Detecting operating system noise with detect-detour

Clark Snyder
HPC
Hewlett Packard Enterprise
clark.snyder@hpe.com

Dean Roe
HPC
Hewlett Packard Enterprise
dean.roe@hpe.com

Nagaraju KN
HPC
Hewlett Packard Enterprise
nagaraju.kn@hpe.com

ABSTRACT

HPC applications, especially those that frequently perform global synchronization operations, can be negatively affected by background operating system (OS) activity. The background actions of interest are processing hardware interrupts, software interrupts, and process context switches. While these actions are necessary to the operation of the OS, from the application's point of view, they are viewed as "OS noise" that affects performance, and the system should be tuned to minimize them. Identifying sources of OS noise is crucial for application performance but can be difficult. Few options exist to identify sources of OS noise without getting into the intricacies of the underlying kernel internals. The `detect-detour` tool makes use of the Linux kernel extended Berkeley Packet Filter (eBPF) [1] feature to help system administrators identify sources of OS noise without requiring them to be kernel experts.

KEYWORDS

HPC, eBPF, Linux, kernel, noise, interrupt, ftrace, osnoise tracer, IRQ

1 Introduction

HPC applications that run at scale, especially those using the bulk synchronous parallel (BSP) programming paradigm and doing frequent global synchronization or collective operations across all the processes/ranks in the application, are very sensitive to anything that delays one or more ranks from reaching the synchronization point. Delays in one rank slow down the entire application.

One of the factors that affects synchronization of ranks is OS noise. Any operating system activity that is not directly invoked by the application and interferes with the time an application is executing on a processor can be considered OS noise. The main types of OS noise are:

- Context switching
- Hardware interrupts
- Software interrupts, such as handling timers, performing Read Copy Update (RCU) synchronization mechanisms, etc.

Each of these OS noise events cause the processor to detour away from running the application. Since these detours are not synchronized across ranks, their adverse effect on the performance of the HPC application is amplified as the number of ranks increase.

The goal for any HPC system is to have as few unsynchronized interruptions as possible to reduce application wait time at synchronization points. To achieve this, the factors causing unsynchronized interruptions must be properly identified and addressed.

It is often challenging for HPC users and administrators to identify OS noise and its source. Most of the time, it requires specialized skills such as an understanding of the underlying kernel internals. Tools that are currently available concentrate on identifying the amount of OS noise experienced by specific processes and fail to easily identify the origin of the noise for non-experts. The `detect-detour` tool described in this paper can more easily identify sources of OS noise in a user-friendly manner.

2 Existing tools

There are many tools for detecting noise, such as the following:

- `sysjitter` [2] runs on a single node and measures the time it takes to perform a fixed amount of work, labeling any deviation beyond a threshold as noise. While `sysjitter` is good at detecting any noise, since it is monitoring its own CPU bound artificial workload, it may detect noise that has no significant impact on real applications and not encounter noise generated when real applications are run.
- FTQ and FWQ [8] use a fixed time quantum and a fixed work quantum to measure noise on a node, respectively. FWQ is similar to `sysjitter`, providing a way to measure the amount of noise. Due to using a fixed time quantum, FTQ is useful to determine the frequency of noise.
- P-SNAP [3] is similar to `sysjitter`, in that it runs a calibrated CPU-bound workload, but it runs on multiple nodes and compares the performance across nodes. This provides a better view of the OS noise across a system.
- The OSU MPI allreduce benchmark, part of MVAPICH [4], can be used to detect the presence of OS noise in a system, though it can be affected by network variability as well.

These noise detection tools, while able to help detect some instances of OS noise, are not able to identify the source of the OS noise.

The Linux kernel provides a feature called the osnoise tracer [5], which is based on ftrace [6], to identify the amount of noise in the system. This tool works by creating per-CPU kernel threads and periodically collecting the time spent between two fixed operations. If the time spent is more than a fixed threshold, it is considered OS noise. It uses existing ftrace trace points in the kernel as well as kernel probes to collect noise data.

By default, it creates an osnoise workload and shows the noise experienced by the per-CPU threads created to run the workload. It can run without the per-CPU osnoise workload if the user prefers to run other workloads, but the data reported will be system-wide and cannot be limited to noise affecting specific application processes.

