MEASURING PERFORMANCE IN REAL-TIME LINUX

Frederick M. Proctor
National Institute of Standards and Technology
100 Bureau Drive, Stop 8230, Gaithersburg, MD 20899-8230 USA
frederick.proctor@nist.gov

Abstract

There are many measures of software performance, split between size, speed, and resource use. Linux measures up quite well with these metrics: the kernel is scaleable and small to begin with, and it boots up quickly with a minimum of disk activity. More detailed measures can be made using benchmarks such as those from SPEC [1]. Real-time programmers, however, are more concerned with timing performance, such as the maximum time to service interrupts or the variability in periodic task scheduling. Various methods of measuring this timing variation exist. Measurements on the RTL and RTAI variants of real-time Linux show values on the order of several μsec to tens of μsec. No matter how well-written, interrupt service routines and schedulers will eventually run up against timing uncertainties in the underlying hardware due to features such as caches, pipelines, and speculative execution. Techniques to minimize these effects for periodic scheduling can reduce the variation down to below a tenth of a microsecond, for tasks that run at periods of a few tens of microseconds. This analysis and experimental results show that real-time Linux is suitable for fairly aggressive real-time tasks.

1 Introduction

Performance measures are figures of merit that indicate how well a system behaves. Traditional operating system performance measures include size (both in memory and on disk), speed, and resource usage (e.g., disk and network access). Qualitative experience with Linux shows that it measures up quite well against these general metrics, compared with other operating systems: the kernel is modular, scaleable, and small to begin with; it boots up quickly and with a minimum of disk activity; the buffer cache limits access to the disk during normal use; and the network stack is highly optimized. Quantitative measurements can be made using benchmark test software, such as those from SPEC [1]. These benchmarks test mainly hardware performance, although the operating system layer is indirectly included and is directly tested with file system and process benchmarks. Real-time programmers, however, are more concerned with an operating system's time-related performance than with its average performance. Time-related performance metrics include variation and worst-case values of interrupt service routine (ISR) latency and scheduling periodicity. Techniques exist to measure these, and can be adapted to improve performance in some cases.

2 Measurement Techniques

There are a number of ways that performance measures can be gathered and interpreted. Two broad categories of tests can be performed: external tests, in which a measurement system outside the platform captures the measurement data; and internal tests, in which software running on the platform itself does the data capture. External tests have the advantages that the entire platform is tested, and the test equipment serves as an experimental control. The drawback is that test equipment is an additional cost. Internal tests have the advantage that no additional equipment is needed, so costs are low and tests can be easily duplicated. Internal tests can also be in-

[1] No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied. Certain commercial equipment, instruments, or materials are identified in this report in order to facilitate understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose. This publication was prepared by United States Government employees as part of their official duties and is, therefore, a work of the U.S. Government and not subject to copyright.

cluded in the final deployed system, allowing periodic or continual monitoring in the field. Another benefit of internal testing is that it can be adapted to correct timing variation. The drawback is that the system being tested is also doing the testing, raising the possibility of "cheating". For example, if the source of timing variation affects measurement in the same way, the variation may be invisible. In this work, internal tests were conducted. External testing methods are described in [2].

3 Environment

Measuring the variance and worst-case values of the ISR latency and scheduling periodicity is difficult in general since it requires that all the factors contributing to these metrics are present during the test in their worst-case combination. Certainly the tests should be conducted in an environment that closely matches the expected environment of the deployed system, especially with respect to likely sources of timing perturbation such as disk or network access. This is not an intrinsic feature of the tests, but a guideline on how the tests should be administered. In this work, tests were conducted in environments with low to high disk and network loading, to demonstrate the effects of these environmental factors on performance.

The RTOSes chosen for testing are Real-Time Linux (RTL) from New Mexico Tech [3], and Real-Time Application Interface (RTAI) from Milan Polytechnic [4], although the techniques are independent of the operating system.

