- Overhead varies with the number of tasks used
- Overhead is the same for the two frameworks
- Overhead is small, despite the high number of events being traced



SUSE

Conclusions

SUSE

Conclusions

Kernel Tracing
- ftrace and LTTng are equally powerful and useful
- ftrace has the big advantage of being integrated in the kernel. Nothing is needed for starting using it!

User space & Combined Kernel and User space tracing
- LTTng can do it, ftrace can't!

Graphical Tools
- KernelShark is very handy as a trace-cmd front end
- Trace Compass is kind of heavy (comes with Eclipse, etc) but much more powerful and flexible

Overhead
- Tracing overhead is both workload and load dependent
- Overhead introduced by ftrace and LTTng is pretty much the same

For more information, contact SUSE at:
+1 800 796 3700 (U.S./Canada)
+49 (0)911-740 53-0 (Worldwide)

SUSE
Maxfeldstrasse 5
90409 Nuremberg
www.suse.com

Thank you!

Questions?

SUSE