Exploratory Study on the Linux OS Jitter

Elder Vicente, Rivalino Matias Jr.

School of Computer Science – Federal University of Uberlandia (UFU), MG – Brazil

elder@mestrado.ufu.br, rivalino@fc.ufu.br

Abstract. *We present the results of an experimental study that quantifies the effects of different sources of OS Jitter in the Linux operating system. We found that the processor topology, especially regarding the shared processor cache, has the most significant influence in terms of OS Jitter. Also, we found that in order to reduce the impact of OS Jitter on a given application, the number of computational phases in the algorithm is significantly more important than the number of distributed processes or compute nodes.*

Resumo. *Este trabalho apresenta os resultados de um estudo experimental que quantifica os efeitos de diferentes fontes de OS Jitter em um sistema operacional Linux. Verificou-se que, o número de fases computacionais do algoritmo é significativamente mais importante que o número de processos distribuídos ou nós de computação.*

1. Introduction

The advances in many areas of society have been demanding more computational power to perform complex simulation models. This scenario has forced an increasing demand for high performance computing (HPC). The concept of distributed processing is fundamental because it enables to divide a task in many smaller subtasks and run them, in parallel, on different computing nodes [Garg and De 2006]. Clusters of computers have been increasingly used for high performance distributed processing as alternative for supercomputers. In a typical cluster-based HPC environment, each computing node executes its own operating system. Thus, in addition to the user's application running on the node, there are also operating system (OS) internal routines being executed regularly on the same node. This means that OS routines such as hardware interrupt handlers, kernel threads and timers, and administrative processes, all of them compete with the user application for the node computing resources. This scenario leads to a situation where during the user application runtime it suffers periodically from interferences caused by the OS internal routines. These interferences have been studied (e.g., [Jones et al. 2003], [De et al. 2007]) and reported in the literature as OS Jitter. HPC cluster-based applications are typically designed to run in a paradigm of parallel processing, where instructions are programmed to be executed in many computational phases [Garg and De 2006]. In this approach, after all distributed processes finish a given computational phase they all synchronize and then start executing the subsequent phase ([Gioiosa et al. 2004], [Agarwal et al. 2005] and [Tsafrir et al. 2005]). Since a new phase only starts after all distributed processes conclude the current phase, synchronizing the computing time of all application processes is critical. The last process that terminates a given phase determines the time length of the phase. So, reducing the runtime variability in

each node is a major requirement, given that the occurrence of unexpected delays in a node will spread along other nodes involved in the same computational phase, bringing a longer time to complete the whole task. In this paper, we present an experimental study that quantifies the effect of different sources of OS Jitter in the Linux operating system. We choose the Linux OS because it is used in more than 90% of the HPC clusters listed on the Top500 supercomputer website.

2. Methodology

We adopt the design of experiments (DOE) method [Montgomery 2000] to conduct our study. This method requires the execution of several tests, where controlled changes are made on selected factors of the system, in order to observe and measure the effect of these changes on response variables. Each investigated factor is evaluated with respect to a predetermined set of values (levels). A factor at a specified level is called a treatment. The DOE method allows us to quantify the influence of a factor, individually or combined with other factors, on a specific response variable, we apply it to measure the impact of different sources of OS Jitter on the execution time of a typical HPC application. The HPC application used is a CPU-bound program that performs a matrix multiplication algorithm. Thus, our control group is composed of all treatment executions where the sources of OS Jitter are present. The experimental groups are those treatments where we control the presence and levels of each OS Jitter source investigated. Each treatment test is executed following a protocol: *i*) configure the test bed according to the treatment specification; *ii*) collect the start time (T_1); *iii*) execute the matrix multiplication routine; *iv*) collect the end time (T_2); *v*) replicate steps two to four 53 times; *vi*) write all computation times, $(T_2 - T_1)_{i=1..53}$, into a log file. We replicate every treatment test 53 times in order to have a sample size large enough to ensure a proper estimation of experimental errors and to determine if the differences among treatments are statistically significant. The turnaround time of step three is approximately 10 minutes. To avoid that a treatment test influences the execution of the subsequent treatment, we restart the OS kernel right before starting the execution of a new treatment. For each treatment, we discard the first three replications considering that their results are more likely to suffer influences from file system and processor caches. Thus, our final dataset, per treatment, is composed of 50 run times. To analyze the experimental results we use different statistical techniques. First, we identify which treatments are statistically different. We do not use a parametric approach, such as analysis of variance (ANOVA), because the dataset obtained does not fit the necessary assumptions, especially regarding to independent and identically distributed (i.i.d.) observations. Thus, we use the non-parametric Kruskal-Wallis test [Vam and Vidakovic 2007], which allows us to use ranks of observations, providing statistics equivalent to those obtained with parametric tests. We compare all treatments and the difference between their response variables (run times) are statistically significant if the *p-value* is less than 0.05 (α=5%). For the setup of treatment combinations, and sequence of runs, we adopt the signal matrix method [Jain 1991], which was arranged according to the Yates' order [Montgomery 2000]. Solving the signal matrix, we have a ranking of individual and combined factors that are sorted by their influence degree on the application runtime. Supported by this ranking we can identify the OS Jitter sources with more impact on the test application.