4 ISR Latency

An interrupt is a notification that an external event needs attention. The time between the occurence of the interrupt and its servicing by the ISR is the *ISR latency*. Both hardware and software factors influence ISR latency. Hardware factors arise from the processor's need to finish the current instruction before saving context and switching to the ISR. In a general-purpose processor, instruction lengths vary from a single clock cycle to tens of clock cycles. An interrupt unfortunate enough to occur at the beginning of a long instruction will suffer higher latency than one occuring during a single-clock instruction. Software factors include interrupt masking and priority, in which the servicing of low-priority interrupts is delayed when a higher-priority interrupt is being serviced. Low-priority interrupts occuring during the execution of a high-priority ISR will suffer higher latency than those occuring in isolation. Because interrupts occur unpredictably, it is impossible to characterize the exact timing behavior of a real-time system in their presence. The behavior can only be characterized statistically.

Two figures of merit that apply are the variance (or standard deviation), and the worst-case value. A common technique for measuring these is to write an ISR that generates a measurable output as soon as it executes. This may be done by setting an output bit tied to an external test and measurement system triggered by the interrupt line [5]. If a digital storage oscilloscope is used and repeated interrupts are displayed in persistent mode, the results will show the occurence of the interrupt, a blank region, then progressively more transitions to a high output until a solid high region. The worst-case latency can be directly observed by noting the time between the triggering interrupt and the onset of the solid high region. Determining the variance requires either equipment that computes this itself, or by offloading the stored data for off-line analysis.

ISR latency is not measurable using internal tests, since the time the interrupt occurs will not be captured in software until the associated ISR executes. The first possible measurement can only be made after the ISR latency has passed.

5 Scheduling Jitter

Scheduling jitter is the variation in the intervals between cycles of a periodic task. High jitter is undesirable, but may still satisfy deadline requirements. Jitter high enough to cause tasks to execute after their scheduling deadline is unacceptable. The degree of variation in the task timing, and the worst-case value, are the figures of merit.

One technique to measure jitter is to run a periodic task and measure the actual periods seen by the task. Measurements can be made by external hardware, as was described for ISR latency, or through internal methods using built-in time stamp mechanisms accessible in software [6]. The details of this technique are presented in Section 7. Software techniques are attractive since they are effectively free, although in some cases they cannot discern jitter common to both the task and processor clocks. This is discussed in more detail in Section 7.1.

6 Hardware Effects

No matter how well-written the RTOS, however, hardware effects will become noticeable as interrupts occur more frequently and task deadlines shorten. This is particularly true for hardware platforms

based on general-purpose processors like the Pentium. These processors include features such as memory management units, caches, pipelines, speculative execution, and various system buses that each introduce unpredictability in the time it will take code to execute. The variation is small but noticeable for high-frequency applications. Analytic techniques for predicting the effect on real-time programs exist [7], applicable to programs for which the instructions are known. In general, all the instructions for all the code running on the processor must be included in the analysis. Although this is possible in an open-source operating system like Linux, is is impractical, hence the need for empirical measurement techniques.

Timing variation is less of a problem with platforms such as digital signal processors (DSPs), which forgo optimizations such as caches and pipelines in exchange for determinism. The loss of these optimizations is made up for with an instruction set tailored for high-speed execution of common instructions such as multiply-and-accumulate. However, general-purpose processor platforms remain attractive for real-time applications, particular the IBM PC-compatible, due to their low cost, prevalence, scaleability, available peripherals and input/output boards, and large number of real-time operating systems that support them. It is therefore natural to ask, "How bad is the timing problem?", a question for which performance measures can provide an answer.

6.1 Cache Effects

The cache is one source of timing unpredictability that merits attention due to its significance. The cache is a small section of local memory on the processor that contains copies of frequently-accessed data from external main memory. Cache access is extremely fast due to its locality, much faster than accessing memory off the processor through its external buses. Once data has been loaded into the cache from memory, subsequent accesses are made to the cache. Occasionally, however, cached data will be replaced to make room for more recent data, and the next time the replaced data is accessed it must be reloaded, causing the code to take more time to execute.

The problem with the cache it that it is a shared resource that cannot be locked down to particular tasks. In a multiprogramming operating system like Linux, it is possible (and an eventual certainty) that a non-real-time program will run and cause some real-time code to be removed from the cache. Surprisingly, for a given real-time task period, running the code on a faster processor actually increases the chance real-time code will be removed from the cache. This is because in the real-time task's idle period, more non-real-time code can run, replacing more of the cache.

