Increasing the Clock Interrupt Frequency for Better Support of Real-Time Applications

Constantinos Dovrolis Parameswaran Ramanathan
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Madison, WI 53706
{dovrolis,parmesh}@ece.wisc.edu

March 6, 2000

1 Introduction

Applications that process and/or transfer *Continuous Media* (audio and video) streams become increasingly popular. A CM stream can be thought of as a *sequence of packets*, in which each packet conveys some type of audiovisual information that covers a certain time interval. This information is finally 'consumed' when the application's *playback point* reaches the time interval that the packet covers. It is clear that CM streams have real-time constraints on both the network and the OS delays [1]. These constraints are in the form: *if a packet of the CM stream does not reach the consuming application before its playback point, it is considered lost.* We focus on the OS-induced delay between the time that a packet has been generated or arrived at the end host, and the time that it is actually delivered to the end application. This OS delay includes the time that the packet spends in the I/O device that generated it (e.g., network interface, audio card, video card), in the kernel code, while being processed, and in the kernel buffers, as it waits for the receiving application to wake up and consume it. The problem in this process, however, is that in all the current mainstream systems the OS delays can be unpredictably large and highly variable, depending on the load of the system, the specifics of the process scheduler, some related system configuration parameters, and the way the kernel interacts with I/O devices [2, 3].

In the last ten or more years there has been a lot of research on OS's that can offer real-time guarantees (for a recent example, see [4]). It is generally agreed, however, that these ideas did not find their way to the real-world, mainly because they were too complicated and significantly different from the mainstream OS's. For example, most of the real-time OS's were based on process admission control, reservation of CPU resources, and in some cases they were assuming no virtual memory. In this project we took a more 'down-to-earth' approach asking the question: *without major modifications, are there any simple things that we can do to a mainstream OS so that it can better meet the real-time constraints of CM streams?* Specifically, we consider a mainstream, multiprogramming OS, such as a commercial UNIX variant (e.g., Solaris) or Windows NT, that does not offer any real-time guarantees. Another point that is well understood today is that CM streams do not need to perfectly meet their time requirements, i.e., some packets can miss their playback point without noticeable degradation of the quality that the end user perceives [5]. Consequently, the issue that we look at is *how to increase the fraction of packets that meet their real-time requirement, as much as possible.*

The main issue in meeting the real-time constraints of CM streams is the ability of the OS to perform *Synchronous Execution*, i.e., to execute some user or kernel code exactly every X time units, especially when X is in the order of a few milliseconds (say 5 to 20). We identified all the available mechanisms for synchronous execution in a 'typical' Unix system, and analyzed their limitations. It turns out, as discussed in Section 2, that there are three such general mechanisms: *callouts, alarms, and periodic I/O interrupts.* Our analysis shows that the timing accuracy of all three mechanisms depends on the system *clock interrupt period.* It is the clock interrupt that provides to the system the concept of 'time-passing', and it is the clock interrupt period that limits the minimum time between context switching. It is clear, consequently, that a smaller clock interrupt period will lead to more time-accurate synchronous execution. On the other hand, however, it is also clear that a higher clock interrupt frequency will inevitably lead to an overall performance loss. It is mainly due to this reason that the clock interrupt period has remained constant to 10 msec, or more, in most systems in the last 5 to 10 years, despite the fact that the CPU performance doubles every two years.

Looking at this trade-off between improved synchronous execution and overall system performance, we make the case that it is time for the system vendors to reduce the clock interrupt period to 1 msec. Our claim is that this will lead to a significant improvement in the accuracy of synchronous execution, and consequently better support for CM streams, while the performance loss will not be important due to the large increase in CPU speed and power that we experienced in the last years. To support this case, we run a series of experiments on a SUN Ultra-1 workstation running Solaris 2.6. We decreased the clock interrupt period from 10 msec to 1 msec and evaluated both the resulting performance loss, and the improvement in the time accuracy of synchronous execution. The results agree with our original case, but they also illuminate several issues that we originally ignored. The rest of this report is structured as follows: Section 2 provides a brief background on the issue, it analyzes the different mechanisms that can achieve synchronous execution, and explains our case for increasing the clock interrupt frequency. Section 3 evaluates the overhead in increasing the clock interrupt frequency, while Section 4 evaluates the improvement in the timing accuracy of alarms and periodic I/O interrupts after our modifications. We conclude in Section 5.

