Understanding noise introduced by operating systems and its effects on high performance computing

Mati Kärner
Department of Computer Science, University of Tartu
Tartu, Estonia
Email: mati.karner@ut.ee

***Abstract*—Jitter caused by Operating System (OS) is a well studied factor known to influence the performance of High Performance Computing (HPC) applications. In this study we give a short overview of previous work regarding OS noise. We implement benchmarking suite to quantify noise and present experimental results from four different real life architectures. We also attempt to reduce noise on fresh installation of general purpose OS with the intention to verify the implemented tool and to demonstrate the potentiality of tailoring general purpose OS's to HPC domain.**

Keywords*—Jitter, Operating system noise, High performance computing, Linux.

I. Introduction

OS introduced noise i.e. jitter is a collective overhead caused by hardware, user-space software, kernel daemons and several OS management related applications. Massively Parallel Platforms (MPP) are getting more popular - HPC is not merely an academic-only landscape but an essential tool driving innovation and development in various industries. At the same time HPC applications are getting more complex e.g. we are seeing growing interest in virtualization for achieving fault tolerance as well as platform independency. Some application require dynamic libraries and scripting environments that are often not present on the computing nodes. Two possibles solutions for meeting these demands are customizing general purpose OS's or extending existing light-weight and microkernels.

Multi-user and multiprocessing environments require a short latency time. In order to make sure that each task gets it's time slice among all other competing processes, an on-chip timer is programmed to interrupt the CPU for tasks scheduling purposes. Depending on the system, this might mean that a computation job running alone on a core is needlessly interrupted after every 10 to 1 ms. Moreover, it is common for HPC applications to alternate between computation and synchronization phases. If some of the nodes enter synchronization phase late, all the other nodes will have to wait and the whole computation process gets delayed. In this context, it becomes clear that providing users with the environment they are accustomed to, and at the same time achieving comparable results to light-weight or microkernels, is not a trivial task.

In order to gain deeper understanding on how noise in OS influences the HPC applications, we conducted a literature review on related studies to explore different methods for reducing and quantifying the noise as well as identifying individual noise sources. Using some of the techniques we implemented a benchmarking suite and verified it against four real word architectures. We also performed and customized a clean installation of a general purpose OS with a goal to explore the overall system noise level.

This paper is organized as follows: in Section II we will give a short overview of the previous work. In section III the description of the method is given. Section IV provides experimental results produced in the study. Section V will summarise the results.

II. Previous work

A significant effort has been dedicated to understand how OS introduced noise affects scientific computations. Part of this work also involves studying and developing techniques for quantifying and reducing the noise. The purpose of this section is to give brief overview on some of this work.

Morarie et al. developed a quantitative descriptive method for describing individual noise events in [1]. The method is based on augmenting LTTng (Linux Trace Toolkit Next Generation) to produce a trace consisting of kernel entry and exit points. By analysing the trace they identified individual noise sources. In addition, they compare their results with those of obtained with Finite Time Quantum (FTQ) benchmark proposed by Sottile and Minnich in [2]. FTQ measures a maximum number of basic operations that can be completed during some fixed time interval and number of operations that were actually completed. The difference of these two quantities is identified as noise introduced by OS. Although both methods seem to detect similar events, the results obtained by the former are much more specific, allowing us to differentiate events such as causes of interrupts. The findings show that majority of the noise is comprised of periodic timer interrupts and page faults. Moreover, different applications are likely to be differently affected by the jitter, mainly because of the nature of applications, e.g. some tasks are hard for OS to balance and therefore cause indirect overhead.

In [3] Akkan et al. measure and identify OS noise using Fixed Work Quantum (FWQ) benchmark and Ftrace[1] - a Linux kernel built-in tracing utility, available for several platforms.

[1] https://www.kernel.org/doc/Documentation/trace/ftrace.txt

Contrary to FTQ which counts individual operations, FWQ repeatedly performs a fixed amount of work and measures the time necessary to complete that work. They experiment with tickless Linux prototype, which means offloading OS tasks to dedicated core(s) allowing other cores to work in an uninterrupted manner. In this particular case, they pin cores 3-6 of a 4-socket 6-core AMD Opteron machine to a benchmarking application and leave other two for OS tasks and interrupts. To study cache related effects, they fill all but one L1 cache line available on a core with an array. This way, they are be able to detect when some of the data is evicted from the cache. Results from this experiment show that parallel applications running in tickless environment run faster due to reduced variability time in synchronization phase. On the other hand, since all network packets are processed only on dedicated OS cores, network bandwidth is reduced up to 10%. Disabling ticks also eliminated L1 cache misses, as there were nothing else but benchmarking application running on a single core. Similar study conducted by Petrini et al. in [4], also using FWQ-like technique, confirms the role of timer interrupts (LAPIC/PIC) in OS noise.