One might expect that disabling the processor's cache would reduce jitter. However, other features such as instruction pipelining are part of the processor architecture and cannot be disabled, and the absence of the cache magnifies their contribution to jitter. Even if jitter were reduced, the speed penalty may be intolerable. A test of the BYTE benchmark code [8] on a Pentium PC showed a twenty-fold decrease in speed with the cache disabled.

7 Internal Jitter Measurement

The internal measurement technique uses a simple periodic task that logs time stamps of its invocation into memory for later analysis. Time stamps are measured using the Pentium's internal Time Stamp Counter (TSC), a 64-bit integer that increments once each clock cycle, using the RDTSC instruction [9]. For a 400-megahertz clock, the resolution of the TSC is 2.5 nanoseconds, and it will take more than 1200 years to wrap around.

The task is scheduled by the 8254 Programmable Interval Timer (PIT) interrupt used in both RTL and RTAI. The task is run in pure periodic mode, not one-shot mode, to reduce any influence from reprogramming the PIT. Ideally the logged values should differ by exactly the interrupt period, but variations in the combined execution time of the interrupt service routine, scheduler code, context switches, and task code prior to the RDTSC call will show up as jitter in the analysis.

7.1 Clock Issues

The relationship between clocks is important when making timing measurements. Two logical clocks exist: the clock that drives the processor and its TSC, and the clock that drives the PIT that generates the scheduling interrupt. The connection between the two clocks may by synchronous or asynchronous. If they are synchronous, one clock may be derived from the other. For example, a single quartz oscillator may drive the processor at 400 megahertz, and be divided down to a few megahertz to drive the PIT. In this case, jitter on the quartz oscillator will be invisible: a clock slowdown will slow both the processor and PIT in equal proportions.

If the clocks are asynchronous, they cannot be derived from one another. For example, a quartz oscillator may drive the processor at 400 megahertz, and a second quartz oscillator may drive the PIT chip at

a much lower frequency. In this case, jitter on one clock will not affect the timing of the components driven by the other. This is the benefit of external measurements, in which a presumably low-jitter data acquisition system can detect jitter caused by both cache effects and the underlying processor clock.
Typical PC-compatible computers have a single quartz oscillator that drives both the processor and the PIT, with the PIT functions integrated into the motherboard's South Bridge chipset. Thus the two are synchronous, and the contribution of clock jitter to overall jitter will be undetectable. The magnitude of the underlying clock jitter is not appreciable compared to jitter induced by processor and software effects, typically on the order of a few nanoseconds [10].

7.2 Analysis

The captured TSC values are an increasing sequence of integers in units of processor cycles. Maximum jitter J_{max} can be estimated from this sequence in two ways. The simplest is to take the difference of adjacent values, which is the interval between successive task cycles. These differences should be equal to the interrupt period (adjusted for units), but will vary. There will be a smallest difference and largest difference, corresponding to the shortest time between cycles and the longest time between cycles. J_{max} is the difference between the shortest and largest interval. This is the *cycle-to-cycle* jitter.
A plot of cycle-to-cycle jitter is shown in Figure 1. This plot shows 100 differences of the time stamps for a task nominally scheduled at 500 μsecs. Note that large values are followed by small values, a consequence of late cycles lengthening the interval to their predecessor and shortening the interval to their successor. This "mirroring" doubles the effect a single late task cycle has on J_{max}. In this case, if a single cycle is late by 10 μsecs, the interval before will be 510 μsecs, the interval after will be 490 μsecs, and J_{max} will be 20 μsecs. This effect is detailed in [11]. J_{max} for the full data set (10,000 points) using this method is 5.66 μsec. The data was taken in single-user mode, in which many fewer Linux processes are running than in multi-user mode.
In the second method, the differences between each TSC value and its expected nominal value are computed. The nominal values are not known, but are estimated as lying on the least-squares best fit line to the TSC sequence. The difference between each TSC value and the best-fit line is each cycle's jitter, and J_{max} is computed as the difference between the maximum and minimum cycle jitter values. This is the *period jitter*. Period jitter analysis eliminates the double penalty incurred in cycle-to-cycle analysis, since the neighbors of a single late cycle are only penalized relative to the overall best fit line, not their shared late neighbor. A comparison of the two methods is detailed in [12].
A plot of period jitter is shown in Figure 2. This plot shows the deviation of 100 points from the best-fit line, using the same logged TSC values as for Figure 1. Note that the mirroring has disappeared. J_{max} for the full data set (10,000 points) using this method is 3.60 μsec, about half that computed from cycle-to-cycle analysis.