2 Background, analysis, and our case

We start this section with a brief overview (mainly based on [6, 7]) of some OS components that are related with the rest of this work . Although we use terms that are specific to SUN's Solaris, we believe that most of this material applies to all mainstream OS's.

Clock Interrupt: A periodically generated hardware interrupt that is used by the OS for CPU-usage bookkeeping, context switching, time-of-day updating, processing of callouts, processing of alarms, etc. It is usually the highest priority interrupt (or the second highest, after the power-failure interrupt) and it is never disabled. The period of the clock interrupt in most current UNIX systems is 10 msec.

Callouts: Functions that are executed periodically by the kernel. They are usually associated with system or device-driver activities, and they are available only to kernel code. For example, they are used for retransmissions of network packets, for certain memory-related and scheduler-related functions, for polling devices that do not support interrupts, etc. They are normally assigned a very high system priority, and it is for this reason that their number is normally small and restricted only to kernel code. Callouts are created using the following system calls, defined in the DDI/DKI specification:

```
timeout(kernel_function, args, time_intrv);
delay(number_of_ticks);
```

Alarms: A process can request from the kernel to send it a signal after a certain time interval, much like having an alarm clock. For a real-time alarm, the corresponding signal is SIGALRM. Alarms are set by the BSD system call *setitimer()*:

```
setitimer(ITIMER_REAL, time_intrv, prev_intrv);
```

POSIX 4 provided some high-resolution variants of this system call, called *timer_create()* and *timer_settime()*, but we found that they provide the same accuracy in Solaris 2.6 as the setitimer function. It is important to note that alarms, as well as callouts, are only processed during the clock interrupt handling. Additionally, the fact that an alarm has expired does not necessarily mean that the corresponding process will wake up at once; the actual execution time of the alarm handler by that process depends on the process scheduler and the rest of the system load.

Process Scheduler and Priorities: The kernel component that decides which process/thread runs next. Most Unix systems use a priority-based scheduler in which a process of a certain priority preempts any other process of lower priority. Processes of the same priority are serviced in a round-robin manner, in which each process is given a specific CPU time-slice, called *CPU quantum*. Normally, the priority of a process gradually decreases as it consumes multiple quanta (since it is characterized as a batch job), while the priority of a process that just wakes up after a system call or an I/O operation is increased significantly for a short-term (since it is characterized as an interactive job). The CPU quanta of low-priority processes are normally longer, since these processes were waiting longer to be executed. Solaris extended the BSD scheduler with a set of class-specific schedulers (Real-Time, Time-Sharing, System). In practice, though, only the Time-Sharing class is normally used, since the Real-Time class requires superuser privileges. The priority and CPU quanta calculations are performed on every clock interrupt. As one would expect, the quanta are selected to be multiples of the clock interrupt period, and they are expressed in terms of clock ticks (clock periods), rather than in absolute-time terms. The default quanta values in Solaris 2.6 range from 2 to 20 clock ticks (20 to 200 msec for a 10 msec clock, and from 2 to 20 msec for an 1 msec clock).

