2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing

Compensating for Timing Jitter in Computing Systems with General-Purpose Operating Systems

Abhishek Dubey[†] Gabor Karsai[†] Sherif Abdelwahed[‡]

[†]Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN
[‡]Electrical Engineering and Computer Science, Mississippi State University, Mississippi State, MS

Abstract

Fault-tolerant frameworks for large scale computing clusters require sensor programs, which are executed periodically to facilitate performance and fault management. By construction, these clusters use general purpose operating systems such as Linux that are built for best average case performance and do not provide deterministic scheduling guarantees. Consequently, periodic applications show jitter in execution times relative to the expected execution time. Obtaining a deterministic schedule for periodic tasks in general purpose operating systems is difficult without using kernel-level modifications such as RTAI and RTLinux. However, due to performance and administrative issues kernel modification cannot be used in all scenarios. In this paper, we address the problem of jitter compensation for periodic tasks that cannot rely on modifying the operating system kernel. ; Towards that, (a) we present motivating examples; (b) we present a feedback controller based approach that runs in the user space and actively compensates periodic schedule based on past jitter; This approach is platform-agnostic i.e. it can be used in different operating systems without modification; and (c) we show through analysis and experiments that this approach is platform-agnostic i.e. it can be used in different operating systems without modification and also that it maintains a stable system with bounded total jitter.

1. Introduction

Distributed cluster computing has evolved as an attractive computing paradigm for solving a number of huge scientific computation and discovery problems. These large clusters are built using commodity hardware and general-purpose operating systems such as Linux. They yield the highest performance per dollar but exhibit intermittent faults, which can result in systemic failures when operated over a long continuous period for executing analysis campaigns. Manual administration is necessary, but is generally slow to respond to the intermittent faults. Therefore, we need to supplement the administrative system with a control system that can provide fault prediction, isolation and subsequent mitigation. We are currently designing such a system for LQCD computing clusters at Fermi National Laboratory, Illinois, USA[1].

Most commonly, digital implementations of controllers operate in a periodic fashion, sampling control signals, computing the current state of the system, and performing an actuation based on the computations. In distributed setting, several distributed agents periodically execute on worker nodes and provide sensory information that is then fused centrally or hierarchically to provide a snapshot of the global state of the system [7].

Periodic task execution can be achieved by either using self-scheduling threads that use operating system primitives such as sleep/alarm or register directly with operating system kernels that have built-in support for periodic tasks. Alternatively, a user space discrete event scheduler can be used that arbitrates and wakes up several related sub-tasks as shown in figure 1. This approach is similar to the release guard approach in that a guard-process releases subtask periodically [17]. Release guard protocol ensures that the lower bound on the time-interval between two executions of the same task will be at least greater than the period. However, it does not guarantee an upper bound on the task release intervals.

Periodic task invocations suffer from jitter, defined as the maximum deviation between start time and the release time among all instances of a periodic task [5]. It ranges on the order of microseconds even on hard real-time systems. However, on general-purpose operating systems such as Linux that are designed for good average performance, the typical jitter can range from a few milliseconds when the system is not busy to a few seconds when the system is busy. By construction, computing nodes in cluster use Linux and not a hard real-time operating system. The jit-

[1] http://www.usqcd.org/fnal/

1555-0885/09 $25.00 © 2009 IEEE
DOI 10.1109/ISORC.2009.28

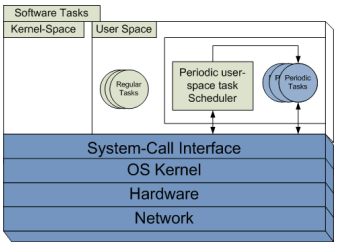

Figure 1. User space schedulers that maintain period by using periodic sleep cycles.