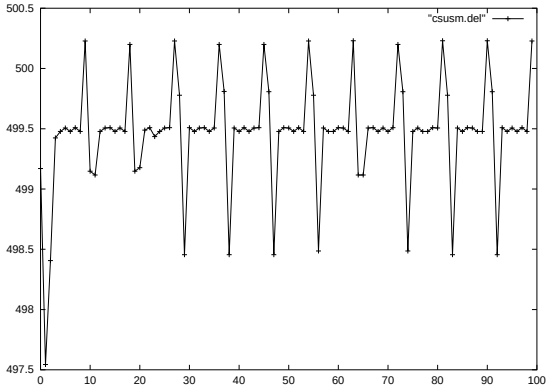

FIGURE 1: *Cycle-to-cycle jitter in μsecs for 100 samples of the TSC on a task scheduled at 500 μsecs. Note the "mirroring" effect, in which high values are followed by low values. This data was taken in single-user mode.*

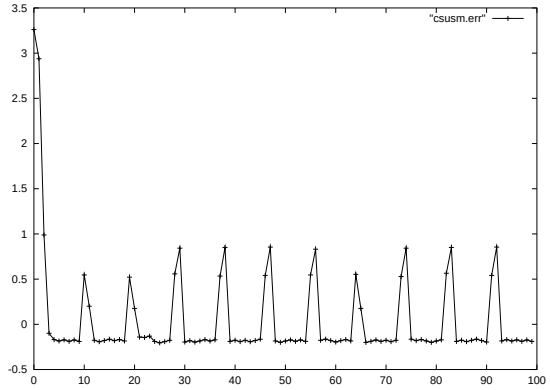

FIGURE 2: *Period jitter in μsecs for 100 samples of the TSC on a task scheduled at 500 μsecs. This plot uses the same source data as for Figure 1, analyzed using a best-fit method. This analysis more clearly reflects the underlying timing.*

Histograms of the period jitter data reveal that jitter values fall into well-defined regions. These regions are the same regardless of the period of the task, as shown in Figure 3. This figure shows several histograms for tests at 50 μsec, 100 μsec, and

200 μsec periods. Note that the histogram peaks appear at the same locations, indicating a common origin for the jitter. Candidates include the scheduler and task cache access patterns, the effect on the cache from other tasks that execute during the logging, and the variation in instruction length between different branches of the scheduler code. Histograms make more clear the clustering of jitter into groups, although the patterns between adjacent samples evident in Figure 2 are lost.

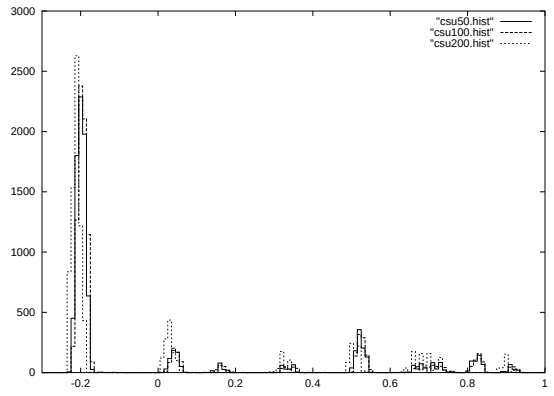

FIGURE 3: *Histogram of period jitter. Three tests are shown superimposed, at 50 μsec, 100 μsec, and 200 μsec periods. The colocation of the peaks indicate a common origin for the jitter.*

8 Environment and Jitter

Although the measurement task runs in real time and is not interrupted by user-level Linux processes, these processes will indirectly affect the real-time task timing through the cache. The previous data was collected in single-user mode, in which most services were disabled. It is natural to assume that the more user-level Linux tasks that run, the more the real-time task will experience cache-related delays. Tests were conducted with both RTL and RTAI in various environments: normal loading, heavy disk loading, and heavy network loading. The results are shown in Figures 4 and 5, for RTAI and RTL, respectively.

