Identifying Sources of Operating System Jitter Through Fine-Grained Kernel Instrumentation*

Pradipta De, Ravi Kothari, Vijay Mann

IBM India Research Laboratory
New Delhi, India
{pradipta.de, rkothari, vijamann}@in.ibm.com

***Abstract*— Understanding the behavior and impact of various sources of Operating System Jitter (OS Jitter) is important not only for tuning a system for HPC applications, but also for the ongoing efforts to create light-weight versions of commercial operating systems such as Linux, that can be used on compute nodes of large scale HPC systems. In this paper, we present a tool that helps in identifying sources of OS Jitter in a commodity operating system such as Linux and measures the impact of OS Jitter through fine grained kernel instrumentation. Our methodology comprises of running a user-level micro-benchmark and measuring the latencies experienced by the benchmark. We then associate each latency to operating system daemons and interrupts using data obtained from kernel instrumentation. We present experimental results that help identify the biggest contributors to the total OS Jitter perceived by an application on a commodity operating system such as Linux. Our results revealed that while 63% of the total jitter comes from timer interrupts, the rest comes from various system daemons and interrupts, most of which can be easily eliminated. The tool presented in this paper can also be used to tune "out of the box" commodity operating systems as well as to detect new sources of operating system jitter that get introduced as software get installed and upgraded on a tuned system.**

Keywords: Operating system noise, jitter, interference, kernel instrumentation, kernel profiling

I. Introduction

Operating system interference, caused primarily due to scheduling of daemon processes, and handling of asynchronous events such as interrupts, constitutes "noise" or "jitter" (henceforth referred to as OS jitter) perceived by an application. Various studies have taken place in the recent past that demonstrate the debilitating effects of OS jitter on parallel applications running on large scale HPC systems [1] [2] [3] [4]. Traditionally, large scale HPC systems have avoided jitter by making use of a specialized light-weight operating system on compute nodes [5] [6]. However, this limits the use of such HPC systems as most applications, which are written for commercial operating systems can not be run on these systems. This has resulted in efforts to create light-weight versions of commodity operating systems such as Linux which can be used on compute nodes of large scale HPC systems [7] [8] [9].

*This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002

Creation of light-weight version of commodity operating system necessitates that a detailed study identifying the sources of OS jitter and a quantitative measurement of their impact on these operating systems be carried out. Most of the studies on OS jitter, so far, have concentrated on the effect of jitter on the scaling of parallel applications and have not really delved into the issue of identifying the biggest contributers to OS jitter. Apart from the well known ill effects of operating system clock ticks or timer interrupts [1], there is little data available about other system daemons and interrupts that contribute to OS jitter. Furthermore, tuning an out of the box commodity operating system is the first step towards mitigating the effects of OS jitter. In the absence of any quantitative information about the jitter caused by various system daemons and interrupts, system administrators resort to their established knowledge and other ad-hoc methods for tuning a system for HPC applications. This process not only requires highly knowledgeable system administrators, but is also error prone given the fact that new versions of these commodity operating systems get released at fairly regular intervals and new sources of OS jitter get introduced in these releases.

Identification of all possible sources of OS jitter and measurement of their impact on an application requires a detailed trace of the OS activity. Most of the existing general purpose OS profiling tools, such as OProfile [10] or the Linux kernel scheduler stats [11], provide a coarse measure in terms of time spent in each kernel function or process and do not uniquely measure the jitter perceived by an application due to each jitter source. Other benchmarks developed specifically for studying OS jitter such as the selfish detour benchmark [12] can be used to measure OS jitter on a wide range of platforms and study its effect on parallel program performance. However, they do not provide any information about what daemons and interrupts contribute to OS jitter and by how much.

In this paper, we present the design and implementation of a tool that helps in identifying sources of OS jitter on a commodity operating system such as Linux and can be used to quantitatively measure the jitter contributed by various system daemons and interrupts. The tool combines the techniques employed by micro-benchmarks used for studying OS jitter with the profiling techniques used by kernel profiling tools such as OProfile [10]. Our methodology comprises of

a running a user-level micro-benchmark and measuring the latencies experienced by the benchmark. We then associate each latency to operating system daemons and interrupts using data obtained from kernel instrumentation. The main contributions of this paper are:

- Design and implementation of a tool that can be used to identify all sources of OS jitter and measure their contribution to the overall jitter experienced by an application. This tool can also be used to identify scheduling patterns of sources of OS jitter. It can be a very useful tool in the context of cluster computing, as it can be deployed to detect sources of jitter that cause a node to behave anomalously once such anomalous nodes in a cluster have been identified.
- Experimental results that help identify the biggest contributers to the total OS jitter perceived by an application on a commodity operating system such as Linux. To the best of our knowledge, this is one of the first such comprehensive studies. Our results reveal that while 63% of the total jitter comes from timer interrupts, the rest comes from various system daemons and interrupts, most of which can be easily eliminated.
- Validation of the methodology through introduction of synthetic daemons and their reliable detection, which illustrates how the tool can be used to detect new sources of OS jitter that get introduced as software get installed and upgraded on a tuned system over a period of time.

The rest of this paper is organized as follows. In Section II we give an overview of our methodology. We give a detailed description of the implementation of our tool in Section III. We present experimental results of running the tool on Linux and a validation of our methodology in Section IV. This is followed by a discussion of related work in this area in Section V. Finally, we conclude in Section VI with a description of our ongoing work and our future research directions.

II. Methodology

In this section, we describe our overall methodology: the requirements for our tool and its design and implementation.