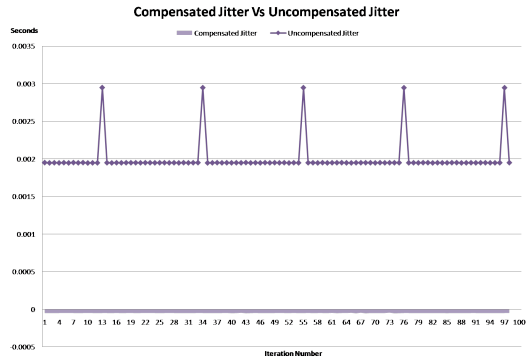

Figure 2. A plot of 99 samples of timing jitter obtained for a task with 1 second period on Red Hat Linux 2.6.9-67.

ter in sensor tasks is a cause of concern for a reliable subsystem. The global controller expects information for all sensors to arrive periodically according to the pre-arranged schedule. However, deviations from that schedule due to jitter can create false positives. For example, a sensor such as heartbeat periodically sends a message to a recipient to indicate that a computing node is alive. Delay of heartbeat due to jitter can cause false positives.

Ensuring that jitter is bounded is a common requirement in real-time operating systems. There exists prior work in the field of jitter reduction and compensation. In [10], the authors studied the degradation of performance in real-time control systems interacting with actual physical systems due to jitter. They compensated for jitter by changing the parameters to make the controller more robust to jitter. Baruah et al. presented kernel space algorithms for reducing jitter in [4]. In the same paper, they outlined a few properties that any jitter control algorithm must have. Some of these properties are: (a) it should not buffer a job that is to be executed; (b) CPU cannot be kept idle if some job has to be executed, (c) when a job completes, it should release the CPU immediately, and (d) the online scheduling overhead should not be as small as EDF.

In this paper, we propose a user space scheme that actively compensates for jitter. This is essential so that the jitter control mechanism can be implemented quickly on new installations and on heterogeneous operating systems. We model the jitter accumulated during each period as a disturbance and then use a discrete-time proportional and integral (PI) feedback controller [11] to compensate for total jitter before the next scheduled activity.

Figure 2 shows a plot of 99 samples of timing jitter obtained for a task with 1 second period on Red Hat Linux 2.6.9-67. The regular pattern in uncompensated jitter suggests a common origin. Figure 2 also shows the compensated jitter profile for the same task when the periodicity of 1 second was implemented using the approach outlined in this paper with a smaller sleep cycle of 20 milliseconds. Average CPU utilization during the whole test was less than 1 percent.

The outline of this paper is as follows: We describe the problem and provide a formal problem statement in section 2. Section 4 introduces our approach. We analytically derive the conditions necessary for controller stability by using discrete-time domain models for the controller and the plant in section 4.3. Finally, we present experimental results from two common operating systems, Linux and Windows in section 5. Section 6 summarizes related research. Then, we conclude with discussions for future work.

2. Problem Context

In this section, we mention two use-cases where timing jitter in periodic task execution affects the quality of service.

2.1. Scheduler framework for Sensors

Our fault-tolerant framework contains a number of distributed periodic sensors. Sensors present on a computing node are executed by using a user space discrete event scheduler [6] (Algorithm 1). Sensors are executed by sleeping periodically for a time T which is equal to the earliest periodic deadline. This sleep is achieved by alarm and signal handlers in Linux and a regular sleep in Microsoft Windows. The precision of periodicity depends upon the accuracy of the sleep function provided by the operating system. However, in many cases the standard Linux kernel provides no upper bound on the difference between the actual times of sleep compared to the requested time. Moreover, jitter accrued in sleep increases as the CPU utilization increases. This is often the case on a cluster which is being actively used. In this case, we need to be able to reduce the jitter accrued during each sleep cycle.

Algorithm 1 Sensor Scheduler

Input: S {Set of periodic sensors. Each sensor has two variables: P(Time period), C (current clock Value)}
Pre Condition: $(\forall s \in S)(s.C = P)$
1: **loop**
2: Set $T = minimum((\forall s \in S)(s.C)$
3: Set Alarm for T {Alarm can be implemented by sleep or using signals in both Linux and Windows}
4: $(\forall s \in S)(s.C \leftarrow s.C - T)$
5: $(\forall s \in S)(s.C = 0 \implies Execute(s))$
6: $(\forall s \in S)(s.C = 0 \implies (s.C \leftarrow s.P))$
7: Run any aperiodic sensors if present.
8: **end loop**

2.2. General Purpose Time Triggered Execution Platform: Frodo

Frodo [18] is an abstract virtual machine, for executing time triggered control tasks of high-confidence systems coupled with a message controller that provides the time triggered [8] transmission of data message across networked nodes, all of which is implemented using readily available infrastructure. The current platform implementation uses embedded PC-s running on Linux 2.6.x kernels and standard Ethernet UDP network for communication.