Periodic I/O Interrupts: Some I/O devices, such as audio or video cards, generate an interrupt every time they have filled up a buffer of a certain size with audio or video data. For example, the following three statements are used in an IP-telephony application for opening the audio device for recording, for setting up the buffer size per interrupt (record buffer size), and for reading a certain number of bytes (AUDIO_SEGMENT) from it to a user buffer:

```
audio_fd = open("/dev/audio", O_RDONLY, 0);
audio_info.record.buffer_size = AUDIO_SEGMENT;
```

```
ioctl(audio_fd, AUDIO_SETINFO, &audio_info);
read(audio_fd, buffer, AUDIO_SEGMENT);
```

Suppose that the audio recording rate is 8 KBps and the AUDIO_SEGMENT is 80 bytes. Then, the audio controller will generate an interrupt every 10 msec. This interrupt can be used as a source of synchronous execution, circumventing the clock interrupt, alarms, or callouts. This also means that using this mechanism, a program can achieve synchronous execution with a period that is less than the clock interrupt period. It has to be noted, though, that these interrupts are of lower priority than the clock interrupt, and thus, the actual dispatching of the user process may occur considerably later than the generation of the interrupt itself. This *dispatching latency* depends again on the specifics of the process scheduler, and the CPU quanta.

Analysis: There are three general mechanisms to achieve synchronous execution: Callouts, Alarms, and Periodic I/O Interrupts (also referred to, in the rest of this report, as just 'interrupts'). There are several differences between these three mechanisms in terms of their availability, time-granularity, accuracy, and overhead. Specifically, callouts are the most accurate way to achieve synchronous execution, but they can only be used for kernel code. Their minimum time-granularity is limited by the clock interrupt period. We do not pursue the study of callouts further here, since we do not have access to the DDI/DKI library and, besides, we are mainly interested in synchronous execution at the user level. It has to be emphasized, however, that a reduction of the clock interrupt period to 1 msec will also reduce the time-granularity of callouts to 1 msec, which is of course desirable. Alarms, on the other hand, do not provide as accurate synchronous execution as callouts, but they are the only way for a user process to execute arbitrary periodic functions. Their time granularity is also limited by the clock interrupt period, while their accuracy greatly depends on the priority of the corresponding process, on the presence of other higher (or equal) priority running processes, and on their quanta. One way to achieve better accuracy in the alarm period is to increase the priority of the corresponding process. This can be achieved, in Solaris for example, using the Real-Time scheduling class (which always preempts the Time-Sharing class), or using the *priocntl* system call. We do not pursue this solution further here, since it is based on the constraint of superuser privileges which is generally a policy issue. On the contrary, we focus on the non-policy, mechanism-related issues, and specifically on the impact of the clock interrupt period and of the CPU quanta. The third mechanism for synchronous execution is the periodic interrupts. They are normally quite more accurate than alarms, because the priority of the process that is awaken by an interrupt is boosted by the OS, but not as accurate as callouts, because these interrupts are not of as high priority as the clock interrupt. Their time-granularity does not depend on the clock interrupt period, but they introduce additional overhead to the system, since they generate frequent independent interrupts. Another problem with this mechanism, is that they do not provide a general solution for the synchronous execution problem, i.e., they can only be used for reading audio/video data from the corresponding devices. This may be sufficient for several multimedia applications, but it certainly does not cover all the real-time requirements of CM applications.

A case for 1 KHz clock interrupt frequency: Based on the above analysis it is clear that a certain benefit in the accuracy of alarms can be expected if we increase the clock interrupt frequency. It should also be clear that this benefit will be larger if we also decrease the CPU quanta, since this change will reduce the dispatching latency occurring when a process waits for another process of equal or higher priority to consume its quantum. The increase of the clock interrupt period and the reduction of the quanta may have a positive impact on the accuracy of the periodic interrupts, since the backlog of system/user activity, that may interfere with the dispatching of the I/O interrupts and their delivery to the user process, will be processed in a more timely manner. Additionally, a higher clock interrupt frequency will lead to smaller time-granularity for both callouts and alarms. This is an obvious reason for increasing the clock interrupt frequency, but not the main reason that we investigate in this project. Specifically, our target-question is: even if we still want to achieve a period of 10 msec, how much better accuracy do we get with a clock interrupt of 1 msec, than with a clock interrupt of 10 msec? Before we evaluate the improved timing accuracy, however, we need to evaluate first the overhead in the overall system performance if the clock interrupt frequency is increased to 1 KHz.