3. Experimental Study

In order to conduct the tests, we use a test bed based on a computer composed of two quad-core sockets (Intel Xeon E5620 2.40GHz), 24 GB memory, and 1 TB SATA disk. The computer microarchitecture has a three-level cache per CPU socket, being the last level (L3) of 12MB and shared by all cores of the same socket. Each core has two individual levels of cache, L1 (32KB) and L2 (256KB). Figure 1 illustrates the processors topology. For the sake of simplicity, we refer to each core as PU #0 to PU #7, where PU stands for processor unit. The test program runs only on PU #1, where we rigorously control the enabling and disabling of OS Jitter sources. The remaining cores are used according to each treatment specification. Our experimental plan is created to evaluate quantitatively the effects of different sources (factors) of OS Jitter on the total run time of the HPC test application. We encode each evaluated factor using upper case letters. Each factor assumes two levels represented by symbols (+) and (-). The level (-) means that the OS Jitter source (factors) is disabled, and (+) means enabled.

3.1. Experiment #1

In Exp. #1, we evaluate five factors. Factor A represents the operating system runlevel. At level (-) the runlevel is 5, which means a higher number of service loaded. Differently, the level (+) sets a minimal number of services loaded. This factor is related to the number of system processes running concurrently with the user application. Factor B represents the kernel timers. Kernel timers are used to allow the execution of kernel or user level routines at a given future time. The level (-) of this factor indicates that we disable the execution of timers on the same processor (PU #1) that executes the test application. The level (+), the timers can be programmed to run on the processor PU #1. We always move timers from PU #1 to PU #0, where PU #0 is the processor we defined to run all timers from PU #1 when this factor is at level (-). This allows us to observe the direct interference of timers. Factor C represents the hardware interrupt request (IRQ). This factor at level (-) indicates that the processor PU #1 does not receive interrupt requests (except from the timer interrupt); All IRQs are redirected to PU #0. On the other hand, this factor at level (+) all IRQs are handled only by the PU #1. This allows us to observe the direct interference of IRQs. Factor D represents the processor affinity of the system processes. This factor at level (-) means that processor affinity is disabled, and thus all system processes can be executed in any processor. This factor at level (+) means that we enable the processor affinity and set all system processes to run only on PU #0. This allows us to observe the direct interference of system processes. Factor E represents the timer interrupt. This factor at level (-) indicates that we disable this interrupt on processor PU #1, where the test application is running. For this experiment, the results show that 91.23% of the test application run time variation is caused by factor E (timer interrupt) and the other factors did not show important contributions when compared with the timer interrupt. After comparing all pairs of treatments, we notice that every treatment where factor E (timer interrupt) is disabled is considered statistically different from the treatments with this factor enabled.