The current implementation of FRODO does not allow the preemption of tasks, i.e. only one periodic control task is released for execution at any given time. In order to maintain the timely execution of the control tasks according to the schedule, FRODO is also responsible for terminating executing tasks that have yet to finish prior to reaching their worst-case execution time (WCET). In a time-triggered system, the schedules for tasks are statically decided. Frodo implements the schedule by using sleep cycles on all nodes. Jitter in sleep cycles affect the number of tasks that can be executed to completion on all the nodes.

3. Problem Description

Consider a generic periodic task τ with a time period T. The ideal release time of such a task would be a sequence $< kT >_{k=1}^{k=\infty}$ relative to some time, t_ϕ. Let $s(kT)$ be the start time of k^{th} instance of this task such that the start is delayed by $tj(kT)$ with respect to the expected start time, $t_\phi + kT$. Then total jitter, $tj(kT) = s(kT) - kT - t_\phi$.

Now, consider the $k + 1^{th}$ instance of the same task. If the schedule is set by using sleep for T time, then it can be seen that the relative time difference between the two start times, $s((k+1)T) - s(kT) \geq T$. Total jitter will keep on increasing in such a case. The proof is as follows:

$$\begin{aligned} tj((k+1)T) &= s((k+1)T) - (k+1)T - t_\phi \\ &\geq s(kT) + T - (k+1)T - t_\phi \\ &\geq s(kT) - kT - t_\phi + T - T \\ &\geq tj(kT) \end{aligned} \tag{1}$$

Equation 1 implies that total jitter will keep increasing if at each instance we set the time for next instance by using the fixed interval T. At any point, $tj(kT)$ will represent all the delays accumulated over time till that point and will be equal to the absolute relative jitter until that instance.

4. Controller Design

4.1. Plant and Controller Model

The discrete-time plant model is given in equation 2. The state variable in this equation is the total jitter at any time sample, $tj(kT)$. $d(kT)$ is the finite jitter accrued during that sleep iteration. For brevity, we will drop the term kT and only use k.

$$tj(k+1) = tj(k) + d(k) \tag{2}$$

Empirically, we have observed that the disturbance during each sleep cycle is proportional to the average CPU utilization and the current priority of the process.

4.2. Feedback Controller

Feedback control has been successfully employed for a long time in analog systems. One of the basic feedback configurations is the proportional, integral and derivate (PID) scheme. Basic principle of the PID control scheme is to act on the control variable through a combination of proportional action, integral action and the derivative action. The proportional action is proportional to the error signal, which is the difference between reference input and the feedback signal. Integral action is proportional to integral of error signal and the derivative action is proportional to the derivative of the error signal [12]. In discrete time systems, the integral and derivative components are approximated by trapezoidal summation and difference equation respectively. Discrete-time PID controller equation is given as:

$$\begin{aligned} c(k) =& K[e(k) + \frac{T}{T_i}\sum_{j=1}^{j=k}\frac{e(j-1)+e(j)}{2} + \\ & \frac{T_d}{T}[e(k) - e(k-1)]] \end{aligned} \tag{3}$$

Where, T is the sampling time period, e is the error signal, c is the controller output, K is the proportional gain, T_d is the derivative time constant and T_i is the integral reset time constant.

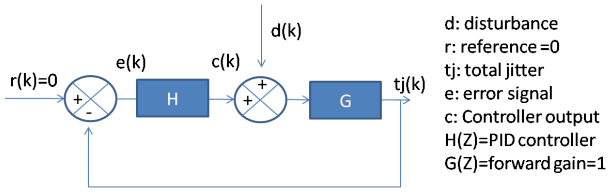

Figure 3. The feedback control loop for the sensor scheduler

Figure 3 shows the control loop for plant specified in equation 2. We implemented the controller using a first order system (PI controller) because we found that quick, sudden changes in current jitter values were causing instability in the derivative term.