3 The overhead of the clock interrupt

It is important to understand the two different sources of overhead that result from increasing the clock interrupt frequency. First, there is the overhead of executing more often the clock interrupt handler. Specifically, if the clock interrupt handler execution takes v microseconds and we decrease the clock interrupt period from p_1 microseconds to p_2 microseconds, it is easy to calculate that the execution time of a CPU intensive program (in the absence of other system or user load) will be increased by a factor:

$$\text{Overhead} = \frac{1 - \frac{v}{p_1}}{1 - \frac{v}{p_2}}$$

For $p_1 = 10000$ (10 msec), $p_2 = 1000$ (1 msec), and $v = 100\mu$s, we would observe that a CPU intensive program would require 10% more time. We performed this experiment using a simple CPU intensive benchmark that we wrote, while the workstation was otherwise idle (at least from user load). The execution time overhead that was measured over several runs was consistently in the range 2 to 3 %, which for most people would be certainly not important. Using the previous equation, we can calculate that the clock interrupt handler execution time in Solaris 2.6 is about 30 μs, which is probably a relatively long time for an interrupt handler. We performed similar measurements using other benchmarks, either written from us, or from the lmbench suit. The general conclusion is that the overhead of the clock interrupt becomes measurable only in CPU-intensive programs. The reason is that if the program spends a significant amount of time waiting for I/O, the more frequent clock interrupts do not preempt its execution in anyway.

Another source of performance overhead due to the increase of the clock interrupt frequency is associated with context switching. Specifically, since a clock interrupt frequency increase is followed by a reduction of the CPU quanta, there will be more frequent context switches. Context switches on the other hand pollute the caches, reducing significantly the memory system performance. Other pollution effects occur in the out-of-order execution tables, the branch predictor tables etc. We attempted to evaluate this type of overhead, using combinations of multiple running processes under different values of the clock interrupt period and of the CPU quanta. Unfortunately, however, we were not able to get repeatable and consistent measurements. Although we were able to measure in some cases a significant performance overhead with the higher clock interrupt frequency (in the order of 20-30%), we were not able to consistently repeat these measurements and it is not clear yet what fraction of them is due to the increased clock interrupt frequency.

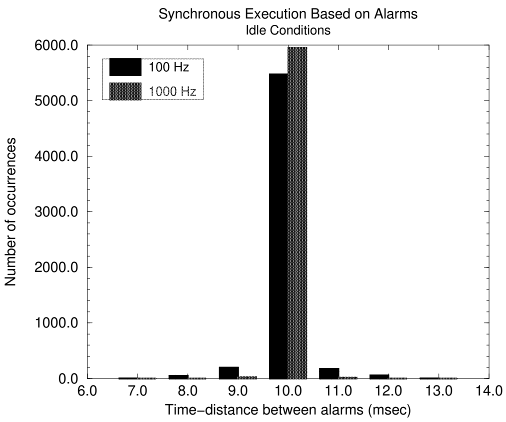
4 Improvement in the timing accuracy of alarms and interrupts

The methodology that we followed for measuring the improvement in the timing accuracy of alarms and interrupts due to the higher clock interrupt frequency is as follows: in the case of alarms, we wrote a simple program that sets a real-time timer with a period of 10 msec. We then measured the interarrivals (time-distance) between the actual execution of our SIGALRM signal handler, for a sequence of 6000 alarms (which is one minute). We performed this experiment for a clock frequency of both 100 Hz and 1 KHz. In addition, the experiments were repeated under a variety of system loads: 'Idle Conditions' (i.e., no user activity), 'Busy GUI' (i.e., random activity in the GUI, such as opening and closing windows, changing screens, reading e-mail, etc), and three runs with one, two, and four CPU-intensive processes running at the background, respectively. It has to be noted that our measurements were repeatable and consistent, even for the case of the 'Busy GUI'.