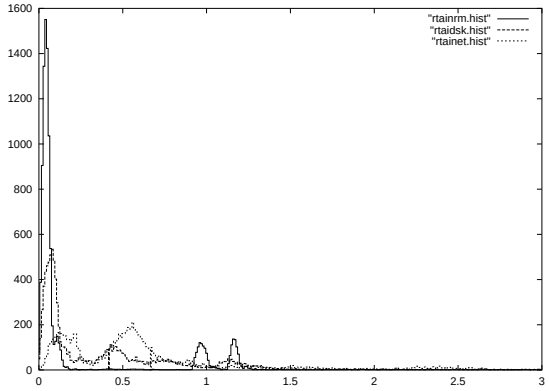

FIGURE 4: *Histograms of RTAI period jitter in various multi-user environments: normal loading, disk loading, and network loading. Note the much wider variation than in Figure 3, due to the many more Linux processes running.*

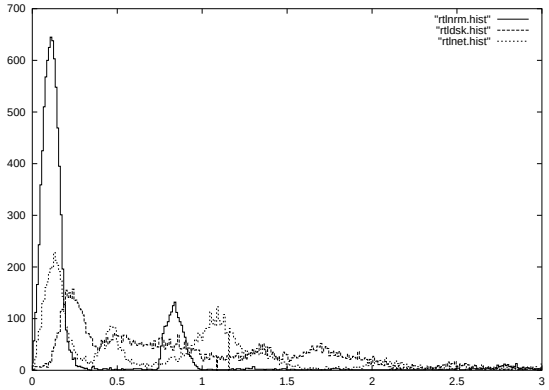

FIGURE 5: *Histograms of RTL period jitter in various environments: normal loading, disk loading, and network loading.*

9 Jitter Compensation

Up to this point the discussion described performance measures and techniques for making them, with a focus on scheduling jitter. It is possible to adapt the software techniques to reduce scheduling jitter, at the expense of ISR latency. If no interrupts are assigned to real-time ISRs, this tradeoff may be worthwhile.

The objective of jitter reduction is to adjust a periodic task so that its time-critical code will execute at a period T, as closely as possible with a minimum of jitter. To accomplish this, the task is adjusted by inserting code prior to the time-critical portion that polls the TSC with interrupts off until a target time stamp is achieved. The time-critical code executes immediately afterward. Clearly the target

time stamps must differ by exactly T. The problem is how to compute the first target time stamp.

On the first occurrence of the task, the time stamp counter is read as the base value. It is impossible to tell how hardware unpredictabilities have affected the first cycle, since no previous time stamps exist for comparison. It may be early, that is, with a minimum of time between the scheduling interrupt and the task code, or it may be late, that is, with maximum of time between the interrupt and task code. Since this is the first time the code has executed, it is likely to be late since it was not yet cached. One can't be sure how late, however.

Figure 6 shows a series of periodically scheduled task cycles tied to an interrupt. The interrupt occurs with a period T and itself is not a source of appreciable jitter, as discussed in Section 7.1. The white portion indicates the scheduler, and the task code is shown in gray. Depending on jitter induced by processor effects, however, the time it takes for the scheduler to complete will vary.

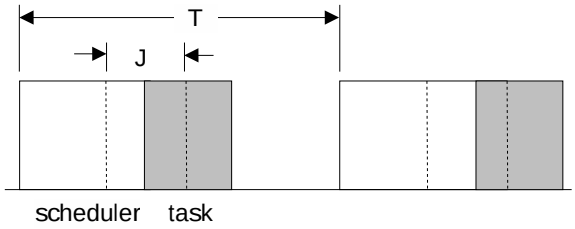

FIGURE 6: *Task scheduling jitter. T indicates the nominal period of the scheduled task. The white region is the time taken by the scheduler ISR, and the gray region is time taken by the task. The dashed boundaries of the region indicated by J show the jitter range. The task code may execute as early as the left dashed boundary, or as late as the right dashed boundary.*

Consider the case where the first cycle is late. If the following cycle is early, it will occur as little as $T - J_{max}$ seconds after the first. Almost J_{max} time will be spent polling. If the following cycle is late, it will occur as much as T seconds after the first. Almost no time will be spent polling. These conditions are shown in Figure 7.

