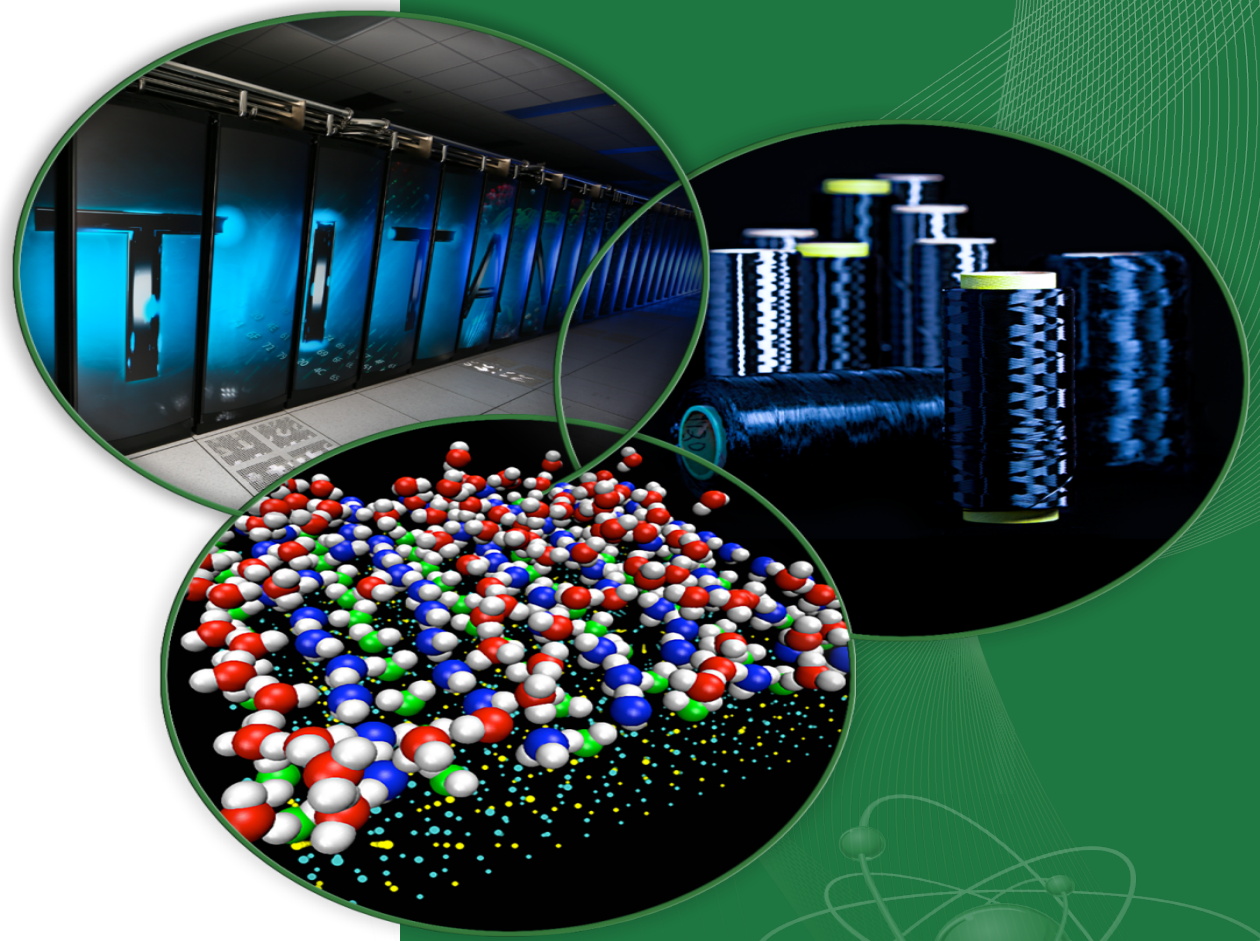
Tracing Lustre

New approach to debugging



Current Lustre debugging tools

- Utility lctl handles profiling
  - developed long before standard kernel profile
  - Can collect logs (lctl dk) or run as a debug daemon
  - Libcfs module parameters
    - Category based (lfsck, sec, …)
    - Subsystem based (mgs, llite, …)
    - lctl set_param debug=….
- Limitations
  - Disliked by upstream
  - Lacks advance filtering
  - Doesn’t scale well
  - Sometimes debug info gets lost
  - Lustre specific
- perf does everything lctl debug does and more