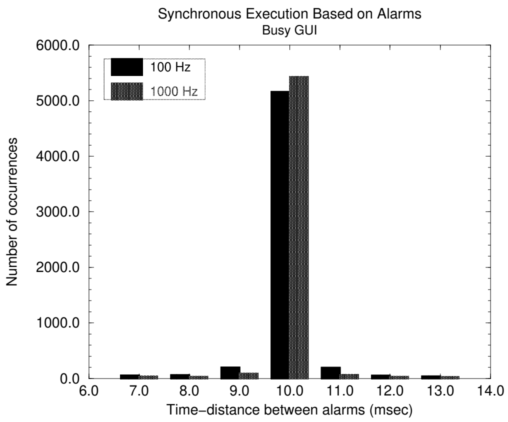
The methodology for the case of periodic I/O interrupts was basically the same. The main difference is that we used an Internet telephony application that we had written before. This application, and specifically the sending part of it, reads audio data every 10 msec from the audio controller, using the code segment that we showed in Section 2. We added some trace code in the application which allowed us to measure the time between successive executions of the application code after an audio interrupt. Again, the measurements cover an interval of 6000 events (or 1 minute). All the timing measurements have been done using the *clock_gettime()* system call, which has a resolution of 1 μsec.

Figure 1 shows the results for the case of alarms. For the case of idle or busy-GUI conditions, most of the alarms occur every 10 msec, while their timing accuracy is clearly improved with the clock interrupt of 1 KHz. In the case of running processes in the background, however, decreasing the clock interrupt period alone does not improve the alarm accuracy. In fact, there are less accurate alarms in that case, compared to the 10 msec clock interrupt. The reason is that with a lower clock interrupt period, the interarrivals of alarms are getting more dispersed in the time axis, especially when the alarm-requesting process has to wait for another process to consume its quantum. It is interesting that the timing accuracy of alarms with a clock interrupt of 1 KHz does not really appear to be any better than with a clock interrupt of 100 Hz. When we also reduce the CPU quanta, however, to be either 2 or 4 clock interrupt periods (as opposed of being between 2 and 10 clock interrupt periods), we get a higher fraction of alarms concentrated around the 10 msec region. Perhaps the most important point in these graphs is the right-most bars, where we show the number of alarms that occurred more than 35 msec apart. Those alarms can be clearly characterized as 'late' alarms. The number of late alarms for the case of 1 KHz clock interrupt and reduced CPU quanta is almost zero, while almost all alarms occur less than 25 msec apart.

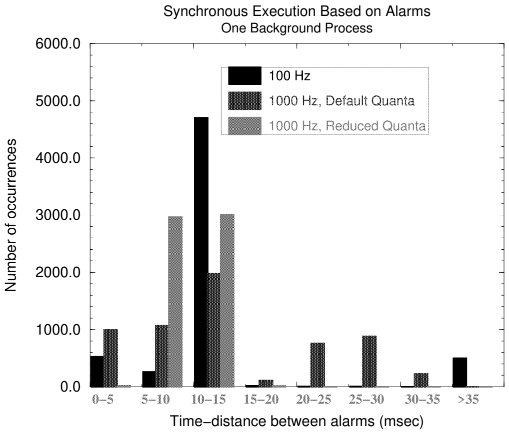
Figure 2 shows the timing distance between successive application executions due to the periodic audio interrupt. It can be noted that the timing accuracy is much better than that of alarms, and it does not depend on the CPU quanta. The reason for the latter observation is that our application gets the highest priority when it is awaken from an interrupt, and so it preempts any other user process. Looking at the entire graph (and not only in the bar that corresponds to the range 10 to 15 msec), we can see that the higher clock interrupt frequency of 1 KHz increases the fraction of interrupts that occur about 10 msec apart. Additionally, as