3.2. Experiment #2

This experiment consists of six factors. The first five factors (A..E) are the same used in Exp. #1, and we introduce the factor F that represents a CPU-bound workload running

in background. A process also running a matrix multiplication program implements this background workload. The factor F at level (-) means that the application performing the background workload is running in a processor (PU #5) that is not sharing L3 cache with the processor PU #1. This factor at level (+), the background workload is running in a processor (PU #2) that shares the L3 cache memory with PU #1. This allows us to observe the interference of other processes sharing processor cache memory with the test application. The test application has an average working set size of 12248 kB (11.96 MB), and the process performing the background workload has 12228 kB (11.94 MB). When evaluating the scenario with shared cache (level +), both processes compete for the entire L3 cache memory. We split the results in four groups (G1 to G4). Since G1 and G2 reproduce the treatments evaluated in Exp. #1, the results obtained were practically the same discussed in previous subsection. The factor F is disabled in all treatments of G1 and G2. On the other hand, this factor is enabled in all treatments of G3 and G4, where in G3 the factor E (timer interrupt) is disabled and in G4 is enabled. All treatments of G3 the test application did not suffer influence of timer interrupts, but from sharing the L3 cache. In G4 both influences, timer interrupts and sharing processor cache, are present. That the individual contribution of factors E (timer interrupt) and F (shared processor L3 cache) on the application run time are very similar, the results show that 33.90% of the test application run time variation is caused by factor F and 24.76% is caused by factor E. We observe that 32.84% of the total variability could not be explained by our factorial design. This may be due to experimental errors introduced with the activation of factor F.

3.3. Experiment #3

This experiment introduces a network workload in addition to the factors evaluated in Exp. #1. This background network workload allows us to observe the interference of network interrupts on the test application. For all evaluated treatments, the network workload runs on PU#2. The network workload is based on an application receiving 500-byte UDP datagrams in a continuous way. Some treatments tested in experiments #1 and #2 were not evaluated, which are related to the IRQ factor in level (+) and timer interrupt factor in level (-). This is necessary because disabling the timer interrupt on PU#1 makes the kernel routines, responsible for the datagram packet processing, work improperly, which causes the loss of network packets. It occurs because these routines use kernel timers that require the timer interrupt enabled. The same applies to the IRQ with respect to the network card interrupt handling. Based on the results, we observe that the joint contribution of factors C (IRQ) and E (timer interrupt) on the application run time is high. The numerical analysis, showed that the average run time is: G1 (9.98 minutes), G3 (10.01), and G4 (14.54), note that we also organized the treatments in four groups. So, when the hardware interrupt request and timer interrupt factors are enabled simultaneously on PU#1, the average run time increases significantly (45%). The network interrupts may have a greater impact than the worst case of sharing cache. We observe that factor C and the iteration CE have very close contributions (approx. 46%) and only 0.003% of the total variability could not be explained by our factorial design.