A. Requirements

OS jitter experienced by an application arises out of scheduling of daemon processes, and handling of asynchronous events such as interrupts. An interruption experienced by an application can be due to any of the following:

(1) a particular daemon or an interrupt occurs when the application is running and has a unique duration every time it occurs; thereby causing an interruption of unique duration to the application every time;
(2) a combination of daemons and interrupts occur in succession when the application is running, causing an interruption to the application that is equal to the sum of their individual execution times, every time that combination occurs;
(3) two or more types of daemons or interrupts have equal durations and occur at various times (not in succession) when the application is running, causing interruptions of equal duration to the application;
(4) two or more interrupts occur in a nested fashion, when the application is running and the total interruption experienced by the application is the sum of duration of all the interrupts (the top level interrupt includes the time taken by all its nested interrupts)

Our goal is to uniquely identify all the above in order to account for all the interruptions experienced by an application. It is also important to identify the daemons or interrupts that occur frequently in succession (case 2 above), as they would potentially lead to a significant interruption to the application when they occur in succession, even though individually they have a short duration. Case 3 can make it difficult to distinguish one source of interruption from the other as they have equal durations. This will be especially true if the kernel activity trace is not fine grained. Any technique that relies just on frequency domain methods will fail to identify such patterns and will be unable to distinguish between these cases (viz. case 2 and case 3 above).

B. Design

Our overall methodology consists of the following steps:

(1) instrument the kernel to record the start and end times of all processes and interrupts;
(2) make the kernel data structures, that record start and end times, visible to user-level applications;
(3) run a user-level application that:
    a) reads the CPU timestamp register in a tight loop (the critical section);
    b) calculates the difference between successive readings (timestamp deltas); and if the difference is greater than some threshold, it adds the timestamp delta to a histogram (henceforth referred to as the user-level histogram);
    c) reads the kernel data structures to find out what processes and interrupts occurred (and for how long) during the execution of the critical section and prints this timeseries data (henceforth referred to as scheduler and interrupt trace) along with the user-level histogram to files;
(4) analyze the user-level histogram and the scheduler and interrupt trace data, and trace the source of all the interruptions observed by the user-level application to a particular process or an interrupt or a combination of processes and interrupts.

Our tool, that works on the above methodology consists of the following four components:

(1) a kernel patch that executes the step 1 above;
(2) a character device and its driver (implemented as a kernel module) that executes the step 2 above;
(3) a user-level micro-benchmark that executes the step 3 above; and
(4) a data analyzer program that executes the step 4 above.

III. Implementation Details

In this section, we give the implementation details about each of the components outlined in the previous section.

A. Kernel Patch

We instrument the kernel (the schedule function and the do_IRQ interrupt handling function) to record the time stamps for the start time and end time of each process and interrupt, along with their names. The kernel records these timestamps in an internal data structure and there is an array of such data structures of a fixed length for both processes as well as interrupts. There is also a pointer to the current valid index in the array, where the next entry gets recorded. We have packaged the kernel instrumentation changes as a patch which is currently available for kernel versions 2.6.17.7 and 2.6.20.7 (Intel and PowerPC).

B. Device Driver Kernel Module

Once, the kernel is instrumented and starts recording the scheduler and interrupt handling data, we need to access these kernel data structures from the user-level application in a manner that has minimal overhead. There are various ways of achieving this. We can expose kernel data through the proc file system or we can make use of a device driver whose memory can be mapped to the kernel data structures. We make use of the latter approach. We create a character device and implement the device driver for that character device as a kernel module. The device driver maps the device memory (which is nothing but the kernel data structures) to user-level in its mmap function call. Any user application can now just open the device file and call mmap on it, like any normal file. The resulting memory pointer now maps to the kernel data structures.

C. User-level Micro-benchmark

The user-level micro-benchmark is based on the fixed work quantum principle, and enhances other similar benchmarks used for studying OS Jitter (such as the selfish detour benchmark [2] [12]). The pseudo code for the user-level benchmark is given in Algorithm 1. The benchmark executes several rounds, where each round consists of the following steps:

Step 1: The current valid index for scheduler and interrupt trace arrays in the kernel is recored using the memory mapped pointers to the already open device files.

Step 2: The CPU timestamp register is then read (using the *rdtsc* instruction on Intel) in a tight loop (the critical section - first *for* loop in Algorithm 1), recording the observations. Each loop consists of a configurable number of iterations (N). By default, we run it for 16 MB iterations. Higher the value of N, the larger the number of samples that can be collected, and higher are the chances of incurring cache misses, TLB misses and page faults.

Step 3: The current valid index for interrupt and scheduler arrays is read again. The contents of the two arrays in the kernel between the two readings of current valid index are then read. This consists of the names, and start and end times of all the processes and interrupt handlers that got scheduled during the execution of the loop. This information is written to a scheduler trace file and an interrupt trace file.

Step 4: The timestamp data generated in the loop is then processed, and the difference between successive readings is calculated. These deltas represent the number of cycles required to read the timestamp register. If the difference is greater than a threshold (we currently set it to be 10 times the minimum difference observed) due to a process getting scheduled, or an interrupt being handled or any other system activity that takes the CPU away from the application, the timestamp delta is added to a histogram (referred to as the user-level histogram). Most of these deltas (around 99% of them) would be very small and these correspond to the actual number of cycles required for the *rdtsc* instruction (it is roughly equal to 88 cycles on Intel Xeon - 0.03 microseconds on a 2.8 GHz machine). However, when a daemon process is scheduled or an interrupt is handled or combinations of these two occur, the deltas are much higher. We improve upon the selfish detour benchmark [2] [12] by ensuring that the only instruction executed in the critical section is the *rdtsc* instruction (and the additional instructions that are part of the execution of the *for*loop) and all processing is done outside the critical section. This reduces the variance in timestamp deltas and helps us detect even the smallest of interruptions accurately. We generate the user-level histogram using a bucketing technique that makes use of variable width bucket and inter-bucket distances. It accurately captures the structure of the timeseries data by using additional buckets for regions where there are more sample points and less number of buckets for regions where there are a few sample points. This finishes one round.