The state space variable as mentioned in equation 2 is total jitter, $tj(k)$. The reference or the ideal value for total jitter is zero. Therefore, from figure 3 we can deduce that the error signal is given by $e(k) = -tj(k)$. The plant and controller equations are:

$$tj(k+1) = tj(k) + d(k) + c(k), \text{ where} \tag{4}$$

$$c(k) = -K[tj(k) + \frac{T}{T_i}\sum_{j=1}^{j=k}\frac{tj(j-1)+tj(j)}{2}] \tag{5}$$

In Z-domain the equations become:

$$C(z) = -[K_p + \frac{K_i}{(1-z^{-1})}]TJ(z), \text{ where} \tag{6}$$

$$K_p = K - \frac{K_i}{2} \text{ is the effective proportional gain} \tag{7}$$

$$K_i = \frac{KT}{T_i} \text{ is the integral gain} \tag{8}$$

4.3. Transfer Function of the Control Loop

Figure 3 shows that $C(z) = -H(z)TJ(z)$. From equation 6 we can write

$$H(z) = [K_p + \frac{K_i}{(1-z^{-1})}] \tag{9}$$

Further, we can write the transfer function, $TF(z) = \frac{1}{1+H(z)}$ as:

$$TF(z) = \frac{1}{1+[K_p + \frac{K_i}{(1-z^{-1})}]} \tag{10}$$

For stability, poles of TF must lie within the unit circle in z-domain or $|z| < 1$. From equation 10, the characteristic equation for finding the poles is $1 + [K_p + \frac{K_i}{(1-z^{-1})}] = 0$. Therefore, the pole is at $z = \frac{1+K_p}{1+K_p+K_i}$. Since K_p and K_i are real, the stability criterion implies:

$$\begin{gathered} -(1+K_p+K_i) < (1+K_p) < (1+K_p+K_i) \\ i.e.\ (1+K_p+K_i) + (1+K_p) > 0, \\ and\ (1+K_p+K_i) - (1+K_p) > 0 \\ i.e\ K_i > 0,\ and\ K_p \geq 0 \end{gathered} \tag{11}$$

Equation 11 implies that the effective proportional constant can be zero. But the integral constant should be chosen to be a positive real number. To rephrase, we need to maintain a history of previous values of total jitter to be able to compensate for current jitter in a stable fashion.

Algorithm 2 Compensated Sleep Implementation

Input: Expected Sleep period T. Maximum number of Iterations $Count$

1: Initialize $T1 \leftarrow CurrentTime()$,$Iterm \leftarrow 0$ {Integral Term}, $c(1) \leftarrow 0$,$k \leftarrow 1$,$tj(1) \leftarrow 0$
2: **repeat**
3: sleep($T + c(k)$)
4: $T2 \leftarrow CurrentTime()$
5: $d(k) \leftarrow T2 - T1 - T$
6: $tj(k+1) \leftarrow tj(k) + d(k)$
7: $Iterm \leftarrow Iterm + (tj(k+1) + tj(k))/2$
8: $c(k+1) \leftarrow -(min(T, K_i * Iterm + K * tj(k+1)))$
9: $k \leftarrow k+1$
10: **until** k = $Count$

Algorithm 2 is the modified algorithm for implementing sleep with feedback controller. The maximum correction that can be applied is equal to the minimum time period T. There can be two approaches to using this algorithm:

A1 Pass a value of period Δ to the subroutine i.e. $T = \Delta$. Set $Count$ to the total number of periods. In this approach, a task scheduled to be executed will be released just after step 3 of the algorithm i.e. sleep step.

A2 Implement a period of Δ as a number of small periods $\delta < \Delta$. Pass δ to the algorithm. Set $Count = \Delta/\delta$. Task will be released after step 10 i.e. when the sleep algorithm returns in this approach.

Approach A2 is preferable when we want to use the control algorithm as just a single instance high resolution sleep

timer. Approach A1 is preferable when there are a number of consecutive sleeps to be implemented. One can also implement approach A1 by using approach A2 in a loop and preserving the state variables $Iterm$ and tj between subsequent invocations of approach A2. We used approach A1 in experiments described in section 5.