Tracing is magical

Using modern tools today on Lustre

- What Linux kernel supports
  - Trace events (perf)
    - Uprobes added in 3.5+ kernels
  - Ftrace (trace_cmd, perf for newer distros)
  - eBPF (bcc tools)
    - Needs 4.9+ kernel
- DWARF support
  - libunwind for old distros, libdw for new
  - perf record -F 99 --call-graph dwarf dd if=/dev/urandom of=/lustre/lustre/testfile.out
- DWARF2 utilities
  - Need debuginfo kernel ☹
  - pahole –C sk_buff vmlinux | less
- AutoFDO gcc plugin using perf (example of perf power)

Perf setup and usage issues

- Default perf is limited. Will most likely need to rebuild
- No stack walking
  - No indenting of perf output
  - Use libunwind/libdw
  - Other option use -fno-omit-frame-pointer
- No debug symbols (see only hexadecimal numbers)
  - Missing debuginfo package.
  - Might need to rebuild
- PMCs are missing on hypervisor systems and VMs
  - Use MSR (Model Specific Registers) instead
- Setting who can use perf
  - /proc/sys/kernel/perf_event_paranoid

Perf workflow

- perf list; perf stat; perf record; perf script or perf report
- Basic commands
  - perf top
  - perf stat “ls”
  - perf list
  - perf annotate
- Sharing perf results : perf archive perf.data
  - Debuginfo : /usr/lib/debug/.build-id/xx/xxxx…
    - Collection of build-id SHA1 checksums
    - Also can have ~/.debug/.build-id/xx/xxx…
  - perf buildid-cache –a; perf buildid list;
  - tar xvf perf.data.tar.bz2 -C ~/.debug

What perf can replace

- Strace
  - perf trace -e read,write dd if=/dev/urandom of=/lustre/lustre/testfile.out
- lctl set_param debug += trace
  - perf ftrace / trace_cmd ; replace ENTRY;EXIT;
  - perf probe –m “path to lnet.ko. -a ’lnet_*%return retval=$retval’ ; replace RETURN
- lctl set_param debug += malloc
  - perf kmem record dd if=/dev/urandom of=/lustre/lustre/testfile.out
  - perf stat -e kmem:kmalloc -e kmem:kfree dd if=/dev/urandom of=/lustre/lustre/testfile.out
  - Also examine L1-dcache*, dTLB-*, branch-*
  - pmu-tools (raw counters) – MESI states
    - ocperf.py record -e l2_lines_in.all -e l2_lines_in.e ...
- Lustre trace events will replace the rest

Lustre trace events

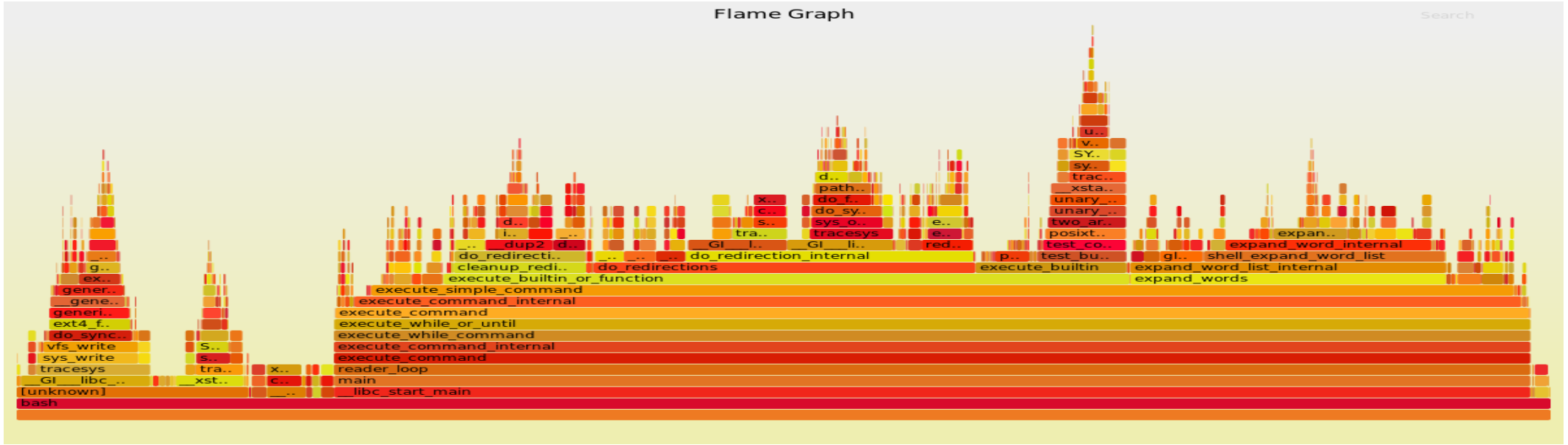
- LU-8980 – Current work to add trace events to Lustre 2.11
- Impact of moving to trace point
  - Can use standard tools like perf. Will add support to lctl as well.
    - lctl set_param debug=** works with tracepoint
    - Libcfs debug module parameter can turn on tracepoint classes
  - Move all 5000+ debugging statements to unique trace point events
    - Con: More complex to create debugging
      - No more simple strings like CDEBUG(“Hello world\n”); See libcfs_trace.h for example
      - tracepoint hates inline functions. True for kprobes as well.
      - No tracepoints in headers
    - Pro: Can filter many things related to the debugging
      - perf record -e libcfs:libcfs_ioctl --filter 'cmd == 3233310033' lnetctl net show’
      - Can do new things like histogram triggers (need 4.7+ kernels)
      - cat /sys/kernel/debug/tracing/events/libcfs/libcfs_ioctl/format
      - Can greatly reduce the scope of debugging. lctl set_param debug+=lfsck is heavy