*Step 5:*The benchmark continues to execute for a user specified period of time, executing these rounds and adding the timestamp deltas to the user-level histogram created in the first round. With this technique of splitting the execution in rounds, it is possible to run the benchmark for really long durations and collect a large number of samples without being limited by memory size for storing samples. When the given time to run elapses, the user-level histogram data for all rounds is written to a file. Approximately 3/4th of the time in a round is spent in writing all the data to file system and in generating histograms from timeseries data. The actual sampling (from the timestamp register) takes place only for the remaining 1/4th of the total time of execution.

D. Data Analyzer Program

The user-level micro-benchmark, upon completion, produces the following 3 output files:

(1) a distribution file containing the user-level histogram;
(2) a time series file containing scheduler trace data (process start and end times along with process names);
(3) a time series file containing interrupt trace data (interrupt start and end times along with interrupt names);

A data analyzer program reads the above 3 files, and generates a merged trace file that includes all those processes and

Algorithm 1 User-level Micro-benchmark

```
sptr = mmap(scheduler_device_file_ptr);
iptr = mmap (interrupt_device_file_ptr);
t1 = gettimeoftheday();
round=0;
while (elapsed_time < period) do
  round++;
  /* start of kernel-level tracing */
  start_scheduler_index = sptr→current_index;
  start_interrupt_index = iptr→current_index;
  /* critical section - reading the timestamp register*/
  for (i = 0 to N) do
    ts[i] = rdtsc();
  end for
  /* end of critical section */
  end_scheduler_index = sptr→current_index;
  end_interrupt_index = iptr→current_index;
  /* end of kernel-level tracing */
  /* calculation of timestamp deltas */
  for i = 0 to N-1 do
    ts[i] = (ts[i+1]-ts[i]);
  end for
  /* collecting scheduler and interrupt trace data*/
  for start_scheduler_index : end_scheduler_index do
    print_to_file(start_time, end_time, process_name);
  end for
  for start_interrupt_index : end_interrupt_index do
    print_to_file(start_time, end_time, interrupt_name);
  end for
  /* generating user-level histogram from timestamp
  deltas*/
  if (round==1) then
    create_distribution (ts);
  else
    add_to_distribution (ts);
  end if
  t2 = gettimeoftheday();
  elapsed_time = t2 - t1;
end while
```

interrupts from files (2) and (3) above, that caused the user-level micro-benchmark to experience a delay. These include:

- all the processes that got scheduled between any two occurrences of the benchmark in the scheduler trace data; and
- all the interrupts that were handled when the benchmark was running (which is inferred from the scheduler trace data).

While generating the merged trace file, the analyzer program takes care of all the different cases identified in the requirements section (II-A) above.

The merged trace file is then used to generate a master histogram which has the same buckets (in terms of the bucket locations) as the user-level histogram (in file 1 above), but the samples are taken from the merged trace file. While we generate this master histogram, we also record various statistics for each bucket (such as the contribution of each process or interrupt or their combination to that bucket). This new master histogram should ideally match the user-level histogram, if all the interruptions experienced by the user level benchmark come from either a different process getting scheduled (a context switch) or an interrupt being handled. In practice, it will not fully match as we don't collect trace data about cache misses, TLB misses and page faults. We make use of the Parzen window method for density estimation (described below) to compare these histograms.

The analyzer program can also generate a histogram for each unique jitter source or just for each of the top 10 contributors to the overall jitter experienced by the user-level micro-benchmark. The analyzer program can operate in two modes: (a) a time domain mode , where it retains certain timing information (which can be used to infer scheduling patterns) such as what interrupts and processes get scheduled in succession, and considers each combination as a unique jitter source and maintains statistics about each of them or (b) a frequency domain mode where it doesn't retain any timing information and maintains statistics only about the individual daemons and interrupts (and not their combinations). The master histogram (that closely matches the user-level histogram) is generated in the time domain mode.

The master histogram of a system with any configuration can be compared to a baseline master histogram (*i.e.* a master histogram generated from the same system after it was optimally tuned) to detect any new sources of OS jitter that may have been introduced as software get installed and upgraded, after the initial tuning. We make use of the Parzen window method of kernel density estimation to plot and compare master histograms corresponding to various configurations of the same system.

Parzen window method for comparing histograms: Parzen-window density estimation [13] is a non-parametric estimation technique which uses the superposition of kernel functions $K(x_i, \sigma_n)$ placed at each data point x_i to estimate the density $P(x)$ in d-dimensions. Succinctly,

$$\hat{P}_n(x) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{h_n^d}K(\frac{x-x_i}{h_n}) \quad (1)$$

where, n is the total number of data points. The kernel function (a commonly chosen kernel is the Gaussian PDF) satisfies the property that,

$$\int_{\mathcal{R}^d} K(\cdot) = 1 \quad (2)$$

and $h_n > 0$ is the so-called bandwidth parameter. With increasing (decreasing) h_n, one tends to get a smoother (less smooth) estimate and this needs to be chosen with care [14]. It can be shown that the Parzen estimator is a consistent estimator in the sense that $\hat{P}_n(x)$ converges to $P(x)$ as $n \rightarrow \infty$.