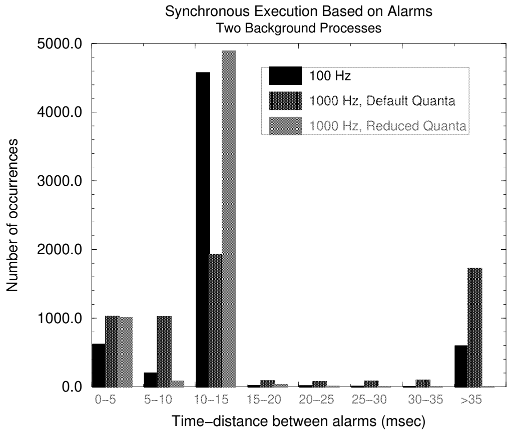
(a) Idle Conditions



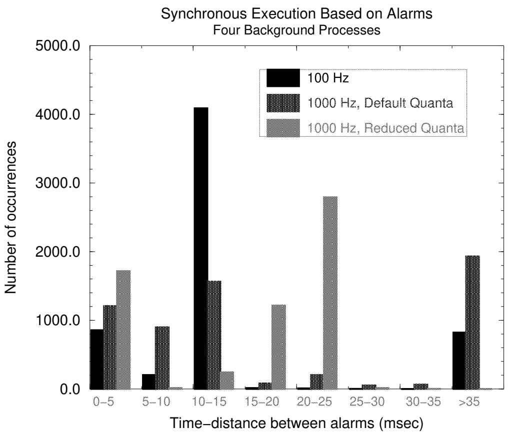
(b) Busy GUI



(c) One background process



(d) Two background processes



(e) Four background processes

Figure 1: The accuracy of alarms under different system loads.

in the case of alarms, the number of 'late' interrupts, i.e., interrupts that occur more than 35 milliseconds apart, with the 1 KHz clock, is almost zero. The reason for this improvement is that with a higher frequency clock interrupt the backlog of system activities gets processed in a more timely manner, reducing the number of times that the interrupt-receiving user process will have to wait because of higher-priority, system activity.

5 Conclusions

Our main finding is that the increase of the clock interrupt frequency to 1 KHz will mainly improve the timing accuracy of alarms, and secondly, of callouts and periodic I/O interrupts. In addition to the clock interrupt period, though, the CPU quanta would also have to be reduced significantly, so that even the longest quantum is only a few multiples of the clock interrupt period. The overhead of these modifications does not seem to be important (2 to 3 %), but some more work is needed on this issue, especially on the cache-pollution effects that may be caused by the more often context switches that will come with a higher clock interrupt frequency.

References

[1] R.Steinmetz and K.Nahrstedt, *Multimedia: Computing, Communications, and Applications.* Prentice Hall, 1995.

[2] J.Nieh, J.G.Hanko, J.D.Northcutt, and D.Wall, "SVR4 UNIX scheduler unacceptable for multimedia applications," in *Proceedings Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Nov. 1993.

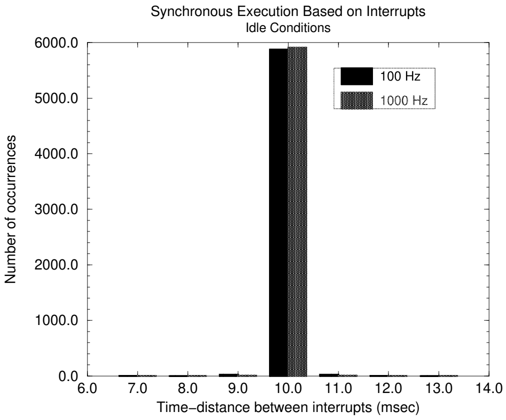
[3] M.B.Jones and J.Regehr, "Issues in using commodity operating systems for time-dependent tasks: Experiences from a study of windows nt," in *Proceedings IEEE Networks and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 1999.