The osnoise tracer does not display the source of the noise by default. Additional tracing configuration settings must be utilized to gather data to help identify the source of the noise. There is no single command that can clearly show the source of noise experienced by a set of application processes. These limitations make it difficult to use when running applications across multiple nodes.

3 `detect-detour` implementation and usage

We created `detect-detour` to address the shortcomings in the current tools by providing these capabilities:

- Monitor specific processes to target the noise sources that impact the application
- Use noise events to control ftrace when additional detail is needed to understand noise sources
- Provide a single command with output to stdout/stderr to make it easy-to-use across any number of nodes

3.1 Implementation

`detect-detour` is composed of two pieces: a user-space component that acts as the main program, and a kernel component that collects the noise related data.

The kernel component is written in C and leverages the eBPF functionality in Linux. The kernel code is responsible for collecting the noise data and providing it to the user-space code.

The user-space component is written in Python. It is responsible for parsing the command line, compiling and inserting the kernel code into the kernel, passing parameters to the kernel code, receiving the noise data from the kernel code, and displaying the noise data to the user. The Python code takes advantage of the BPF Compiler Collection (BCC) [9] toolchain for eBPF, which allows the kernel C source code to be embedded in Python and dynamically compiled and installed when `detect-detour` is run.

The detour data is collected by the kernel code using event hooks for process context switch and hardware and software interrupt entry and exit points in the kernel. The kernel code is responsible for reporting only those detours that affect the monitored processes and exceed the minimum duration threshold. Using eBPF makes the code portable to multiple Linux distributions and across machine architectures without the need to recompile the kernel.

Figure 1 shows a pseudo code representation of how the kernel code uses the eBPF capability to attach to the software IRQ trace points to collect noise data from software IRQs. The code hooks are executed during software IRQ entry and exit points in the kernel. During the exit point, if the duration of the detour exceeds the time threshold supplied to `detect-detour`, additional event data about the detour is collected. The main Python program waits on the event notification from the kernel event hooks and, upon getting the notification, displays the event data.

Kernel logic

```
TRACEPOINT_PROBE (irq, softirq_entry) {
  if (current thread is being monitored) {
    save current time, process info, IRQ data
  }
}


TRACEPOINT_PROBE (irq, softirq_exit) {
  if (current thread is being monitored){
    calculate time delta between softirq entry and exit
    if (time delta > noise threshold) {
      retrieve detour data saved by entry handler
      send perf event to wake Python code
    }
  }
}
```

Python logic

```
while (detect-detour should remain active):
  if (perf event received from kernel code):
    extract detour data from perf event
    print detour data to stdout
```

Figure 1: Kernel and Python logic from `detect-detour`

Since running `detect-detour` creates noise, we used `taskset` to run it on a different CPU than those used by the monitored processes to ensure it did not affect the noise reported.

3.2 Detour Information

When `detect-detour` identifies a detour that exceeds the specified threshold, it prints the following information:

```
# taskset -c 255 detect-detour.py --pname allreduce --threshold 300000 --cont --invol
Number of processes monitored: 128
Tracing detour events continuously ...
 Hit Ctrl-c to end forcefully.
[445283.508877] Softirq: CPU:  20   Pid:  748187 [allreduce]      Duration:      349933   SIRQ:         1 [TIMER]
[445349.124597] Softirq: CPU:  32   Pid:  748199 [allreduce]      Duration:      578732   SIRQ:         3 [NET_RX]
[445440.445366] Softirq: CPU:  21   Pid:  748188 [allreduce]      Duration:      727398   SIRQ:         3 [NET_RX]
[445479.497348] Softirq: CPU:  82   Pid:  748249 [allreduce]      Duration:      429065   SIRQ:         9 [RCU]
All traced processes have exited
```

Figure 2: Example `detect-detour` output

```
# 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
```