Within the context of OS jitter, one may interpret the interruptions experienced by an application (the user-level

micro-benchmark in our case) as data points in 1-dimension (time). A Parzen-Window based density estimate then provides a robust representation of these interruptions and can be used to compare master histograms and user-level histograms as probability distributions. A Parzen window based density estimation does not suffer from aliasing effects resulting from discretization inherent in histogram based density estimation (due to choice of number of bins, bin width, and equal versus variable bin width). Discontinuities in the histogram based density estimation are an artifact of the chosen bin locations and they make it very difficult to grasp the structure of the data.

IV. Experimental Results

We conducted experiments to *(1)* identify various sources of OS Jitter and measure their contribution to the overall jitter experienced by an application for Linux run level 3; and *(2)* validate our methodology by introducing synthetic daemon processes and being able to clearly identify them by comparing their master distributions

Our experiments were conducted on a machine with an Intel(R) Xeon(TM) 2.80GHz CPU that had a cache size of 512 KB and I GB RAM. It ran Fedora Core 5 with kernel version 2.6.17.7. The timer interrupt interval was configured to be 10 milliseconds (*i.e. 100 Hz*).

A. Experiment 1: Identifying various sources of OS Jitter and measuring their impact

In this experiment, we go through the steps described in the methodology section and try to identify all sources of OS Jitter. The results of this experiment can be used as a guideline for tuning an out of the box Linux system.

We run the user-level micro-benchmark for an hour with a value of $N = 16MB$ at run level 3. It executes approximately 1800 rounds and the actual sampling of the timestamp register takes place for 15 minutes. The master histogram generated from the scheduler and interrupt trace files is compared with the user-level histogram generated by the user-level micro-benchmark.

The master histogram and the user-level histograms, plotted as probability distributions using Parzen window method for density estimation, are shown in Figure 1. The y-axis is a logarithmic function of the number of samples in a bucket (the frequency) and the x-axis is the interruption in microseconds. The following observations can be made:

Observation 1: The master distribution and the user-level distribution shows mismatch at certain portions. The left most points (less than $1\mu s$) in the user-level distribution most likely arises due to cache misses, TLB misses and demand paging requests. These points are absent in the master distribution as it consists of only interruptions caused due to processes and interrupts. The extreme right hand side of the user-level distribution also has two peaks (around $4000\mu s$ and the last one at $17000\mu s$) which we are currently unable to account for.

Observation 2: The master distribution appears shifted to the left as compared to the user-level distribution in certain

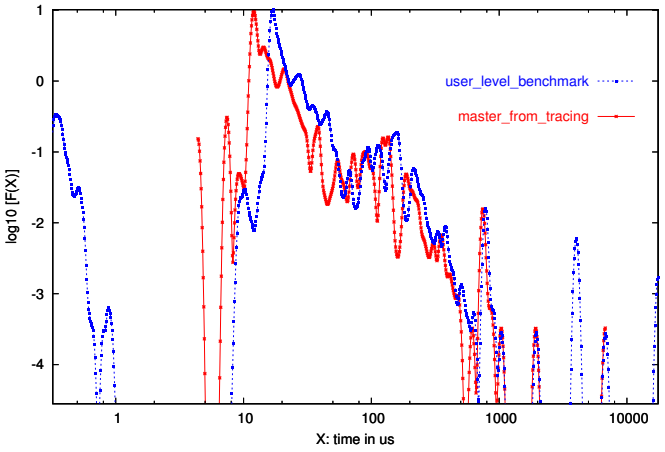

Fig. 1. Comparing user-level timestamp distribution and the master distribution generated from kernel trace data.

areas. This happens due to the following reason. The samples in the master distribution account for only the actual time taken for an interrupt to be handled or a process to run. It doesn't account for any overheads encountered during interrupt handling or context switching. Therefore the values in the master distribution are always lesser than the values in the user-level distribution by some cycles (as the user-level distribution consists of end latencies perceived by the user-level benchmark). This shows up in the graph as a left shift in the master distribution. This shift is more visible in the left side of the graph and becomes less prominent as we move towards the right (as the overhead cycles are now a very small percentage of the actual time taken by an interrupt or a daemon).

The data analyzer program is run in the frequency domain mode to generate statistics about all individual processes and interrupts that contribute to jitter perceived by the user level benchmark. These details are given in table I which is sorted in the decreasing order of total jitter percentage. The biggest offenders are at the top of the table. The timer interrupt contributes to nearly 63% of all the jitter experienced by the application. Most of the top offenders fall under the category of frequently occurring small to medium interruptions. The second biggest offender is the *hidd* daemon (the Bluetooth Hardware Interface Device Daemon) that starts up the bluetooth service. This is unlikely to be used by most HPC applications and can be easily removed as it contributes to nearly 9% of the total jitter. Most of the jitter sources that cause infrequent but long interruptions (like *hald*- hardware abstraction layer daemon, *crond*, *atd* and *runparts* daemons - all used for scheduling jobs, *cupsd* - common unix printing system daemon) occur in the bottom half of the table and can also be easily eliminated. Jitter caused due to kernel threads such as the journaling file system thread (*kjournald*), the block device kernel thread (*kblockd*) and the kernel default worker queue thread (*events0*) on the other hand is harder to eliminate.