In case of multiple schedulers, we can use independent controllers. Independence here implies the control loop independence. We will show some results in this regard in Section 5.3.

4.4. Steady State Error Analysis

For steady state error, we assume that the system is stable and then use the final value theorem. The final value theorem states that if the system error is given by $e(k)$, with $E(z) = \mathcal{Z}[e(k)]$, then the steady state value of the error, e_{ss}, is:

$$e_{ss} = \lim_{k\to\infty} e(k) = \lim_{z\to 1}[(1-z^{-1})E(z)] \qquad (12)$$

Since $E(z) = -TJ(z)$ (see figure 3), from equation 10 we can write $E(z) = -TF(z)D(z)$. Hence, from equation 12 we can conclude that the steady state error is given by

$$e_{ss} = -\lim_{z\to 1}[\frac{(1-z^{-1})D(z)}{1+[K_p + \frac{K_i}{(1-z^{-1})}]}], \text{ or}$$

$$e_{ss} = -\lim_{z\to 1}[\frac{(1-z^{-1})^2 D(z)}{(1+K_p)(1-z^{-1})+K_i}] \qquad (13)$$

Note: Arguably, we can achieve jitter control by setting the next sleep duration as the difference between the next expected release time and the current time. However, this will be same as setting the sleep duration based on the current jitter i.e. the last expected release time and the current time. This scheme is equivalent to a proportional control. It is a well known fact that proportional control cannot achieve a non-zero steady state error. Therefore, if we do not use the integral controller we will never be able to achieve a steady-state of zero jitter.

For a unit-step disturbance, $d(kT) = 1$, we know $D(Z) = \mathcal{Z}[d(kT)] = \frac{1}{1-z^{-1}}$. Plugging this value into equation 13, we see that the steady state error for a unit-step disturbance will be 0 only if $K_i \neq 0$.

5. Experiments on Linux and Windows OS

In this section, we will present results obtained by implementing the sensor scheduler with feedback loop i.e. algorithm 2 for controlling the total jitter. All these experiments use a value of $K_p = 1.0$ and $K_i = 1.2$. We chose these values empirically.

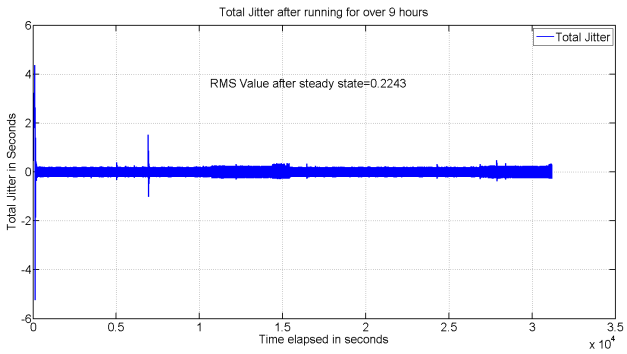

Figure 4. Total Jitter accumulated in last 9 hours.

These experiments were carried out on Linux, since it is the operating system of choice in our computing cluster. Figure 4 shows the total jitter accumulated by the sensor scheduler over a period of 9 hours. As can be seen in the figure, the root mean square (RMS) value of the jitter was 0.2243, only a 4.4% of the sampling time period of 5 seconds.

Next, we present results from specific experiments that we conducted for judging the effects of various parameters such as CPU load, the priority under which the framework is running and controlling the jitter of more than one periodic scheduler. Lastly, we show that the feedback controller we have presented in this paper works on other commercial operating systems as well, in this case, Windows XP.

5.1. Experiment1: Step Response: Varying CPU Load in Linux

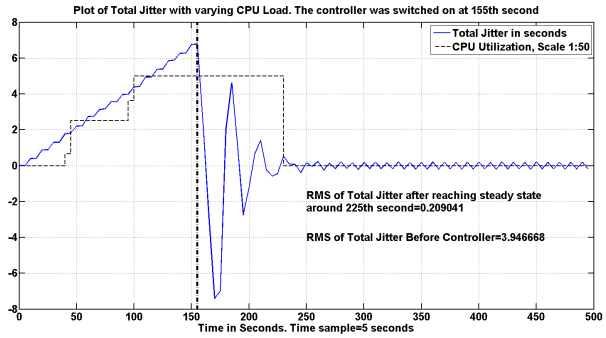

Figure 5. Plot of Total Jitter with varying CPU Load. The controller was switched on at 155th second (dotted line).