Comparison of Lustre debug logs

- Standard lctl dk dump
  - 00004000:02000000:2.0F:1495061091.134329:0:14491:0:(linux-cpu.c:1098:cfs_cpu_init()) HW nodes: 2, HW CPU cores: 8, npartitions: 2
  - 100000001:01000000:3.0F:1495061091.406204:0:14491:0:(linux-crypto.c:376:cfs_crypto_performance_test()) Crypto hash algorithm adler32 speed = 1151 MB/s
  - 00000001:01000000:3.0:1495061091.656663:0:14491:0:(linux-crypto.c:376:cfs_crypto_performance_test()) Crypto hash algorithm crc32 speed = 1431 MB/s
  - 00000001:01000000:3.0:1495061091.906271:0:14491:0:(linux-crypto.c:376:cfs_crypto_performance_test()) Crypto hash algorithm crc32c speed = 7955 MB/s
  - 00004000:00000080:0.0:1495061256.496695:0:14615:0:(module.c:119:libcfs_ioctl()) libcfs ioctl cmd 3221775675
- Tracepoint dump
  - modprobe-14602 [005] .... 612.817160: libcfs_console_cpt_setup: (linux-cpu.c:1098:cfs_cpu_init) HW nodes: 2, HW CPU cores: 8, npartitions: 2
  - modprobe-14602 [004] .... 613.086729: libcfs_config_crypto: (linux-crypto.c:376:cfs_crypto_performance_test) Crypto hash algorithm adler32 speed = 1155 MB/s
  - modprobe-14602 [004] .... 613.336435: libcfs_config_crypto: (linux-crypto.c:376:cfs_crypto_performance_test) Crypto hash algorithm crc32 speed = 1427 MB/s
  - modprobe-14602 [004] .... 613.585950: libcfs_config_crypto: (linux-crypto.c:376:cfs_crypto_performance_test) Crypto hash algorithm crc32c speed = 7927 MB/s
  - lnetctl-14626 [004] .... 613.597451: libcfs_ioctl: (module.c:119:libcfs_ioctl) libcfs ioctl cmd 3221775675

Flame Graphs for Lustre

- git clone https://github.com/brendangregg/FlameGraph
  - By Brendan Gregg
- Example use
  - perf record -F 99 -a -g -- dd if=/dev/urandom of=/lustre/lustre/testfile.out
  - FlameGraphg]# ./flamegrpah.pl `perf script –I ~/perf.data | ./stackcollapse.pl` > perf-lustre.svg
  - No debuginfo no output



eBPF – Extended Berkley Packet Filters

- Needs a 4.4+ kernel
  - 4.4 : uprobes, kprobes, bpf output
  - 4.9 : stack trace, tracepoints, PMC + software events
- Developed for tcpdump in the 90s
  - tcpdump host 127.0.0.1 and port 22 -d (dump JIT code)
- Expanded to create a software defined network
  - Touch packets, define routes
- Changes to eBPF
  - Add more registers
  - Made virtual machine more powerful
  - Maps (key value stores)
  - more than sockets (kprobes etc)
- Instead of creating new module create BPF btye code and upload it

Lustre and eBPF

- eBPF is great at gather in time stats
  - Could replace lots of debugfs files
- Far more scalable for dynamic probing
- Far lower performance impact than even perf events
- Code is sand box so crashes don’t take down the system
- With eBPF can do chain graphs for kernel threads