Noise Source	Lowest Interruption(us)	Highest Interruption (us)	frequency	mean (us)	total jitter(us)	std dev(us)	total jitter(%)
timer	9.74	1042.05	76997	14.7	1131522.36	828.1	63.27
hidd	4.52	364.31	3404	49.17	167375.21	3450.97	9.35
python	4.52	220.17	1337	106.58	142494.64	5985.53	7.96
haldaddonstor	4.52	364.31	1522	61.03	92892.55	3708.57	5.19
ide1	10.92	64.74	3364	21.57	72569.8	1256.24	4.05
events0	4.65	101.33	433	81.18	35152.02	4452.23	1.96
eth0	7.12	80.35	1122	24.17	27115.4	1584.66	1.51
automount	4.98	173.85	156	162.72	25383.71	8703.52	1.41
sendmail	5.19	364.31	159	146.69	23323.57	8714.47	1.30
pdflush	4.93	220.17	161	100.7	16213.35	6389.97	0.90
idmapd	6.27	364.31	147	71.07	10446.99	5064.51	0.58
init	5.67	160.75	156	56.65	8836.82	3210.1	0.49
kblockd0	5.83	220.17	82	99.11	8127.18	6298.97	0.45
kjournald	0.92	154.15	181	39.61	7169.62	3531.18	0.40
kedac	4.7	110.16	705	9.71	6843.7	787.98	0.38
hald	348.69	363.31	13	353.48	4595.28	19435.05	0.25
watchdog0	0.92	7.67	735	5.47	4022.33	291.58	0.22
crond	6.01	164.43	18	91.21	1641.8	5460.71	0.09
syslogd	48.1	58.32	24	51.36	1232.68	2773.01	0.06
cupsd	117.92	349.48	2	236.06	472.12	19563.9	0.02
atd	130.07	134.3	3	132.09	396.28	8545.5	0.02
smartd	57.44	81.97	3	73.08	219.25	4781.35	0.01
runparts	160.76	164.43	1	164.24	164.24	0	0.01
xfs	36.34	36.86	1	36.49	36.49	0	0.002

TABLE I
SOURCES OF OS JITTER ON LINUX (FEDORA CORE 5) RUNNING IN RUN LEVEL 3

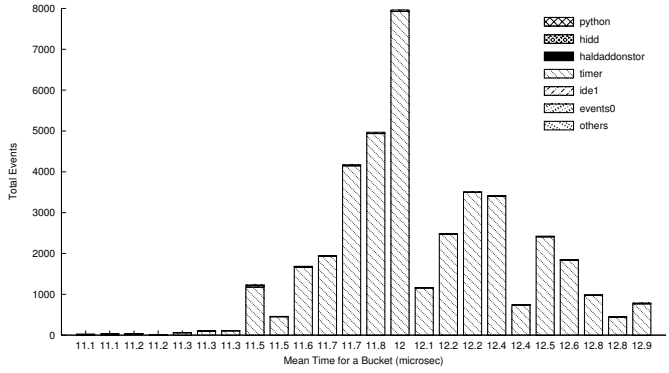

Fig. 2. Histogram buckets from the master distribution in the $11 - 13\mu s$ range.

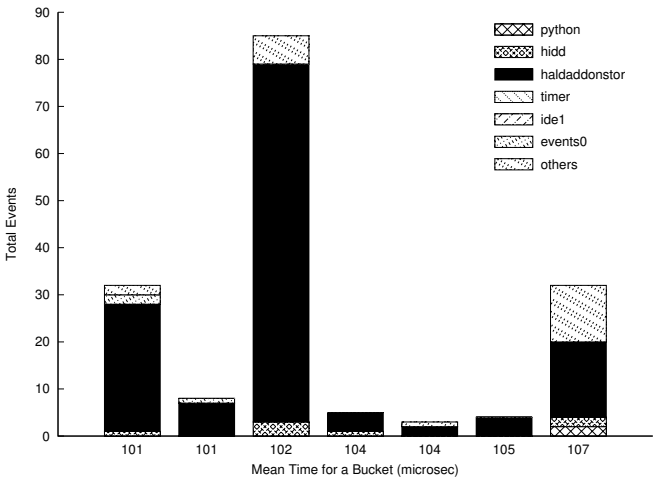

Fig. 3. Histogram buckets from the master distribution in the $100 - 110\mu s$ range.

In order to find out what daemons and interrupts contribute to a particular interruption peak in the master distribution graph, we can zoom into any of the peaks and look at the contents of the underlying histogram buckets. The histogram buckets corresponding to the 11-13 microsecond peak are shown in Figure 2. We can observe that in this range, the timer interrupt causes nearly 100% of the jitter. On the other hand, the buckets in the 100-110 microsecond peak (figure 3) illustrate that the haldaddonstor daemon contributes to most of the jitter in this range.

B. Experiment 2: Validation of the methodology through introduction of synthetic daemons

The goal of this experiment is to verify that our methodology works and it can be used to detect any new source of OS jitter (that get introduced as software get upgraded, or added) even after initial tuning of the system has been done.

In this experiment we introduce synthetic daemons that exhibit the following different characteristics:

(1) a single daemon process, that has a periodicity of 10 seconds, and executes a loop that that takes approximately 2500 microseconds to finish before it goes off to sleep.
(2) two daemon processes, each of which have a periodicity of 10 seconds and start at the same time. They also execute a loop of duration 1200 microseconds each, and then sleep.
(3) two daemon processes, one having a periodicity of 10 seconds and another having a periodicity of 10.5 seconds. They start at the same time and execute similar work of duration 2500 microseconds and then sleep.

We have chosen the above three cases as they can be used to simulate the following three scenarios in which jitter sources can affect an application.

(1) A single daemon process, having a unique duration, will get scheduled by itself and sometimes in combination with other daemons. The duration of the daemon will appear as an interruption of unique duration to the application.
(2) Two daemon processes, that start at the same time, take the same amount of time to do their work and have the same periodicity will most likely get scheduled in succession (especially if their priorities are higher than other processes) and their total time of execution will appear as jitter to the application.
(3) Two daemon processes, that start at the same time, take the same amount of time to do their work and have different periodicity will get scheduled by themselves most of the time (and only rarely in succession). The individual times taken by them to finish their work (which in this case are the same) will appear as jitter to the application.