Using a theoretical probabilistic model Tsafrir et. al showed in [5] that the noise in parallel jobs grows linearly w.r.t. the number of nodes it occupies. But only if noise probability is small enough. Once the job exceeds a particular size, a detour is nearly certain to occur. This model assumes Bernoulli distribution, implying bimodality. The authors admit that this might not be the case in practice, since often there are different probabilities for different noise events. An empty loop is ran on all cores with computation phase calibrated to take certain amount of time. Individual run times of loops were saved for later processing. The collected dataset yielded an distribution used to interpolate detour probability. A low overhead tracing utility *klogger* was developed to study noise. As in [3], all the daemons usually not found in scientific OS were removed. Authors conclude that majority of fine-grained noise is caused by system interrupts. Context switches between kernel and user mode evict application data from cache causing cache misses. They propose a concept called smart timers which aims at reducing overhead by avoiding unnecessary periodic ticks. This is achieved by setting one-shot timers and doing all kernel accounting upon each kernel entry rather than periodically.

Kothari et al. developed a tool in [6] that identifies and quantifies OS jitter, leaving aside cache misses, TLB (Translation Lookaside Buffer) misses and page faults. By running a loop on a patched kernel with tracepoints, a timestamp was logged upon each entry of a loop. From this data, noise is identified by subtracting successive timestamps and comparing the resulting value against some threshold. Study claims that 63% of the total jitter comes from timer interrupts. Rest is the result of various system daemons and interrupts, most of which can be eliminated.

The effectiveness of co-scheduling solutions on 4096 CPU IBM Power6 multicore cluster is studied by Seelam et al. in [7]. Each node in the cluster runs an IBM AIX OS image. Authors developed a micro-benchmark that performs series of computations on every CPU followed by a global synchronization operation. More precisely, the method employs two nested loops: the inner one which is used for performing calculations and the outer one for collective synchronisation. The task in the inner loop is calibrated to run a fixed amount of time. Thus, any noise encountered during the computation will cause nodes enter the synchronization phase late. Study highlights several interesting findings. Firstly, not all cores experience similar noise. Primary CPU is likely to experience more noise due to additional responsibilities such as load balancing. Moreover, some Linux SMP (Symmetric Multi Processing) kernels tend to set default affinity for many interrupts to CPU 0 [8], thus further increasing noise. They observe that average tick processing times grows as the function of the number of CPUs. This finding is also confirmed in their later study [9], in which they observe that jitter not only varies between the different cores but also between the hardware threads in each core. Secondly, study claims that the number of CPU that contribute to the overall noise grows proportionally to the degree at which computation interval gets smaller. They also stress that from the perspective of scalability it is important to identify and mitigate jitter sources with high variance.

Ferreira et al. construct kernel-level noise injection framework [10] to characterize the impact of noise on HPC applications in Cray XT3/4 series machine consisting of 13,000 nodes. Each node contains 2.4 GHz dual-core AMD Opteron processor and Cray SeaStar network interface, which is connected to the CPU via a HyperTransport link. The framework allows to specify noise in terms of duration and frequency. Variety of noise patterns are studied on different applications running on Catamount lightweight operating system. They show that show that 1000 Hz 25μs noise interference can cause a 30% slowdown in application performance on ten thousand nodes. In fact, noise with this frequency can have up to 5% of slowdown on application considered relatively insensitive to noise. They also observe, that HPC application can often absorb substantial amounts of high-frequency noise, but tend to be significantly affected by low frequency noise.

III. METHOD

We implemented micro-benchmark utility *bench* to quantify and identify noise events. Using Selfish detour method proposed by Beckman et al. in [11] and tracing utility Ftrace, we collect and identify individual noise sources. Compared to the other Linux tracers such as *ktrace*, LTTng etc., Ftrace offers an extensive feature set and provides a convenient tree like output of system calls.