Figure 3: `detect-detour` output when saving trace output showing `khugepaged`

- **Timestamp***:* Time, since boot, of the start of the detour in [seconds.microseconds] format
- **Detour type***:* Irq, Sched, or Softirq
- **CPU***:* CPU on which the detour occurred
- **Pid***:* ID of process that was being monitored and was interrupted
- **Process Name:** Name of the process that was being monitored and was interrupted
- **Duration***:* The duration, in nanoseconds, of the detour
- **Details**: Additional details related to the detour type:
  - **Irq**: the hardware IRQ number and handler name of the IRQ that interrupted the monitored process
  - **Sched**: the ID and name of the new process that interrupted the process being monitored
  - **Softirq**: the software IRQ number and handler name of the software interrupt that interrupted the monitored process

Figure 2 is an example of `detect-detour` output. In this instance, processes named allreduce were continuously monitored for detours longer than 300 microseconds. Four detours were detected. Each detour occurred on a different CPU and affected a different allreduce process. All the detours were caused by software IRQs, however a variety of software IRQ types were reported (TIMER, NET_RX, RCU).

3.3 Kernel Traces

`detect-detour` can also be used in conjunction with kernel tracing capabilities provided in the Linux kernel to get more detail about the source of the OS noise reported by `detect-detour`. A variety of kernel tracing capabilities can be used. For example, the `function_graph` tracer displays a graph of kernel functions executed with corresponding information such as timestamps and function durations. Linux also provides a variety of event tracers [7] which can write event-specific data into the trace buffer as event operations execute. Some of the event tracers we have experimented with to better understand OS noise include timer, irq, and rcu.

Figure 3 is another example of `detect-detour` output, this time showing the trace buffer being captured to disk. In this instance, processes named allreduce were continuously monitored for detours longer than one millisecond. Each time a detour was detected, the trace buffer was copied to a file in `/tmp/trace_data/`. One detour was due to an allreduce process being interrupted by a `khugepaged` process and the remaining detours were due to software IRQs.

Figure 4 shows `function_graph` tracer output that was automatically saved to the `/tmp/trace_data/cpu70-20250408-121042958018` file when the detour caused by `khugepaged` was detected. The timestamp provided by `detect-detour` was used to find the corresponding events of interest in the `function_graph` output. It shows the `khugepaged` process (pid 1774) being scheduled on CPU 70 which had previously been running an allreduce process (pid 170709). Figure 4 only shows a small amount of `function_graph` tracer output before and after this occurred, but it provides an example of how kernel tracing data can be used in conjunction with `detect-detour` to better understand what caused a detour, what the detour event is doing, and corresponding data such as function durations.

```
# 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();
```

Figure 4: `function_graph` tracer output showing noise from `khugepaged`

3.4 Capabilities

`detect-detour` provides arguments that allow the user to control its behavior. The capabilities provided are:

- **One-shot monitoring:** `detect-detour` exits as soon as a detour is detected for the processes being monitored. This is the default behavior.
- **Continuous monitoring:** `detect-detour` continuously monitors for detours until it is interrupted by the user, until the timeout is reached, or until all the monitored processes exit. This is the behavior when the `--cont` argument is specified.
- **Timed monitoring:** `detect-detour` continuously monitors for detours for a specified period. This is the behavior when the `--timeout` argument is specified.
- **Kernel trace data preservation:** `detect-detour` copies kernel trace buffer content to a user-specified path each time a detour is detected. This preserves the data for future examination and is especially useful when continuously monitoring for detours. This is the behavior when the `--dir` argument is specified.
- **Tracing control:** `detect-detour` automatically enables kernel tracing when it starts and automatically disables kernel tracing when it exits. This is the behavior when the `–-trace` argument is specified.
- **Process selection:** The processes monitored can be specified using process IDs, parent process IDs, process names, or parent process names. When parent IDs or parent names are used, all existing processes which are descendants of the parent process are monitored. This is controlled by the `--pids`, `--ppids`, `--pnames`, and `--ppnames` arguments.
- **Involuntary detour detection:** `detect-detour` only reports process scheduling detours if the monitored process was context switched by another process as opposed to the monitored process voluntarily relinquishing the CPU. This is the behavior when the `--invol` argument is specified.
- **Detour threshold:** This defines the time threshold, in nanoseconds, that must be exceeded for a detour to be reported. This is controlled by the `--threshold` argument.