The above three scenarios correspond to the four scenarios described in section II-A, which describe how jitter can affect an application. The only scenario that we currently do not simulate is the one that involves nested execution of interrupts.

1) Experiment 2.1: Single synthetic daemon: In this experiment we start a synthetic daemon process at the same time as the user-level micro-benchmark. The synthetic daemon has a periodicity of 10 seconds. After every 10 seconds, it does work that takes approximately 2500 microseconds to finish. This work can be any operation. We have chosen it to be Linear Congruential Generator (LCG) operation defined by the recurrence relation:

```
x(j+1) = (a * x(j) + b) mod p
```

To ensure that the synthetic daemon does not get descheduled by another process before finishing its work, it raises its priority to the maximum real time priority. After completing the work, it reduces its priority back to normal and goes off to sleep.

We run this experiment for an hour at run level 3 and with N=16 MB for the user-level micro-benchmark. We generate the master distribution for this run and compare it with the master distribution for the default run level 3 (from experiment 1) without the synthetic daemon process. We plot both these distributions, using Parzen window method for kernel density estimation. This is shown in Figure 4. The peak for the jitter caused by the synthetic daemon can be clearly observed around 2500 microseconds (surrounded by a dotted rectangle). Next, we zoom into this peak in the master distribution of this run to find out what daemons contribute to this peak. The histogram buckets corresponding to this peak are shown in Figure 5. The synthetic daemon binary is called "dummydaemon1". The analyzer program concatenates all sources of jitter that occur in succession with an underscore ("_").

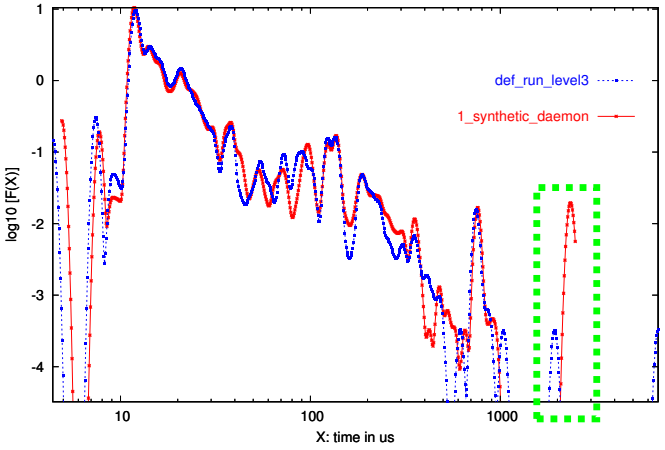

Fig. 4. Comparison of master distributions: default run level 3 and single synthetic daemon

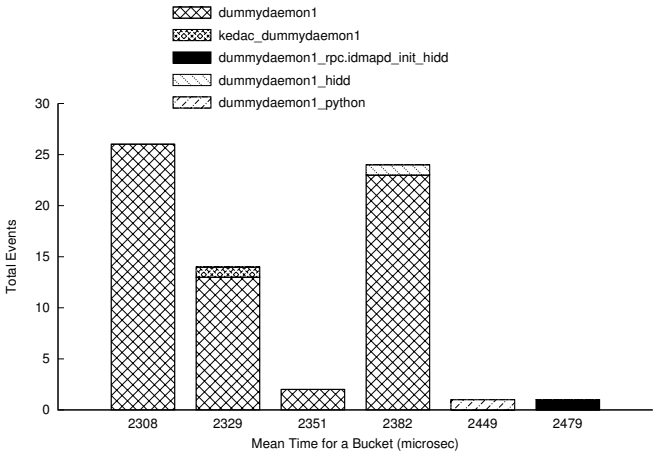

Fig. 5. Histogram buckets from single synthetic daemon master distribution in the range 2000-3000 μs range

2) Experiment 2.2: Two synthetic daemon processes with different periodicity: In this experiment, we start two synthetic daemon processes at the same time as the user-level micro-benchmark. These synthetic daemon processes are identical to the synthetic daemon process in experiment 2.1 except that one of them has a periodicity of 10.5 seconds. The difference

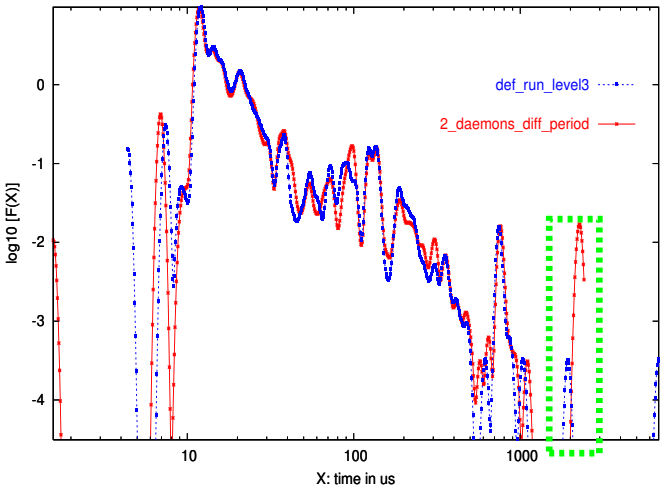

Fig. 6. Comparison of master distributions: default run level 3 and two synthetic daemons with different periodicity