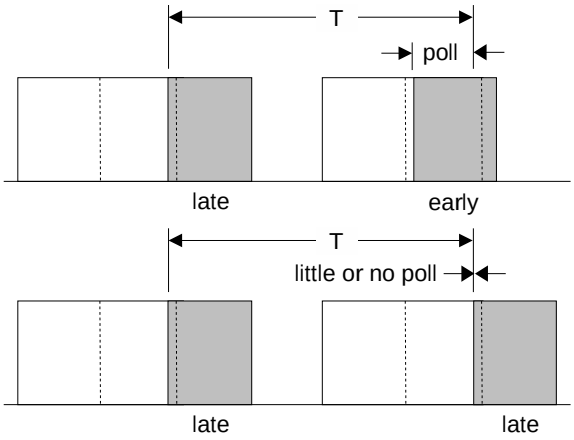

FIGURE 7: *A late first cycle followed by cycles in the extreme early and late jitter range. If the second cycle is early, almost J_{max} time will be taken up by polling. If the second cycle is late, almost no time will be taken up by polling.*

Now consider the case where the first cycle is early. If the following cycle is also early, it will occur as little as T seconds after the first. Almost no time will be spent polling. If the following cycle is late, it will occur as much as $T + J_{max}$ seconds after the first. The target time stamp will have been missed by as much as J_{max}, past the point where polling can be done. These conditions are shown in Figure 8.

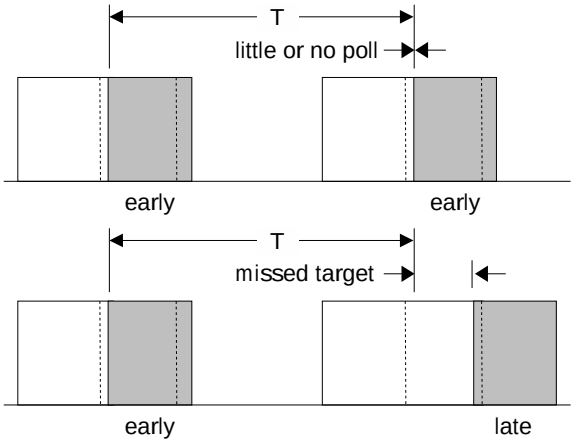

FIGURE 8: *An early first cycle followed by cycles in the extreme early and late ranges. If the second cycle is early, little time will be taken up by polling. If the second cycle is late, the target will be missed.*

The problem can be solved by deferring the time-critical task code in the first cycle, and running a "dummy" cycle that just establishes the TSC reading as a baseline B and computes the next TSC target for the following "real" cycle. This target TSC

will be $B + T + J_{max}$. Subsequent target TSCs will differ by exactly T.

The amount spent polling depends on the initial baseline B. If the first (baseline) task was early, then the ensuing target TSCs will come earlier, and less time will be spent polling. The maximum time will be J_{max}, which occurs for early task cycles. The minimum time will be 0, which occurs for late task cycles. Normally the task code executes early from cache, as shown in the jitter plots, so the expected polling time is slightly less than J_{max}. The additional processor load is therefore J_{max}/T. For a nominal 500-μsec period T and a worst-case J_{max} of 5 μsec, this is a 1% additional load.

However, it is likely that the baseline task was late, since it was not yet cached. If it was late, then the ensuing target TSCs will come later, and more time will be spent polling. The maximum time will be $2J_{max}$, which occurs for early task cycles. The minimum time will be J_{max}, which occurs for late task cycles. Normally the task code executes early from cache, so the expected polling time is slightly less than $2J_{max}$. The additional processor load is therefore $2J_{max}/T$. For a nominal 500-μsec period T and a worst-case J_{max} of 5 μsec, this is a 2% additional load.

To minimize the time spent polling, it is worthwhile to ensure that the baseline task cycle executes early. This can be be made more likely by running several additional dummy cycles to cache up the code. It is not a guarantee, since between the last dummy cycle and the baseline cycle, another process can dirty up the cache.

9.1 Clock Drift

The jitter compensation technique just described will fail if the processor clock and interrupt clock are asynchronous. Over time, the scheduling interrupt will come earlier or later than the target TSC. If drift is such that the task is scheduled earlier and earlier, more and more time will be spent polling, up to the entire period T. All tasks will starve for want of processor cycles hoarded by the polling. If drift is such that the task is scheduled later and later, less and less time will be spent polling, until the target TSC is missed.