Platform	CPU	OS	*rdcts()* (μs)	*gettimeofday()* (μs)
Laptop	Intel(R) Core(TM) i5-3437U 2.4GHz	Ubuntu 13.10 (Linux 3.11.0-18)	0.0146	0.5629
KIA	AMD A10-5800K APU with Radeon(tm) HD Graphics 3.8GHz	Fedora 20 (Linux 3.13.6-200)	0.0111	0.4188
Vedur	AMD Opteron(TM) 6276 2.3GHz	SL6.5 (Linux 2.6.32-358.2.6)	0.0191	0.2839
Rocket	Intel(R) Xeon(R) E5-2660 v2 2.20GHz	SL6.5 (Linux 2.6.32-431.1.2)	0.0451	0.4087

TABLE I: Characteristics and benchmarks of different systems.

```
29378.940821 |   0)  bench-28781   |               |  do_IRQ() {
29378.940822 |   0)  bench-28781   |               |    irq_enter() {
29378.940822 |   0)  bench-28781   |   0.074 us    |      rcu_irq_enter();
29378.940823 |   0)  bench-28781   |   0.052 us    |      vtime_account_irq_enter();
29378.940825 |   0)  bench-28781   |   1.002 us    |    }
```

Fig. 1: `function_graph` output displaying timestamp, CPU, process, PID, duration and function name respectively.

A. Ftrace

Ftrace is a generic tracing framework instrumenting Linux kernel. Among all available tracers in Ftrace, the one concerning this paper is `function_graph` tracer, which probes a function on its entry and its exit. This is done by using a dynamically allocated stack of return addresses. On function entry the tracer overwrites the return address of each function traced to set a custom probe, while saving the original return address. When the instrumented routine finishes it returns to Ftrace, which calls the function graph tracer with the function exit tracepoint data. After that the real return address is popped from the stack and control is handed over to the real caller.

Ftrace utilizes implicit instrumentation, i.e. implicit tracepoints are placed into the code automatically by the compiler [12]. If a flag indicating compile-time support for Ftrace is set, the compiler adds code calling assembly routine `mcount` to the prologue of each function. Some systems that support dynamic tracepoints are tailored with Ftrace enabled by default, because operating system is able to replace trace points with `nop` instructions whenever the tracer is disabled, thus producing little or no overhead.

The trace is kept in preallocated ring-buffer, a circular data structure consisting of linked pages. Page refers to a 4096-byte structure, which is default (virtual memory) page size for x86 Linux kernels. Trace is read from the buffers and written to disc after the benchmarking process is completed. The buffer size is modifiable and restricted to the amount of available memory. A practical problem with `function_graph` is that trace fills up very quickly. Because processing large text files consumes too much time, increasing the buffer size is solution only up to a certain point (~ 64 MB). Also, other OS related concerns that might influence the benchmarking procedure arise, e.g. swapping.

B. Selfish detour

The idea behind selfish detour (SD) is running a loop that identifies and stores informations about noise events. Inside the loop a timestamp is recorded and compared to the one recorded during the last iteration. If the difference between the timestamps exceeds a certain threshold we conclude that we have been hit by a noise event, otherwise, we assume we've been working uninterruptedly. A pseudo code of the method is provided in Figure 2.

We are also interested in the minimum number of cycles needed to execute the loop. This way we can calculate the length of each detour by subtracting the minimum length of the loop from the difference of consecutive pairwise entries in the array.

The drawback of this method is that it doesn't take into account memory related noise effects. Only data used in the process is an array for collecting the results. Also, not much can be said about the noise sources. For the this purpose Ftrace is used.

```
current = rdcts()
while(count < maxiter):
  prev = current
  current = rdcts()
  td = current - prev

  if(td < threshold):
    results[count++] = prev
        results[count++] = current

  # Count min. ticks required to execute
  # this loop.
  if(td < minticks):
        minticks = td
```

Fig. 2: Pseudo code of selfish detour loop.

C. Bench

Bench is implemented as MPI application meant to run on all cores to get accurate results. As in [11], we have implemented the timestamp method using inline assembler code which retrieves the current value from CPU-s timestamp counter (TSC) register (see `rdcts()` in Figure 2). As a novelty we benchmark each CPU for minimum loop length and choose the minimum from all of the results as the global minimum used to calculate the noise durations. This is done to prevent factors introduced in [7], [9] from compromising the overall results. The global minimum also constitutes as maximum resolution of the benchmarking utility. On all of the systems we observed during this study, the minimum length of a loop never exceeded $0.01\mu s$.