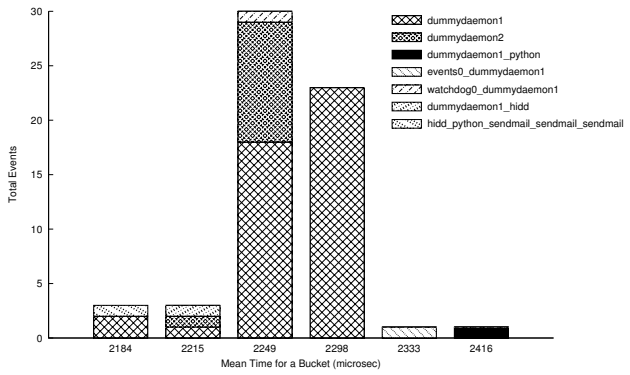

Fig. 7. Histogram buckets from two synthetic daemons (different periodicity) master distribution in the range 2000-3000 μs range

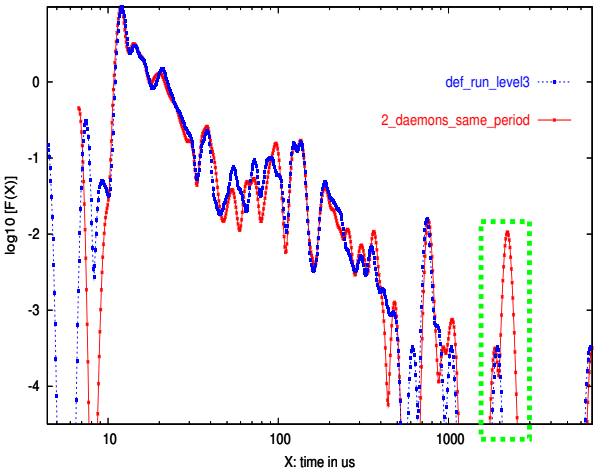

Fig. 8. Comparison of master distributions: default run level 3 and two synthetic daemons with same periodicity

in the periodicity prevents these two synthetic daemons to be scheduled in succession. As in the previous experiment, we run this experiment for an hour in run level 3 and with N=16 MB for the user-level micro-benchmark. The master distribution for this run and that for default run level 3 are plotted using Parzen window method. This is shown in Figure 6. The peak for the jitter caused by the two synthetic daemons can be observed around 2500 microseconds (surrounded by a dotted rectangle). When we zoom into this peak, we find out that there are two main contributors to this peak: the two synthetic daemons - "dummydaemon1" and "dummydaemon2" (as compared to just a single synthetic daemon in experiment 1). The histogram buckets corresponding to this peak are shown in Figure 7. The interesting thing to note in this experiment is that even though two daemons each with an execution time of 2500 microseconds are running (as compared to a single daemon with an execution time of 2500 microseconds in experiment 2.1), the number of interruptions experienced by the application do not get doubled because of the difference in periodicity of the two synthetic daemons.

3) Experiment 2.3: Two synthetic daemon processes with same periodicity: This experiment is similar to experiment 2.2, except that both the synthetic daemon processes have the same periodicity of 10 seconds and each one of them takes only around 1200 microseconds to finish their work. This is done to ensure that these two synthetic daemons get scheduled in succession and the jitter perceived by the user-level micro-benchmark is the sum of the time taken by these daemons (*i.e.* 2500 microseconds, thereby ensuring that the jitter experienced by the benchmark is similar to that in the previous two cases).

This experiment is also run for one hour in run level 3 and with N=16 MB for the user-level micro-benchmark. The master distribution for this run and that for the default run level 3 are shown in Figure 8. The peak for the jitter caused by the two synthetic daemons can be observed around 2500 microseconds. When we zoom into this peak, we find that majority of this peak is being contributed by a single jitter source - "dummydaemon1_dummydaemon2". The histogram buckets corresponding to this peak are shown in Figure 9.

V. Related Work

Several groups in industry and academia have looked at the problem of OS jitter. Most of the studies on OS system jitter, so far, have concentrated either on measuring overall jitter experienced by an application [1] [2] [12] or on estimating the effect of jitter on the scaling of parallel applications [15] [3] [4]. These studies have not really delved into the issue of who are the biggest contributers to OS jitter. There has been some work done towards identifying sources of OS jitter. Petrini et al. identified sources of OS jitter that affected optimal performance on ASCI Q [15]. However, their analysis was limited to that cluster configuration and they did not give details of their methodology. Furthermore, they

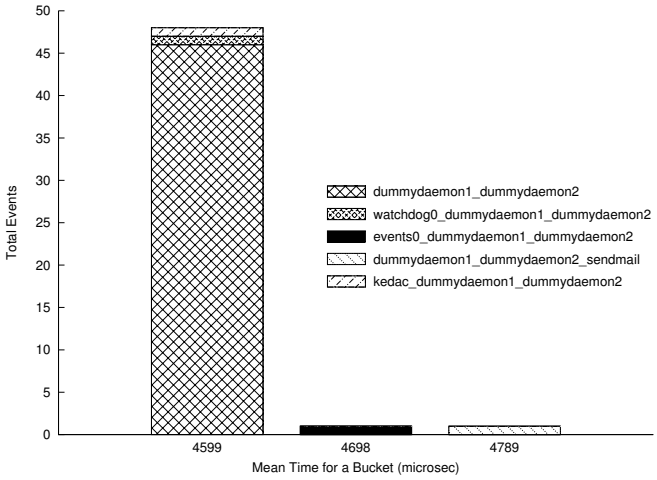

Fig. 9. Histogram buckets from two synthetic (same periodicity) daemons master distribution in the range 2000-3000 μs range

clubbed all kernel activity as one, whereas this paper presents a methodology that can be used to measure the impact of each individual sources of jitter within the kernel. In a follow up paper [16], they described a generic methodology, like ours, that made use of OProfile [10] and kernel instrumentation to identify sources of OS jitter. They restricted their analysis to interrupts and did not describe how new sources of jitter could be detected.

