Quantum Support for Multiprocessor Pfair Scheduling in Linux

John M. Calandrino and James H. Anderson
Department of Computer Science, The University of North Carolina at Chapel Hill

Abstract

This paper discusses several modifications to the Linux operating system in order to support aligned, staggered, *and* desynchronized *quanta across multiple processors, where a quantum is a unit of processor allocation. We also evaluate one approach for supporting aligned quanta. These types of quanta are required for global multiprocessor real-time scheduling algorithms such as PD^2 (a Pfair algorithm). Additionally, we consider the issue of increasing the frequency of timer interrupts to allow for the smallest quantum possible, and discuss impediments to higher timer interrupt frequencies. Smaller quanta are important, as they can increase the efficiency of quantum-based scheduling algorithms; however, they are more sensitive to quantum alignment error.*

1 Introduction

In multiprocessor real-time systems, global scheduling algorithms are often believed to be inefficient when compared with partitioned approaches, primarily due to increased scheduling, preemption, and migration costs. Pfair scheduling algorithms [2, 1, 7] are the only known optimal multiprocessor scheduling approach. However, they are also global scheduling algorithms. Their optimality is based on the assumption, common in theoretical work in real-time systems, that the costs noted above are negligible. While the emergence of multicore architectures is making this a more reasonable assumption than before, quantum-based algorithms such as Pfair algorithms also suffer from other less-discussed issues in practice.

First, it is often hard to achieve the types of quanta required for such algorithms. Optimal Pfair scheduling algorithms require perfectly *aligned* quanta. In addition, suboptimal Pfair algorithms have been proposed that ensure bounded deadline tardiness and that require quanta that are either *staggered* [6] or *desynchronized* [4]. These types of quanta, which are defined and discussed more thoroughly in Section 4, are unsupported, or difficult to support, especially in general-purpose operating systems.

Second, requiring that all scheduling decisions be made only on quantum boundaries can create scenarios where a substantial amount of processor utilization is wasted. For example, a task that has an execution cost equal to 1.01 quanta will have to be allocated two quanta for every job. If we use an optimal Pfair algorithm for scheduling, then the second quantum allocated to a job will be 99% unutilized, and as a result, the job will waste nearly half of its total processor allocation. In the worst case, where quantum sizes are very large relative to the execution costs (or execution costs modulo the quantum size) of tasks, this could result in a highly inefficient scheduling algorithm that underutilizes the system.

This paper describes an effort to alleviate these issues in the Linux operating system. Specifically, we explore strategies for supporting quanta that are perfectly aligned across processors, staggered, or desynchronized, primarily to provide support for efficient implementations of quantum-based, global, multiprocessor real-time scheduling algorithms such as PD^2, a Pfair algorithm. Additionally, we investigate the issue of increasing the frequency of timer interrupts as much as possible, thereby minimizing the duration of a quantum and the potential inefficiency of quantum-based scheduling approaches.

This paper is organized as follows. Section 2 briefly introduces Pfair scheduling algorithms. Section 3 provides an introduction to the timer facilities provided in hardware, and how they are used in Linux, specifically in the context of when and how the scheduler is called. Section 4 describes how we might modify Linux to achieve various types of quanta, and discusses the pros and cons of several approaches. Section 5 demonstrates the benefit of our approach for generating aligned quanta. Finally, Section 6 concludes.

2 Pfair Scheduling Algorithms

We now provide a brief introduction to Pfair scheduling algorithms, which can be used to schedule *periodic task systems*. Each task in such a system is invoked repeatedly; each such invocation is called a *job* of the task. A periodic task is specified by a *period*, which denotes the (exact) separation between its successive job releases, and by an *execution cost*, which denotes the maximum execution time of any of its jobs. Each job of a task has a deadline corresponding to the release time of the next job of that task. Task periods and execution costs are assumed to be integral with respect to the length of the system's scheduling quantum, and therefore non-integral execution costs must be rounded up to the next integer value. The *utilization* or *weight* of a task is given by the ratio of its execution cost and period.

