Hewlett Packard
Enterprise

Detecting operating system noise with detect-detour

Clark Snyder, HPE
Dean Roe, HPE
Nagaraju K N, HPE

May 4, 2025

Problem statement

The issue

- Applications running at scale using bulk synchronous parallel model with frequent global synchronization or collective operations across all processes are sensitive to anything delaying ranks from reaching synchronization points

Operating system (OS) noise

- OS activities that prevent application from executing when it is ready
- OS noise sources
  - Context switching away from the application
  - Hardware interrupts
  - Software interrupts – timers, Read Copy Update (RCU) synchronization mechanisms, bottom half interrupt handlers, etc.

The challenge

- Identifying exact cause of OS noise

Existing tools

Some tools can detect noise

- But they are limited
  - Do not identify the cause
  - Use artificial workload which may not reflect noise experienced by applications
- Single-node tests:
  - **sysjitter**: fixed amount of work, reports times where work took longer than expected, shows noise duration
  - **FTQ**: fixed time quanta, results can be analyzed to see noise frequency
  - **FWQ**: fixed work quanta, similar to sysjitter
- Multi-node tests:
  - **P-SNAP**: fixed work quanta on each node, uses MPI to aggregate results from multiple nodes
  - **OSU MPI Allreduce**: Microbenchmark showing time taken for BSP global collective operations

Kernel osnoise tracer can identify causes of OS noise

- Unable to limit monitoring to a user-supplied set of processes
  - Can only monitor either monitor all processes or just its built-in artificial workload
- Difficult to set up
- Does not identify sources of noise by default

detect-detour

Goals

- Provide an easy-to-use tool that identifies OS noise sources
- Monitor any given processes for noise – allows monitoring real-world applications
- Use noise events to stop kernel ftrace on a detour – to dig deeper into noise sources if needed
- Provide single command with output to stdout/stderr for ease of use

Implementation

- Uses extended Berkely Packet Filter (eBPF) feature to insert code into kernel to monitor for and identify noise sources that affect user's application
  - Detects detours that meet user-supplied filter criteria and collects detour information for those detours
  - Queues detour events for user-space
- User-space code provides interface between kernel code and user
  - Compiles kernel code and uses eBPF to insert it into kernel
  - Receives detour events from kernel code
  - Reports detours to stdout
  - Manages interaction with kernel ftrace

detect-detour capabilities

Detour detection
- Identifies 3 detour types: Irq, Sched, Softirq – fully defined on next slide
- Identifies detour duration
- Monitors specific processes identified by process ID or name (--pids, --ppids, --pname, --ppname, optional, at least one is required)

Detour filtering
- Report detours whose duration exceeds a threshold (--threshold, required)
- Limit scheduling detour reports to involuntary process switch (--invol, optional)

Execution control
- One-shot – stop on detection of first detour (default)
- Continuous – keep running until stopped by timeout or manual interrupt (--cont, optional)
- Timeout – stop execution when time limit expires (--timeout, optional)

Kernel ftrace control
- Start / stop tracing on detect-detour startup and exit, respectively (--trace, optional)
- Copy trace buffer to files on each detected detour (--dir, optional)

Example output

```
# taskset -c 39 detect-detour.py --cont --pname do_work --threshold 5000
Number of processes monitored: 20
Tracing detour events continuously ...
 Hit Ctrl-c to end forcefully.
[298953.793452] Softirq: CPU:  15   Pid: 39794 [do_work]        Duration:        6389   SIRQ:       3 [NET_RX]
[298953.793481] Sched:   CPU:  15   Pid: 39794 [do_work]        Duration:       23033   PidNew: 39414 [sshd]
[298966.489938] Irq:     CPU:  11   Pid: 39790 [do_work]        Duration:       13164   IRQ:       78 [mlx5_async40@pci:0000:5c:00.0]
^CExiting due to Ctrl c....
```

- All detours show time of detour start, detour type, CPU, affected process ID and name, and detour duration in nanoseconds

Irq
- Hardware interrupt handler top half, i.e. running in interrupt context
- Identifies interrupt number and handler name

Sched
- Process switch by scheduler
- Identifies the new process ID and name

Softirq
- Software interrupt, includes IRQ handler bottom half, read copy update, tasklets, timers, network receive & send, block I/O
- Identifies software interrupt number and name