In another work, Tsafrir et al. studied OS noise, more specifically the impact of OS timer interrupts on parallel application performance [17]. Their methodology for determining the jitter component also revolves around micro benchmarking the kernel through use of accurate timers. In terms of the design of the technique, it is similar to ours and presents a much more comprehensive analysis of the effect of timer interrupts. However, the value provided by our tool lies not only in the identification all sources of OS jitter and measuring their impact but also in comparison of various configurations of a system to detect new sources of jitter that can get introduced as software get installed and upgraded. We show what are the daemons and interruptions that occur in an "out of the box" system configured with default run level configuration, and what is their contribution to total jitter.

The technique of sampling the timestamp register at a very high rate in a loop based on the fixed work quantum principle, has been used in various benchmarks for studying OS jitter [2] [12] [18]. We use a similar benchmark loop for collecting timing samples. We enhance this benchmark in two ways. First, we add a kernel tracing mechanism that allows us to associate each interruption to an operating system daemon or interrupt. Second, we ensure that the only instruction executed in the critical loop is the instruction that reads the timestamp register (and some additional instructions that constitute the *for* loop itself). All processing of timing samples is done outside the critical section. This reduces the variance in timestamp deltas and helps us detect even the smallest of interruptions accurately.

VI. Conclusions and Future Work

In this paper we presented the design and implementation of a tool that can be used to identify sources of OS jitter and quantitatively measure their contribution to the total jitter experienced by an application. We presented experimental results of running the tool on Linux in run level 3 and gave details about the top contributers to OS jitter. Our results revealed that while 63% of the total jitter comes from timer interrupts, the rest comes from various system daemons and interrupts, most of which can be easily eliminated. We also presented an experimental validation of our methodology through the introduction of synthetic daemons and described how our tool can be used to detect new sources of OS jitter that get introduced, as software get installed and upgraded on a tuned system.

The tool presented in this paper does not collect any information about cache misses, TLB misses and page faults. We have already enhanced our kernel patch to collect data about page faults and have conducted some preliminary experiments. We intend to do the same with cache misses and TLB misses. On platforms that have a virtualization layer (like the PowerPC), the contribution of virtualization to OS jitter also needs to be accounted for. We also intend to run our tool on the new tickless kernels and measure the total jitter on them.

One of the limitations of the current tool is that it prints the kernel instrumentation data to files in each round and the processing of these files to create master distributions is done as a post processing step. We intend to integrate the data analyzer program with the user-level micros benchmark so that the processing of kernel instrumentation data can be done at the end of each round itself to create the master distributions. This will allow the tool to be run for really long hours (may be days) without worrying about the length of the trace files, and detect those sources of OS jitter whose periodicity is of the order of days. We are also currently involved in deploying this tool as part of a larger toolkit that can identify anomalous nodes in a cluster and point to sources of OS jitter that cause such nodes to behave anomalously.

References

[1] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System Noise, OS Clock Ticks, and Fine-grained Parallel Applications," in *ICS*, 2005.

[2] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale," in *IEEE International Conference on Cluster Computing*, 2006.

[3] S. Agarwal, R. Garg, and N. K. Vishnoi, "The Impact of Noise on the Scaling of Collectives," in *High Performance Computing (HiPC)*, 2005.

[4] R. Garg and P. De, "The Impact of Noise on the Scaling of Collectives: An Empirical Evaluation," in *High Performance Computing (HiPC)*, 2006.

[5] S. M. Kelly and R. Brightwell, "Software architecture of the light weight kernel, catamount," in *47th Cray User Group Conference*, May 2005.

[6] R. Brighwell, R. Reisen, K. Underwood, T. B. Hudson, P. Bridges, and A. B. Maccabe, "IEEE International Conference on Cluster Computing," in *A Performance Comparison of Linux and a Lightweight Kernel*, 2003.

[7] "Right-weight Linux Kernel Project at Los Alamos National Laboratory." [Online]. Available: http://public.lanl.gov/cluster/projects/index.html

[8] L. S. Kaplan, "Lightweight Linux for High-Performance Computing," in *LinuxWorld.com*, December 2006. [Online]. Available: http://www.linuxworld.com/news/2006/120406-lightweight-linux.html
[9] "Zeptoos: The small linux for big computers." [Online]. Available: http://www-unix.mcs.anl.gov/zeptoos/
[10] "OProfile: A System Profiler for Linux." [Online]. Available: http://sourceforge.net/projects/oprofile/
[11] "Linux kernel scheduler statistics." [Online]. Available: http://www.mjmwired.net/kernel/Documentation/sched-stats.txt
[12] "Selfish detour benchmark suite." [Online]. Available: http://www-unix.mcs.anl.gov/zeptoos/software/index.php
[13] E. Parzen, "On the estimation of a probability density function and the mode," in *Annals of Math. Stats., Vol. 33, pp. 1065–1076*, 1962.
[14] R. P. W. Duin, "On the choice of smoothing parameters for parzen estimators of probability density functions," in *IEEE Transactions on Computers, Vol. C-25(11), pp. 1175–1179*, 1976.
[15] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8192 Processors of ASCI Q," in *ACM Supercomputing*, 2003.
[16] R. Gioiosa, F. Petrini, and K. Davis, "Fourth IEEE International Symposium on Signal Processing and Information Technology," in *Analysis of System Overhead on Parallel Computers*, 2004.
[17] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications," in *International Conference on Supercomputing (ICS)*, 2005.
[18] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the Effects of Operating System Interference on Extreme-Scale Parallel Machines," Tech. Rep., Jan 2007.