Drift can occur even if the two clocks are synchronous, if the conversion between the units for scheduling period and time stamps is not exactly known. From the earlier analysis, each successive target time stamp is increased by the scheduling period T, as programmed into the PIT. However, any scheduling period will be quantized to the resolution of the PIT, whose frequency is 1,193,180 s^{-1}. In general, this quantization will not correspond to an integer number of processor cycles. Since the conversion of T into time stamps will be inexact, the scheduling interrupts and target TSC values will drift.

9.2 Jitter Compensation with Clock Drift

The analysis in Section 9 can be adapted to work in the presence of clock drift. Drift can be detected if either the polling time exceeds $2J_{max}$ or if the target TSC was missed. In the latter case, the time-critical code will be delayed by some small amount, which may not be acceptable. Once detected, the task can be rescheduled for a longer or shorter period by preprogramming the PIT.

It is possible to compensate for jitter in the presence of clock drift without the chance of delaying time-critical code, or reprogramming the PIT. The technique is to schedule a *subtask* at a shorter period than the nominal period. The period need only be slightly shorter than $T - J_{max}$, but most of the time will be spent polling and task starvation will result. Shorter periods will result in less polling, since most subtask cycles will note that the interval to the target TSC is greater than the period, and return immediately. Eventually a cycle will poll to match the TSC. If the period is too short, however, the increased overhead of invoking subtasks that return immediately offsets the benefit of a reduced polling time for the final cycle.

To estimate the processor load with this technique, note that the number of subtask invocations per nominal period T is T/A, where A is the actual shorter period. $T/A-1$ of these cycles will return immediately without polling, with S denoting the time to service these tasks. The last cycle will poll for a duration of A in the worst case. The accumulated overhead of these subtasks is $(T/A - 1)S + A$, and the processor load is

$$load = \frac{(T/A - 1)S + A}{T}$$

Minimizing this with respect to A and calculating the corresponding load yields

$$A_{min} = \sqrt{ST}$$

$$load_{min} = \frac{2\sqrt{ST} - S}{T}$$

Subtask periods shorter than A_{min} will incur too much time servicing interrupts. Periods greater than A_{min} will incur too much time polling during the final cycle. For a nominal 500-μsec period T and a estimated service time S of 2 μsec, the optimal subtask time A_{min} is 32 μsec, which incurs a 12% additional

load. Figure 9 shows a plot of load for this example, as A varies from very short to very long periods.

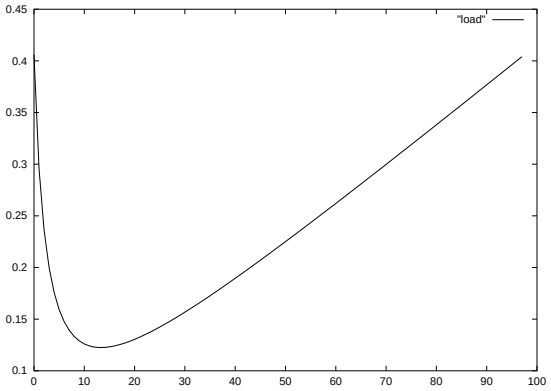

FIGURE 9: *Theoretical processor load as a function of A, the period of a task repeatedly targeting a 500 µsec nominal period. The minimum load is 12%, for a 32 µsec subtask period A.*

This technique results in a significant reduction of jitter, as shown in Figure 10. Here, the uncompensated task shown whose jitter was shown in Figure 2 was targeted with a 50 µsec subtask. The period J_{max} of the uncompensated task was 3.60 µsec, and the compensated J_{max} was 0.098 µsec, a reduction by a factor of almost 40 times. Note that even the first cycle was compensated. The exact processor loading is unknown since S is unknown. Assuming S to be 2 µsec, the loading is about 14%.