As stated earlier, each CPU is benched separately. This means that careful attention has to be paid on CPU-s that become idle. More precisely, we don't want the processors exiting loop earlier to take up noise events. In addition we'd like to prevent all other perturbing processes such as load balancing from happening. Borrowing an idea from *Netgauge*[2] network benchmarking tool, we have placed a `MPI_Barrier` operation before the results are being collected. Most MPI implementations busy-wait on blocking operations to ensure

[2]http://htor.inf.ethz.ch/research/netgauge/

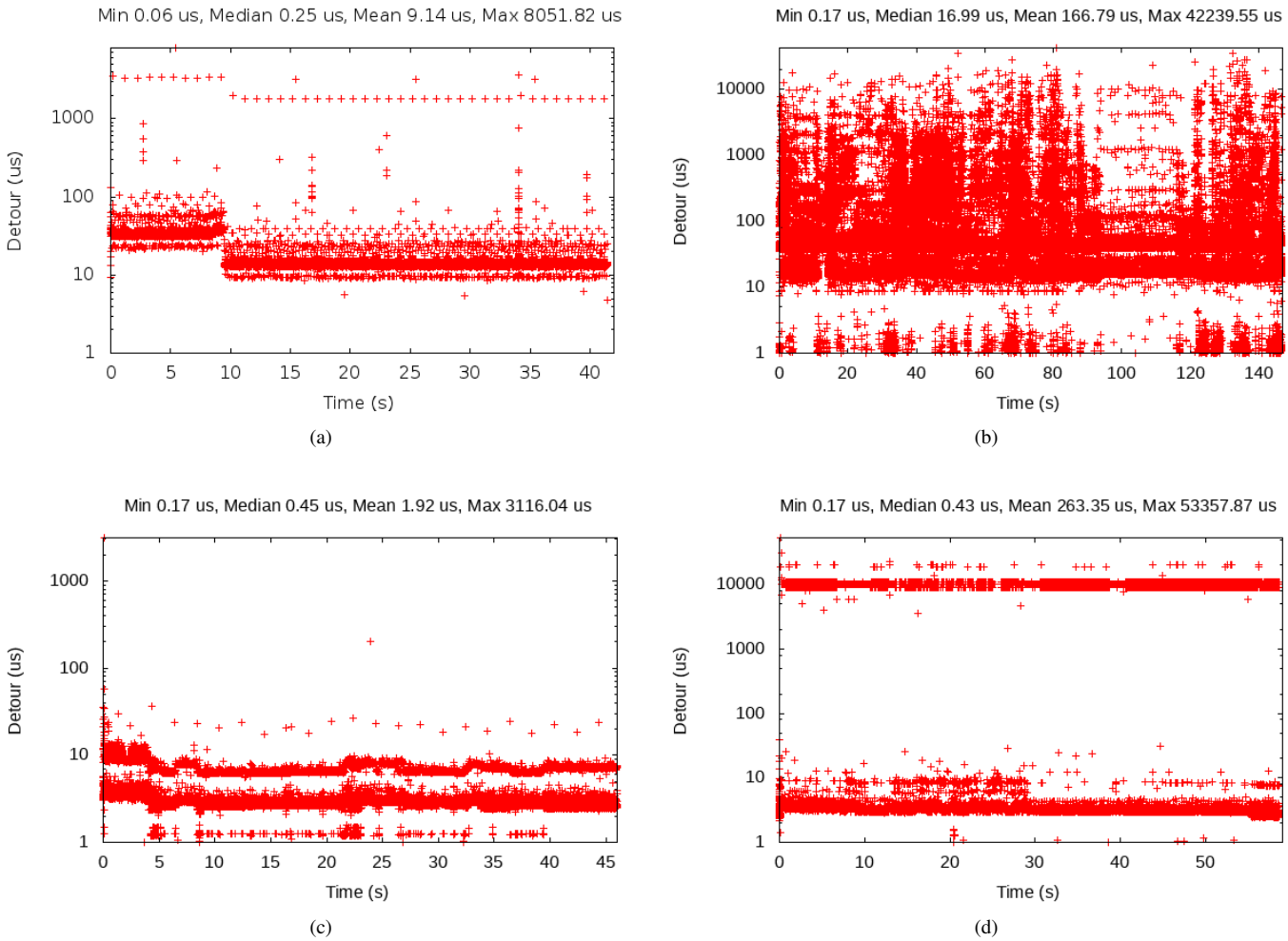

Fig. 3: Noise from KIA (a) and Laptop (b).
CPU-s with the same ID from two consecutive nodes of Vedur cluster, (c) and (d).

high availability of the processor. This means that CPU is utilized up to 100% even when no real computation are made.