3.5 Using `detect-detour` to investigate OS noise

The goal of an investigation into OS noise is to improve the performance of one or more applications. The process may involve several iterations, where each iteration consists of measuring application performance as a baseline, identifying one or more OS noise sources, mitigating the identified noise sources, then measuring performance to determine the improvement. The iterations stop once the performance is acceptable, or the cost of additional improvements outweighs the benefit. The following discussion will focus on the second step, where `detect-detour` is used.

Before using `detect-detour`, pick the application to run and monitor. This could be an actual application, a proxy application, or a benchmark. Since `detect-detour` also causes noise, we recommend binding the application processes to specific CPUs and using the `taskset` utility to run `detect-detour` on a separate CPU to minimize interference with the application.

Next, determine the noise threshold to use and how long to monitor for detours. Data from `sysjitter`, FWQ, or P-SNAP can be helpful when choosing a threshold, while data from FTQ can be helpful when choosing the monitoring duration. To use `detect-detour` to help choose the threshold and monitoring duration, we recommend experimenting with different thresholds and run durations in continuous monitoring mode to get an idea of the detour durations.

```
# Short run with low threshold
detect-detour --pids <pids> --threshold 100000 \
 --cont --timeout 15

# Long run with higher threshold
detect-detour --pids <pids> --threshold 1000000 \
 --cont --timeout 360
```

Figure 5: `detect-detour` command examples

Figure 5 shows two example `detect-detour` commands to use when figuring out the threshold and timeout values to use. Start with the first command, a short run, to get an idea of the range of detour durations to refine the threshold value. In our experience, the detours to examine first are greater than 100 microseconds. The second command, a long run, is an example that may be used once a reasonable threshold and timeout have been chosen. It uses a higher threshold, one millisecond, and a longer timeout, six minutes, to find periodic detours that may only recur every three minutes or less.

If the previous steps do not reveal enough information about the OS noise, then it may be helpful to use the kernel ftrace feature in conjunction with `detect-detour`. For instance, if a kernel kworker thread is causing long duration detours over five milliseconds, using the ftrace `function_graph` tracer could shed light on exactly what that kworker thread is doing. In this case, since the detour duration has already been identified, that should inform the threshold value to choose to minimize the number of detours reported. Configure the ftrace `function_graph` tracer first, then run `detect-detour` with the chosen threshold. If it is certain the first detour will be the one to capture, using single-shot mode, an example of which is shown in the first command in Figure 6, may be appropriate. If the desired detour may not be the first, then using continuous mode with a timeout while capturing trace snapshots, as shown in the second command in Figure 6, or manually interrupting `detect-detour` is more appropriate.

```
# Single shot, capture trace of first detour only
detect-detour --pids <pids> --threshold 5000000

# Continuous, capture traces of detours until timeout
detect-detour --pids <pids> --cont --threshold 5000000 \
 --timeout 360 --dir /tmp/noise
```

Figure 6: `detect-detour` command examples with tracing

4 Development History

For this paper we used and described the capabilities of the latest, as yet unreleased, version of `detect-detour`. It has many capabilities that have been added over time as we have used and continued to evolve the tool.

Our initial goal for `detect-detour` was to provide a tool that could automatically stop the kernel tracing when any one of a set of specified processes experience a detour exceeding a configurable threshold, making it easier to use the kernel tracers to catch the sources of noise. We released this initial version as part of HPE Cray Supercomputing User Services Software (USS) 1.0.

We soon realized that `detect-detour` could be used by itself to identify noise sources. For USS 1.1, we added an option to keep monitoring for detours beyond the first one, an option to set a time limit for monitoring, an option to save the kernel trace buffer to a file for each detected detour, an option to show only involuntary scheduling events, and an option to trace the children of one or more processes.

It became obvious during work on a customer system, that it was hard to identify a process, IRQ, or software IRQ by just their number, so in USS 1.2 we enhanced the output to include process names, IRQ handler names, and software IRQ names.

In the USS 1.3 release, `detect-detour` gained the ability to automatically exit when all the monitored processes have exited and to accept names for the processes to monitor.