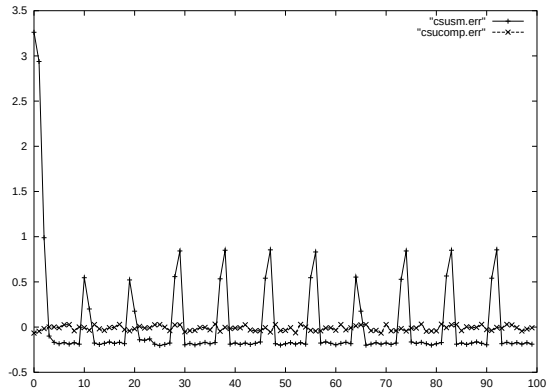

FIGURE 10: *Compensated jitter. Here, a 50 µsec subtask is targeting a 500 µsec nominal period. The period jitter is reduced from 3.60 µsec to 0.098 µsec, a reduction of almost 40 times.*

9.3 Jitter Compensation and ISR Latency

It is possible to built jitter compensation into the real-time scheduler directly. For a single periodic task, this will probably result in satisfactory performance, as demonstrated in the previous example. However, the technique degrades as more tasks are scheduled, since more time will be spent polling on the various target TSC values. The technique also increases the ISR latency, since polling is done with interrupts off. In the example shown in Figure 10, the subtask period was 50 µsecs, and it is possible to spent this much time polling during the final cycle that targets the TSC. This will seriously degrade ISR latency.

10 Summary

Real-time programmers are more interested in time-related OS performance measures such as ISR latency and scheduling jitter than in metrics such as average time to execute benchmark computational tests. Techniques for measuring ISR latency typically rely on external test and measurement equipment. External techniques can also measure jitter, but internal techniques that use the processor's built-in time stamp counter have the benefit of low cost and the potential to be used as a compensatory technique. Cycle-to-cycle and period jitter analysis were described, with period jitter recommended as a performance measure. Synchronous and asynchronous clock issues were discussed, noting that synchronous clocks may need to be dealt with as if they were asynchronous if the relationship between derived clocks is only approximately known. Techniques for jitter compensation for both synchronous and asynchronous clocks were described, and results show the ability to reduce jitter by a factor of almost 40. These techniques can be employed by the operating system, with a tradeoff between jitter reduction and ISR latency.

References

[1] Standard Performance Evaluation Corporation, SPEC Software Performance Benchmarks, `www.spec.org`

[2] OMAC Users Group, *Hard Real-Time Extensions of Windows NT Evaluation Report*, `www.arcweb.com/omac/Techdocs/ntrtrpt2.pdf`, 1998.

[3] Real-Time Linux, `www.rtlinux.org`

[4] Real-Time Application Interface, `www.rtlinux.org`

[5] Jack G. Ganssle, *Interrupt Latency*, Embedded Systems Programming Magazine, pp. 73-76, October 2001.

[6] Phil Wilshire, *Real Time Linux: Testing and Evaluation*, PROCEEDINGS OF THE SECOND REAL-TIME LINUX WORKSHOP, Orlando, FL, 2000. Also available as `ftp://ftp.thinkingnerds.com/pub/projects/rtos-ws/p-a03_wilshire.pdf`

[7] Friedheld Stappert, *Predicting Pipelining and Caching Behavior of Hard Real-Time Programs*, PROCEEDINGS OF THE NINTH EUROMICRO WORKSHOP ON REAL-TIME SYSTEMS, pp. 80-86, 1997.

[8] Uwe F. Mayer, BYTEMark Benchmark for Linux, `www.tux.org/~mayer/linux/bmark.html`

[9] Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, Order Number 243191, 1999. Also available as `developer.intel.com/design/pentiumii/manuals/243191.htm`

[10] Piyush Sevalia, *Straightforward Techniques Cut Jitter In PLL-Based Clock Drivers*, ELECTRONIC DESIGN NEWS, July 6, 1995. Also available as `http://www.cypress.com/design/techarticles/950706edn.html`

[11] Frederick M. Proctor and William P. Shackleford, *Timing Studies of Real-Time Linux for Control*, PROCEEDINGS OF THE 2001 ASME COMPUTERS IN ENGINEERING CONFERENCE, Pittsburgh, PA, 2001.

[12] Frederick M. Proctor and William P. Shackleford, *Real-Time Operating System Timing Jitter and its Impact on Motor Control*, PROCEEDINGS OF THE SPIE INTERNATIONAL SYMPOSIUM ON INTELLIGENT SYSTEMS AND ADVANCED MANUFACTURING, VOL. 4563, Boston, MA, 2001.