As a next step, each process calculates the duration of benchmarking operation as well as the total sum of noise events. This data is then passed to master process which together with it's own results outputs the ratio of the summed noise and summed durations converted to percentage. This quantity constitutes the overall extra effort CPU makes relative to the total length of the calculation. Bench also produces *gnuplot* compatible plots displaying jitter per CPU, such as seen in Figure 3.

To quantify individual noise sources we use Ftrace `function_gaph` tracer output. Bench can automatically start and stop Ftrace tracing process, or not use that facility at all. This becomes necessary when we benchmark systems where Ftrace is either unavailable, incompatible or we don't have sufficient privileges. Sample of `function_gaph` output is provided in Figure 1.

Aligning the output from previous steps, i.e. start and end timestamps of detours, with the Ftrace log, we try to identify as much as noise sources as possible. We are not concerned about timer value differences across the CPU-s as the tracing is done on each CPU separately. Also, we believe that the overhead caused by running Ftrace simultaneously with benchmarking loop is subtle and doesn't depend on function being traced.

Due the peculiarities in Ftrace implementation, last tree digits of TSC value are not visible in the log, meaning that some of the events might appear in the trace log as they have started or ended at the same time, or both. Concerning the study, this is not a problem, because we use $1\mu s$ as a threshold (translates to 2900 cycles on a 2,9 GHz machine). According to measurements in [11], most architectures have typical detours ranging from one to several μs.

IV. Results

We tested the benchmark suite on four different real world architectures. An overview of the most important characteristics for each platform is provided in Table I. For two of the systems, namely Laptop and KIA, we had sufficient access privileges to run Ftrace on them. On Rocket and Vedur we

had to limit ourselves just to experimenting how increasing the number of cores/nodes affected the results.

A. Benchmarking the laptop and KIA

We ran Bench both on noisy laptop and KIA taking 5 samples on each. We used $1\mu s$ as a threshold and collected 10^5 noise events. These two environments were completely different in terms of noise. Laptop ran several user programs with extensive network traffic, while KIA had a post-installation environment with nothing extra except `sshd` daemon running. Both machines had frequency scaling turned off.

The average noise overhead measured on laptop was 9.1106% with std. dev 2.2%. On KIA, the same quantities were 2.0955% and 0.02% respectively. The noise can be observed on plots (a) and (b) in Figure 3, both collected from CPU 0. On y-axis we have the detour duration in μs (logscale) and on x-axis the time in seconds from the beginning to the end of the benchmarking process. In terms of environment, both plots depict what we excepted, except the sudden decrease of noise on KIA. This effect can be seen on all CPU-s through out all of the samples we took. Tools developed in this study are not able to provide information as for why this is happening, so one could only speculate.

Exploring the Ftrace output revealed that majority of the explorable noise is caused by local timer interrupts, 95.75% for laptop and 99.55% for KIA. Roughly four percent of other noise comprising events on laptop were hardware interrupts. Rest of the jitter was mainly caused by kernel threads such as `kworker`. Both machines also experienced page faults, 0.22% - 0.35%.

In another experiment, we tried to disable as many as redundant or non-critical daemons and services on KIA as possible. In this category are all graphic and multimedia related services that are not needed in HPC headless environment e.g. `gdm` (*Gnome Display Manager*), biometry daemons such as `fprintd`, connectivity services e.g. `bluetoothd` and other miscellaneous utilities. We did, however, preserve security related features such as firewall and logging services. We then rerun Bench and achieved 0.2925% decrease in noise, that is 1.803% measured in total CPU overhead with 0.01% sdt. dev. Despite the low gain, we are still optimistic. This was by no means the maximal attempt to clean up redundant services. One could achieve better results simply by filtering logging and audit targets, disabling `SELinux`, replacing wasteful daemons with more light-weight ones etc. We plan to improve these results in the future.