In Pfair scheduling algorithms [2, 7], a task T of weight $T.wt$ is scheduled one quantum at a time in a way that approximates an *ideal* allocation in which it receives $L \cdot T.wt$ time over any interval of length L. This is accomplished by sub-dividing each task into a sequence of quantum-length *subtasks*, each of which must execute within a certain time *window*, the end of which is its *deadline*. Subtasks are scheduled on an earliest-deadline-first basis, and tie-breaking rules are used in case of a deadline tie. A task's subtasks may execute on any processor, but not at the same time (*i.e.*, tasks must execute sequentially). The most efficient known optimal Pfair algorithm is PD^2 [1, 7], which uses two tie-breaking rules. The performance of PD^2 depends substantially on the type of scheduling quanta used, as we will discuss further in Section 4. Additionally, if a task is allocated a quantum when it requires less execution time, the unused portion of that quantum is "wasted." This is in contrast to other global multiprocessor scheduling approaches, such as global earliest-deadline-first (EDF), where such a task would relinquish its assigned quantum "early," allowing another task to be scheduled.

3 Introduction to Timers in Linux

We begin our discussion of timers by describing the platform we assume in this paper. Our platform is a symmetric multiprocessor (SMP) architecture consisting of four 32-bit Intel(R) Xeon(TM) processors running at 2.70GHz. Each processor has private 8K instruction and data caches, and a private, unified, 512K L2 cache. The machine has 2GB of main memory. Our platform uses the Linux 2.6.9 kernel configured to run on an SMP architecture, and most aspects of what is discussed in this paper should remain the same for (at least) any release version of Linux 2.6. The timer interrupt hardware and its operation in Linux are shown in Figure 1. Note that the following overview of timer interrupts in Linux is based heavily on the material in [3].

3.1 Local Timer Interrupts

In the configuration described above, each processor contains an Advanced Programmable Interrupt Controller, or APIC, which is on the same chip as the processor itself. A *local APIC timer* is capable of generating *local timer interrupts* at each processor, without the need for an external timer interrupt source. In Linux, scheduling decisions are made in the local timer interrupt handlers, thus supporting the notion of a scheduling quantum. As each APIC is programmed to generate these interrupts at the same frequency on all processors, and all local APIC timers are driven by the same external clock signal, the size of a quantum is identical across all processors. However, this does *not* imply that such quanta are *aligned* across all processors—for this to be the case, all local APIC timers would have to be programmed and enabled at the same time (within some level of precision). In Linux, this is unlikely, since processor 0 enables its APIC timer with a different routine from all other processors in the system, and the time at which each processor in the system will come online and begin its boot sequence with respect to every other processor is

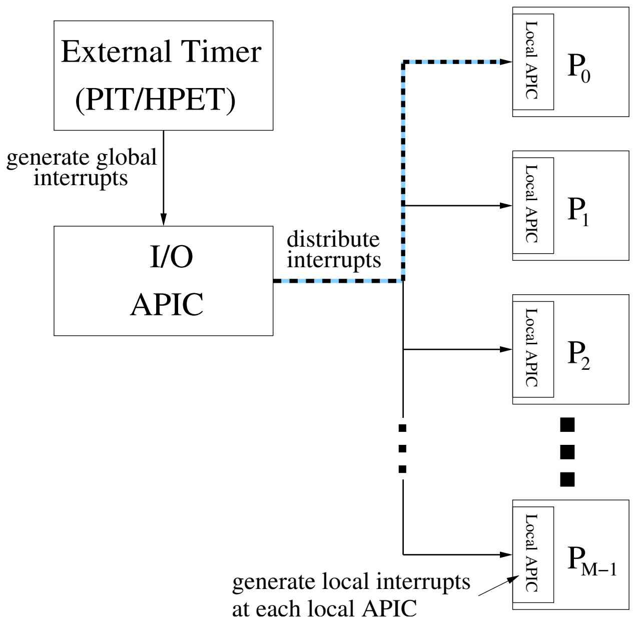

External Timer (PIT/HPET)
generate global interrupts
I/O APIC
distribute interrupts
Local APIC
P0
P1
P2
PM−1
generate local interrupts at each local APIC

Figure 1: Timer interrupt hardware. The external timer chip generates global timer interrupts, which are passed to the IO-APIC and distributed to a single processor. Local timer interrupts are generated by the local APIC at every processor.

not predictable. In fact, the amount of time between when different processors come online is not only unpredictable, but could even exceed the duration of a quantum in the extreme case. It is important to note, however, that since all timers are driven by the same clock signal, the amount by which quanta are misaligned across all processors should not change significantly after the system boots. Therefore, if we had some way of initially synchronizing all processors such that all APIC timers were enabled at the same time, we would be able to support aligned quanta.