[4] D.K.Y.Yau and S.S.Lam, "Adaptive rate-controlled scheduling for multimedia applications," *IEEE/ACM Transactions on Networking*, vol. 5, pp. 475–488, Aug. 1997.

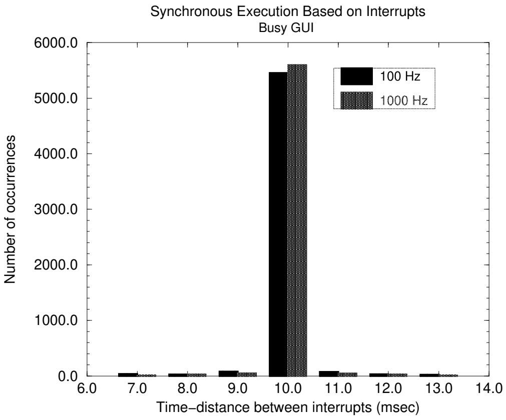
[5] H.Schulzrinne, S.Casner, R.Frederick, and V.Jacobson, *RTP: A Treansport Protocol for Real-Time Applications*, Jan. 1996. RFC 1889.

[6] U. Vahalia, *UNIX Internals, the new frontiers.* Prentice Hall, 1996.

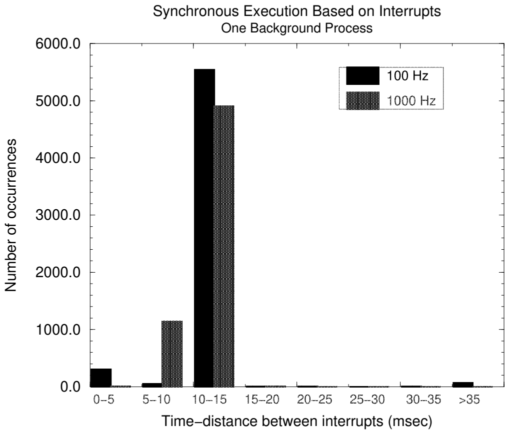
[7] B.Goodheart and J.Cox, *The Magic Garden Explained.* Prentice Hall, 1994.



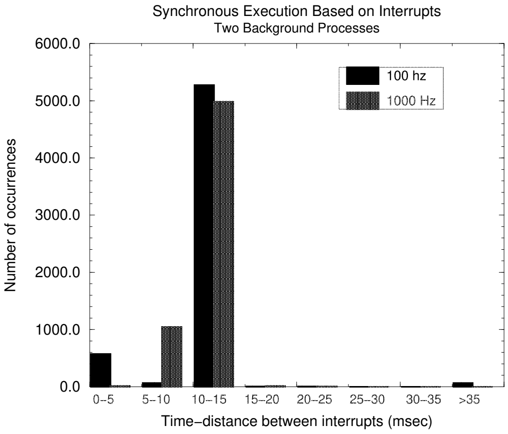
(a) Idle Conditions



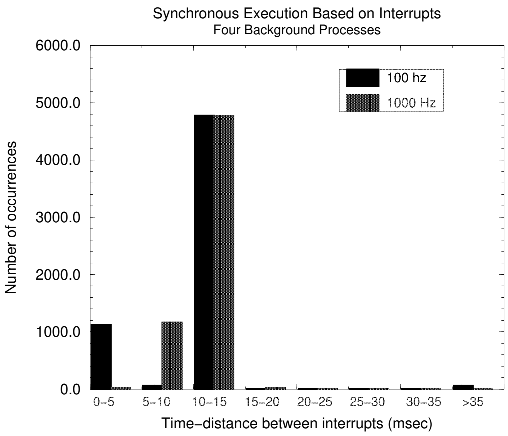
(b) Busy GUI



(c) One background process



(d) Two background processes



(e) Four background processes

Figure 2: The accuracy of periodic I/O interrupts under different system loads.