B. Cluster

Results on Vedur and Rocket clusters are worrisome. See Table II for Vedur. A general rule when running Bench on both clusters is that when benching only one node we get a surprising 32% for Vedur and 5% for Rocket (20 cores on each node), then a sudden drop after which the noise starts to grow. The detour trace are not free of anomalies either, both on Rocket and Vedur we can witness more than order of magnitude difference in noise levels for individual CPU-s on the same node. Figure 3 (c) and (d) show CPU-s with the same ID-s on different nodes. Notice the great variance between noise duration.

Nodes / CPU-s	CPU overhead (%)	Standard deviation (%)
1/32	22.8615	3.73625
2/64	13.4848	2.59186
3/96	14.2309	5.22443
6/192	19.3965	3.6288
10/320	22.9351	3.5549
20/640	27.4348	6.08552

TABLE II: Vedur benchmarks.

As stated earlier, we didn't have sufficient privileges to run Ftrace on these systems, nor were we able to control frequency scaling and other power management related settings. CPU throttling might in fact be the cause of these strange results. At the beginning of measurements each CPU is benchmarked to estimate the number of cycles per second. All other measurements are dependent on this estimate and are thus biased if the CPU frequency changes during the benchmarking process. Moreover, we have reason to believe that not all nodes on Vedur have similar clock rate.

V. Conclusion

Numerous works have explored the influence of OS jitter on different platforms. In this paper we implemented experimental benchmarking suite and showed that majority of noise is indeed caused by periodic timer interrupts and hardware IRQ-s, thus confirming the results of previous studies. Using the real world architecture, we have also showed that some of the noise can be eliminated by removing and disabling non-critical daemons.

The future work should concentrate on improving the method in controlled cluster environment. Results from two other architectures provide confidence that the method can identify and quantify different sources of noise, although more effort is needed to characterize detected jitter patterns as well as their effects on collective operations.

References

[1] A. Morari, R. Gioiosa, R. Wisniewski, F. Cazorla, and M. Valero, "A quantitative analysis of os noise," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 852–863.

[2] M. Sottile and R. Minnich, "Analysis of microbenchmarks for performance tuning of clusters," in *Cluster Computing, 2004 IEEE International Conference on*, Sept 2004, pp. 371–377.

[3] H. Akkan, M. Lang, and L. Liebrock, "Understanding and isolating the noise in the linux kernel," *International Journal of High Performance Computing Applications*, 2013. [Online]. Available: http://hpc.sagepub.com/content/early/2013/02/27/1094342013477892.abstract

[4] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "Analysis of system overhead on parallel computers," in *Signal Processing and Information Technology, 2004. Proceedings of the Fourth IEEE International Symposium on*, Dec 2004, pp. 387–390.

[5] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, os clock ticks, and fine-grained parallel applications," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05. New York, NY, USA: ACM, 2005, pp. 303–312. [Online]. Available: http://doi.acm.org/10.1145/1088149.1088190

[6] P. De, R. Kothari, and V. Mann, "Identifying sources of operating system jitter through fine-grained kernel instrumentation," in *Cluster Computing, 2007 IEEE International Conference on*, Sept 2007, pp. 331–340.

[7] S. Seelam, L. Fong, A. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea, "Extreme scale computing: Modeling the impact of system noise in multicore clustered systems," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–12.

[8] A. Sandler. (2008, Apr.) Smp affinity and proper interrupt handling in linux. [Online]. Available: http://www.alexonlinux.com/smp-affinity-and-proper-interrupt-handling-in-linux

[9] S. Seelam, L. Fong, J. Lewars, J. Divirgilio, B. Veale, and K. Gildea, "Characterization of system services and their performance impact in multi-core nodes," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 104–117.

[10] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 19:1–19:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413390

[11] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, no. 1, pp. 3–16, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10586-007-0047-2

[12] T. Bird, "Measuring function duration with ftrace," in *Proceedings of the Linux Symposium*, ser. LS '09, 2009, pp. 47–56. [Online]. Available: http://www.linuxsymposium.org/2009/ls-2009-proceedings.pdf

[13] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.12

[14] E. Vicente and R. Matias, "Exploratory study on the linux os jitter," in *Computing System Engineering (SBESC), 2012 Brazilian Symposium on*, Nov 2012, pp. 19–24.

[15] M. Desnoyers. (2010, Jul.) Generic ring buffer library. [Online]. Available: http://lwn.net/Articles/395778/