Emulating a step response is difficult in an actual operating system since the disturbance depends on a number of

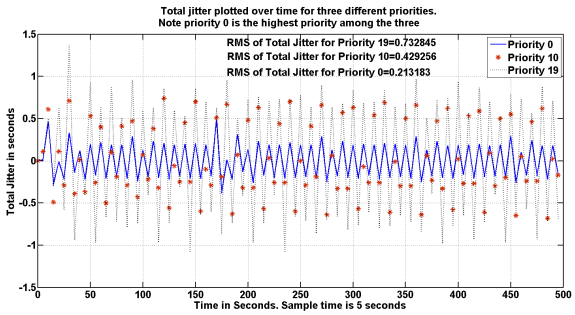

Figure 6. Total jitter plotted over time for three different priorities. Note priority 0 is the highest priority among the three.

parameters such as the time taken by sensors for execution and the scheduling delay, which itself is dependent on the CPU load. The closest we could hope to achieve a step disturbance was to load the CPU in steps to the 100% capacity manually. For this purpose, we used a number of prime number generators freely available over the internet to load the CPU. Furthermore, to emulate the step response, we initially started the framework without the controller, and then started the controller at a predefined time, in this case the 21^{st} iteration of the sensor scheduler.

Figure 5 shows the result of this experiment. Notice that the total jitter is initially rising, and then after the controller is switched on, exhibits a traditional first order step response. In steady state, the RMS value of total jitter was 0.209. This shows that even when the system was heavily loaded, the total jitter was bounded.

5.2. Experiment2: Effect of Current Priority on Total Jitter

The performance of controller also depends on its operating system priority. All processes executing under Linux have a concept of niceness number. This number can be altered by nice and renice commands [14]. A niceness of -20 is the highest priority, while the priority of +19 is the lowest. By default, unless specifically altered, all processes start with a niceness of 0.

Figure 6 plots the total jitter accumulated when scheduler and controller were executed thrice, with three different priorities 0, 10, 19. As expected, even though the control loop stabilized in all three cases, the RMS value of total jitter increased as we lowered the priority. We can attribute this observation to the fact that a lower priority process will have to wait longer for the CPU time, which will increase the disturbance.

5.3. Experiment3: Controlling Jitter of Multiple Periodic Processes

In this experiment, we invoked eight different instances of the sensor scheduler to show that we can compensate the jitter of different applications with different controllers. Each scheduler had its own feedback controller built-in. All interactions between any pair of periodic applications can be attributed to their own random disturbance pattern. Thus, allowing the control loop design to work as it is.

Table 1. Mean, Variance and Root Mean Square values of Total Jitter in seconds for 8 processes referred in figure 7.

Process	Mean	Variance	RMS
1	0.0002	0.0001	0.0116
2	0.0002	0.0001	0.0118
3	0.0002	0.0001	0.0113
4	0.0002	0.0002	0.0127
5	0.0002	0.0002	0.0134
6	0.0003	0.0003	0.0159
7	0.0002	0.0002	0.0155
8	0.0002	0.0002	0.0143

Figure 7 shows the total jitter variation for each of the 8 processes. All processes have a period of 5 seconds. Table 1 provides the mean, variance and RMS value of all the processes. We can notice that all instances have similar mean and RMS value. The implication of this result is the conclusion that different controllers running together did not introduce any undue behavior in the control loop that would degrade the total jitter value.

5.4. Experiment4: Controlling Jitter of a Periodic Application in Windows XP

This experiment highlights the ability of the controller algorithm to work on an operating system other than Linux, in this case Windows XP. Since the sensors we had were specific to Linux, we created pseudo sensors that slept for a random period of time, uniformly distributed between $[0, 1]$ seconds. This helped us to emulate the part of disturbance due to time required for sensor execution. Figure 8 show the result under 100% CPU utilization and regular CPU load (around 10-20%). The 100% load was emulated by using a prime number generator. The system was initially started without the controller to check the transient response. The controller was switched on 95 seconds after the scheduler was started. It can be seen that the response is similar for both load conditions, however, the steady state RMS value of total jitter is more when the system was under a heavier load.