To make it easier to correlate kernel traces to the detours identified by `detect-detour`, we added timestamps to the detour output in USS 1.3. Unfortunately, the kernel tracer and the eBPF code use different clock sources, so the timestamps do not always match.

The latest version of `detect-detour`, which we have used for this paper, will be part of the upcoming USS 1.4 release. This latest version now displays a count of the total processes being monitored, and it displays the name of the monitored process for each detour.

5 Next Steps

Currently, `detect-detour` must be run by a user with elevated permission since the eBPF code needs to be executed in kernel mode, inserting the hooks requires altering kernel memory, and managing the kernel tracing also modifies the kernel behavior. There are ways to reduce the level of permission needed, such as by allowing unprivileged BPF usage. More investigation is needed to determine if this is possible, practical, or even desirable.

The way involuntary process context switches are identified could potentially be improved. Currently it is based only on the priority comparison of the outgoing and incoming processes. If the incoming process has a higher priority than the current process, it is considered an involuntary context switch and is considered a source of noise. Otherwise, `detect-detour` does not consider the context switch to be a source of noise as the current process may be intentionally yielding the processor.

Since the timestamps displayed by `detect-detour` do not always align with the kernel tracing timestamps, we are exploring ways to make it easier to correlate the `detect-detour` output with the kernel tracing data. The kernel trace functionality provides several clock sources to use. We are investigating whether one of the non-default clocks better aligns with the clock available through eBPF. Another option being considered is to have the `detect-detour` kernel code inject its own data into the kernel trace buffer, which may make it easier to correlate the tracing data with the output from `detect-detour`.

In the current implementation, when `detect-detour` saves a snapshot of the kernel trace buffer, it copies the entire trace buffer for all CPUs to a file in the file system. The copy is slow and saves more data than is needed. There are two changes being explored to address these problems.

First, we are examining having `detect-detour` create trace buffer snapshots when a detour is detected, as opposed to copying the trace buffer to a file system. This is being evaluated because it can be costly to copy the trace data to a file system, affecting the timeliness and usefulness of the trace data captured. The Linux kernel tracing snapshot functionality is being utilized for this option.

The second change we are examining is to have `detect-detour` copy or snapshot only the per-CPU trace buffer for the CPU where the detour was detected as opposed to the global trace buffer used by all CPUs. In our experience, examining the global trace buffer often does not provide useful insight, and using the per-CPU buffer is more manageable and has less of an impact to the system.

6 Conclusion

While there are tools that make use of the eBPF kernel feature for various other purposes, using it for noise detection and identification appears to be unique. By taking advantage of eBPF and its access to kernel trace points, `detect-detour` is able to identify all OS noise sources. It also provides the advantage of being able to monitor a specific process or set of processes to identify only the noise events that are actually affecting an application of interest. `detect-detour` provides a simple, but powerful, interface that can be used on any Linux system and can be run across multiple nodes with ease to identify OS noise sources and collect meaningful noise data for all processes of an application.

REFERENCES

[1] eBPF Foundation. Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io/

[2] David Riddoch and Alexei Zakharov. 2023. sysjitter. https://github.com/alexeiz/sysjitter

[3] University of California and Los Alamos National Laboratory. 2006. Performance and Architecture Laboratory (PAL), System Noise Activity Program. In Trinity NERSC 8 RFP benchmarks. http://www.nersc.gov/assets/Trinity--NERSC-8-RFP/Benchmarks/June28/psnap-1.2June28.tar

[4] Department of Computer Science and Engineering, The Ohio State University. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, RoCE, and Slingshot: OSU Micro-Benchmarks 7.5. 2024. https://mvapich.cse.ohio-state.edu/benchmarks/

[5] The kernel development community. 2025. OSNOISE Tracer. https://docs.kernel.org/trace/osnoise-tracer.html

[6] The kernel development community. 2025. ftrace - Function Tracer. https://docs.kernel.org/trace/ftrace.html

[7] The kernel development community. 2025. Event Tracing. https://docs.kernel.org/trace/events.html

[8] Los Alamos National Laboratory. 1991. FWQ and FTQ. https://asc.llnl.gov/coral-benchmarks#ftq.

[9] IO Visor project, Linux Foundation. 2025. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.