Example using ftrace

```
# taskset -c 255 detect-detour.py --pname allreduce --threshold 1000000 --dir /tmp/trace_data --cont --invol
Number of processes monitored: 128
Tracing detour events continuously ...
Hit Ctrl-c to end forcefully.
[076698.627186] Sched:   CPU:  70   Pid:  170709 [allreduce]      Duration:   587583546   PidNew:    1774 [khugepaged]
Target file=/tmp/trace_data/cpu70-20250408-121042958018
[076774.584549] Softirq: CPU:   1   Pid:  170640 [allreduce]      Duration:     1119729   SIRQ:         1 [TIMER]
Target file=/tmp/trace_data/cpu1-20250408-121158914459
[076821.688555] Softirq: CPU:   1   Pid:  170640 [allreduce]      Duration:     1125932   SIRQ:         1 [TIMER]
Target file=/tmp/trace_data/cpu1-20250408-121246018444
[076866.744597] Softirq: CPU:   1   Pid:  170640 [allreduce]      Duration:     1167454   SIRQ:         1 [TIMER]
Target file=/tmp/trace_data/cpu1-20250408-121331074467
```

- Kernel function_graph tracer was configured and started before running detect-detour
- Monitor allreduce processes:
  - For detours over 1 millisecond
  - Limit reporting of scheduling detours to only involuntary detours
  - Save a copy of the CPU trace buffer for each reported detour
- Found a 587 millisecond detour due to process “khugepaged”
  - Details are in function_graph trace output in `/tmp/trace_data/cpu70-20250408-121042958018`

function_graph trace output

```
# vi /tmp/trace_data/cpu70-20250408-121042958018

76698.781467 |    70)   3.637 us    |                                    }
76698.781468 |    70)   0.230 us    |                                    migrate_enable();
76698.781468 |    70)   0.211 us    |                                    rcu_read_unlock_strict();
76698.781468 |    70)   9.468 us    |                                  }
76698.781469 |    70)   0.221 us    |                                  enter_lazy_tlb();
76698.781470 |    70)   0.721 us    |                                  save_fpregs_to_fpstate();
 ------------------------------------------
  70) allredu-170709 =>  khugepa-1774
 ------------------------------------------

76698.781472 |    70)               |  finish_task_switch() {
76698.781472 |    70)   0.210 us    |    raw_spin_rq_unlock();
76698.781473 |    70)   1.162 us    |  }
76698.781473 |    70)               |  __timer_delete_sync() {
76698.781473 |    70)               |    __try_to_del_timer_sync() {
76698.781473 |    70)               |      lock_timer_base() {
76698.781474 |    70)   0.321 us    |        _raw_spin_lock_irqsave();
```

Future work

- Improve timestamp correlation with kernel ftrace
  - By default, ftrace clock is different from the one available to eBPF programs
  - Using ftrace “mono” clock looks promising
- When copying the trace buffer to a file, copy the per-CPU trace buffer
  - This will save only relevant traces and reduce time spent saving trace buffer
- Look into using ftrace snapshots as a low overhead replacement for copying trace buffers to files

Recommendations and Conclusion

Recommendations

- Pick the application to monitor
  - It should be the actual application or provide a reasonable approximation
- Start with thresholds in the millisecond range for OS noise investigations
  - Thresholds too low will create too much data and identify detours that, in practice, have minimal impact
  - Keeps rate of detected detours within the ability for detect-detour to process
- Bind the application and detect-detour to different CPUs
  - Prevents detect-detour from being identified as OS noise source
- Once a noise source is identified, look at ways to mitigate it, such as:
  - Eliminating it, e.g. disabling an unnecessary service
  - Changing when it occurs, e.g. run it between jobs
  - Isolate the noise source on a CPU dedicated to system overhead
  - Reducing the frequency at which it occurs or the duration of the detour

Conclusion

- detect-detour provides a powerful utility for identifying OS noise sources
- When used with the kernel ftrace capability it can help drill down into details of OS noise sources

Documentation

Links to documentation on HPE support site

- HPE Cray Supercomputing User Services Software Administration Guide for HPE Performance Cluster Manager (1.3.0) (S-8064)
  - OS Noise Detection section
- HPE Cray Supercomputing User Services Software Administration Guide: CSM on HPE Cray Supercomputing EX Systems (1.3.0) (S-8063)
  - OS Noise Detection section

Thank you

clark.snyder@hpe.com
dean.roe@hpe.com
nagaraju.kn@hpe.com