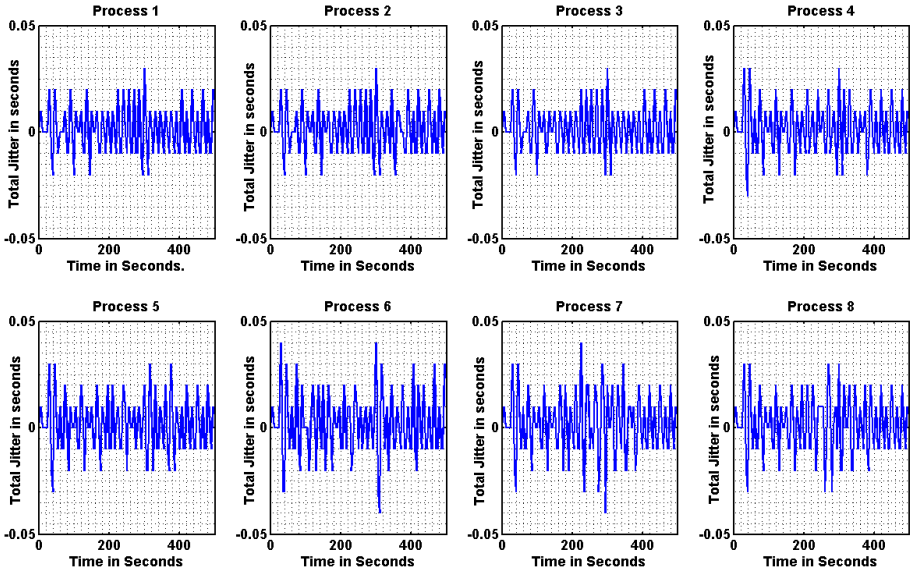

Figure 7. Plot of Total Jitter for 8 different periodic processes with the same time period of 5 seconds. Each process is being controlled by its own controller

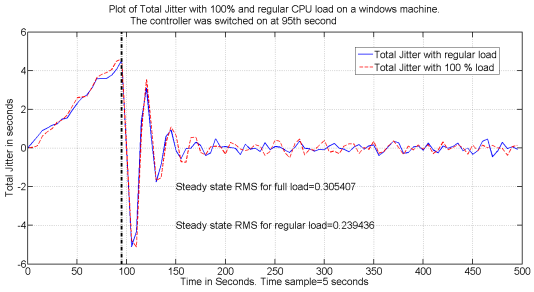

Figure 8. Plot of Total Jitter with 100% and regular CPU load on a windows machine. The controller was switched on at 95th second (dotted line).

6. Related Research

Feedback control based approaches are a common tool for solving a number of engineering problems. A recent roadmap provided by European Network of Excellence ARTIST2 on Embedded System Design focuses on achieving performance and adaptivity in real-time computing systems by using control theory, see [3] and references therein. The roadmap identifies six different research areas including CPU resource control and feedback-based scheduling. Our work in this paper relates closely to feedback scheduling. A state-of-the-art survey is given in [2].

One of the early results in the case for feedback control based scheduling algorithms was presented in [15] by Stankovic et al. In their approach, they used a PID controller to regulate deadline-miss ratio for a set of soft real-time tasks, by using CPU utilization as a control variable. They provided an extended version of their work in [9]. In that, they described their feedback control real-time scheduling framework for adaptive real-time systems and provided general guidelines for designing feedback loop for different quality of service (QoS) parameters. Implementation of this framework requires kernel level modification to an operating system. However, we require a user space solution to the problem to maintain flexibility of choosing any general purpose operating system.

In [13], Sha et al. presented a queuing model based feedback controller to keep the performance of a network server to desired levels. In the same spirit, the authors of [1] provided various models for a web server and designed feedback controller for QoS adaptation. Feedback control theory has also been applied in controlling CPU utilization in real-time systems [19]. Steere et al. introduced a new scheduling scheme based on periodicity of tasks in [16]. Their scheme was to allocate each thread a percentage of CPU cycles over a period, and then use a feedback loop to control both proportion and period.