3.4. Simulation

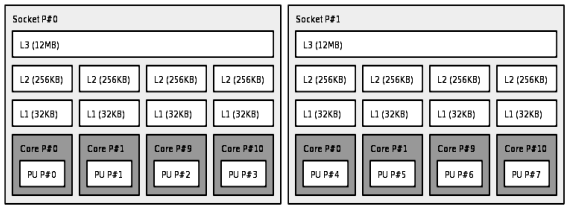

Figure 1. Processor topology

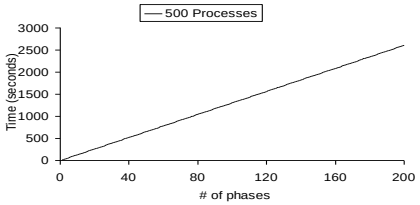

Figure 3. Effect for 500 processes

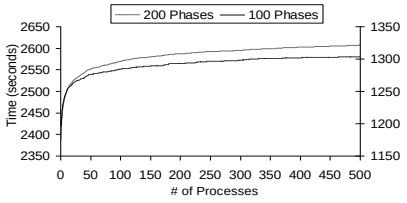

Figure 2. Effect for 100 and 200 phases

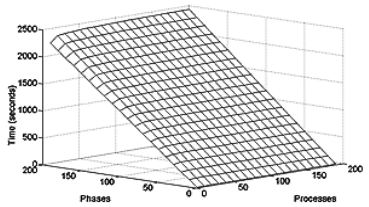

Figure 4. Summary of the simulation

We simulate the impact of the OS Jitter on a HPC application running in multiple compute nodes and composed of multiple computational phases. First, we select two treatments (T10 and T23) from Exp. #1. In T10 all investigated factors are disabled and in T23 they are all enabled. Based on these treatments, we generated a third dataset with the differences between T23 and T10. This new dataset is used to obtain the probability density function (*pdf*) of the run time delay caused by the OS Jitter. We conduct a goodness-of-fit test, with 95% of confidence level, and found that this sample follows a normal distribution. Next, we use this *pdf* to simulate the delay occurrences on each computational phase of each instance of the application running on multiple computing nodes. We vary the number of computational phases per process (1 to 200) and the number of processes (1 to 500). We consider only one application process running per node, so varying the number of processes means changing the number of compute nodes. The simulation results show that when we vary the amount of processes (or compute nodes), to any amount of phases, the application execution time growth logarithmically. In Figure 2, for different number of computational phases, we observe that for few processes (e.g., < 20) the growth of the curve is quite sharp. For more than that, the increase in the application time tends to moderate. This happens because with few processes taking part at each phase, there is a smaller probability that in a given phase some of these processes suffer from OS Jitter influences whose delay is close to the highest possible values. If the amount of processes rises (e.g. > 20), then increases the probability of delays caused by OS Jitter, per phase, to be close to the highest observed delays. Thus, the average delay is close to the highest possible delay. When raising the number of phases the application run time rose linearly (see Figure 3). Increasing the number of phases, the probability of delays caused by OS Jitter inside of each cluster node also increases. Since the nodes are working in parallel, the summation of these increased probabilities explains this linear behavior. Summarizing the simulation results, in Figure 4 we present the sensitivity analysis of the runtime delay with respect to the number of processes and number of phases. We conclude that in order to reduce the effects of OS Jitter on the runtime of distributed applications, it is a major requirement to reduce the number of computational phases per processes, even though it would require a significant increase on the number of processes (or nodes).

5. Final Remarks

The recent advances in areas such as power saving and processor topology have changed the way the OS kernels work. These changes consequently affect how the OS routines interfere on the user applications. The controlled use of features such as CPU frequency scaling and tickless kernel have not been considered in the previous studies, requiring update. In addition to including these aspects, we also use a comprehensive methodology based on robust statistical techniques that allow us to analyze the experimental data in a rigorous way. Several previous works have indicated the timer interrupt as the most influential source of OS Jitter. However, we found that sharing processor cache has a similar impact on the application run time than timer interrupts, or even more, when these two factors are combined. We also observe that the number of computational phases in a distributed application has a higher impact on the runtime delay, due to OS Jitter, than the number of processes running across the cluster. In the future works, we will evaluate the OS Jitter to other types of workloads, including I/O-bound and hybrids (CPU-bound and I/O-bound).

References

[Agarwal et al. 2005] Agarwal, S., Garg, R., and Vishnoi, N. K. (2005): The impact of noise on the scaling of collectives: a theoretical approach. In *Proc. of IEEE Int'l Conf. on High Performance Comp*., Goa, India, Dec. 280–289.

[De et al. 2007] De, P., Kothari, R., and Mann, V. (2007): Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *Proc. of the IEEE International Conference on Cluster Computing*, Washington, USA, 331–340.

[Garg and De 2006] Garg, R. and De, P. (2006): The impact of noise on the scaling of collectives: an empirical evaluation. In *Proc. of 13th IEEE International Conference on High Performance Computing (HiPC)*, Bangalore, India.

[Gioiosa et al. 2004] Gioiosa, R., Petrini, F., Davis, K., and Lebaillif-Delamare, F. (2004): Analysis of system overhead on parallel computers. In *Proc. of IEEE Symposium of Signal Processing and Information Tech*, 387–390.

[Jain 1991] R. Jain (1991): *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, New York, NY.

[Jones et al. 2003] Jones, T. R., Brenner, L. B., Fier, J. M. (2003): Impacts of operating systems on the scalibility of parallel applications. Tech. Rep. UCRL-MI-202629, Lawrence Livermore National Laboratory.

[Montgomery 2000] D. C. Montgomery (2000): *Design and Analysis of Experiments*. John Wiley, 3rd edition.

[Tsafrir et al. 2005] Tsafrir, D., Etsion, Y., Feitelson, D. G., and Kirkpatrick, S. (2005): System noise, os clock ticks, and fine-grained parallel applications. In *Proc. of Int'l Conf. on Supercomputing*, New York, NY, USA 303–312.

[Vam and Vidakovic 2007] Vam, P.H. and Vidakovic, B. (2007): *Nonparametric Statistics With Applications to Science and Engineering*. Wiley-Interscience.