These approaches depend on the ability to implement an algorithm at the kernel level of the operating system. Our approach presented in this paper uses a generic user space

approach that can be implemented on different operating systems readily.

7. Conclusions and Future Research

In this paper, we presented a feedback controller to compensate for jitter in periodic tasks. From the experiments presented in the paper, we can suggest the generality of the control algorithm. By applying it to either, the periodic application or the scheduler responsible for releasing the application periodically, we can obtain a stable steady state value for total jitter even from a general purpose operating system. During all the experiments we noticed that the overhead induced due to the controller was very small. We have started using this controller in the Fermi Lab LQCD cluster reliability subsystem and have seen considerable jitter improvement.

Our approach does not buffer a ready job and does not hold the CPU resources when the job is completed. Furthermore, its scheduling overhead is more than the simple EDF algorithm as it maintains a history and actively compensates for jitter. We conclude that it satisfies the guidelines set by Baruah et al. in [4] for jitter control algorithms.

In the future, we will extend this work so that we can use a single controller for controlling jitter in multiple periodic applications. Moreover, we are working on extending this work to include model predictive controller. Such controllers will prove effective if we do not have a linear plant model.

8. Acknowledgments

This work was supported in part by DoE SciDAC program under the contract No. DOE DE-FC02-06 ER41442. Sherif Abdelwahed also acknowledges support from the NSF SOD Program, contact number CNS-0613971. Lastly, we will also like to mention our gratitude to Dr Aniruddha Gokhale of Vanderbilt University for his advise and help.

References

[1] T. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96, January 2002.

[2] K.-E. Årzén, B. Bernhardsson, J. Eker, A. Cervin, K. Nilsson, P. Persson, and L. Sha. Integrated control and scheduling. Technical Report ISRN LUTFD2/TFRT--7586--SE, Department of Automatic Control, Lund Institute of Technology, Sweden, aug 1999.

[3] K.-E. Årzén, A. Robertsson, D. Henriksson, M. Johansson, H. Hjalmarsson, and K. H. Johansson. Conclusions of the artist2 roadmap on control of computing systems. *SIGBED Rev.*, 3(3):11–20, 2006.

[4] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari. Scheduling periodic task systems to minimize output jitter. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 62, Washington, DC, USA, 1999. IEEE Computer Society.

[5] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2005.

[6] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[7] A. Dubey, S. Nordstrom, T. Keskinpala, S. Neema, T. Bapty, and G. Karsai. Towards a verifiable real-time, autonomic, fault mitigation framework for large scale real-time systems. *Innovations in Systems and Software Engineering*, 3:33–52, March 2007.

[8] H. Kopetz, M. Holzmann, and W. Elmenreich. A universal smart transducer interface: Ttp/a. *International Journal of Computer System, Science, and Engineering*, 16(2), Mar. 2001.

[9] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms*. *Real-Time Systems*, 23(1-2):85–126, 2002.

[10] P. Martí, J. M. Fuertes, K. Ramamritham, and G. Fohler. Jitter compensation for real-time control systems. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 39, Washington, DC, USA, 2001. IEEE Computer Society.

[11] K. Ogata. *Discrete-time control systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[12] K. Ogata. *Modern control engineering (3rd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[13] L. Sha, X. Liu, Y. Lu, and T. Abdelzaher. Queueing model based network server performance control. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 81–90, Washington, DC, USA, 2002. IEEE Computer Society.

[14] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[15] J. A. Stankovic, C. Lu, and S. H. Son. The case for feedback control real-time scheduling. Technical report, Charlottesville, VA, USA, 1998.

[16] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 145–158, Berkeley, CA, USA, 1999. USENIX Association.

[17] J. Sun and J. Liu. Synchronization protocols in distributed real-time systems. *Distributed Computing Systems, International Conference on*, 0:38, 1996.

[18] R. Thibodeaux. The specification and implementation of a model of computation. Master's thesis, Vanderbilt University, February 2008.

[19] X. Wang, C. Lu, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Trans. Parallel Distrib. Syst.*, 16(6):550–561, 2005.