3.2 Global Timer Interrupts

There is also a second type of timer interrupt generated in Linux, the *global timer interrupt*. These interrupts are generated by an external source—a separate timer chip, most often the Programmable Interval Timer, or PIT. A second, more accurate timer chip, called the High Precision Event Timer, or HPET, also exists, and is expected to subsume the PIT in the near future. In Linux, this timer is programmed to generate interrupts at the same frequency as the local APIC timers—in fact, every local APIC uses the global timer interrupt to calibrate itself during setup in Linux.

Interrupts from this chip are received by a device known as the Input/Output Advanced Programmable Interrupt Controller, or IO-APIC. This device is responsible for determining how to distribute the global timer interrupt across all processors. In Linux, the IO-APIC is programmed to distribute each global timer interrupt to a single processor (received by the local APIC at that processor as an external interrupt) using a pseudo-"round-robin" policy. The global timer interrupt handlers perform periodic processor-independent activities such as updating system

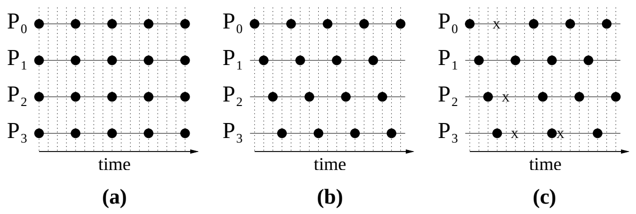


Figure 2: Various types of quanta on a four-processor machine: **(a)** aligned quanta; **(b)** staggered quanta; **(c)** desynchronized quanta. Black dots indicate the quantum boundaries at each processor. An "X" indicates where a processor went idle (due to the completion of its scheduled job before the end of the quantum), thus indicating both where the processor called the scheduler and the start of the next quantum.

uptime (measured in *jiffies*, or the interval of time between two consecutive global timer interrupts). While any scheduler called in the local timer interrupt may use the system uptime when making scheduling decisions, there are no actual scheduling decisions made during the global timer interrupt.

4 Supporting Quanta in Linux

We next describe several types of quanta and the ways in which we modified Linux in an attempt to support them. Three types of quanta are supported, as mentioned briefly in Section 1, and shown in Figure 2. With *aligned quanta*, quantum boundaries are at identical times on every processor, *i.e.*, quanta are *aligned* across all processors. With *staggered quanta*, first proposed in [6], quantum boundaries are not aligned but are instead *staggered* so that every processor performs scheduling activities at different, but predictable, times. For example, consider a four-processor system with a quantum size of 4 time units. If processor 0 has a quantum boundary at time x, then processors 1, 2, and 3, would (ideally) have quantum boundaries at times $x + 1$, $x + 2$, and $x + 3$, respectively. Note that all processors still have quanta of identical size. Staggered quanta alleviate issues related to bus contention that can occur with aligned quanta during rescheduling, as the overhead of scheduling and related costs (*i.e.*, preemption and migration) are now better distributed across time, rather than creating large spikes in bus activity at every (aligned) quantum boundary.

Finally, with *desynchronized quanta*, processors may make scheduling decisions between quantum boundaries, if a processor goes idle during that time due to the (early) completion of the previously-scheduled job. When scheduling decisions are made between quantum boundaries, the next quantum boundary is set to be one quantum away, thus leading to a controlled desynchronization of quanta across all processors. Such quanta can initially be aligned or staggered, but staggered, if available, would likely generate less overhead.

Note that, while we attempt to support both staggered and desynchronized quanta, both result in a sub-optimal Pfair algorithms, where deadlines can be missed by one

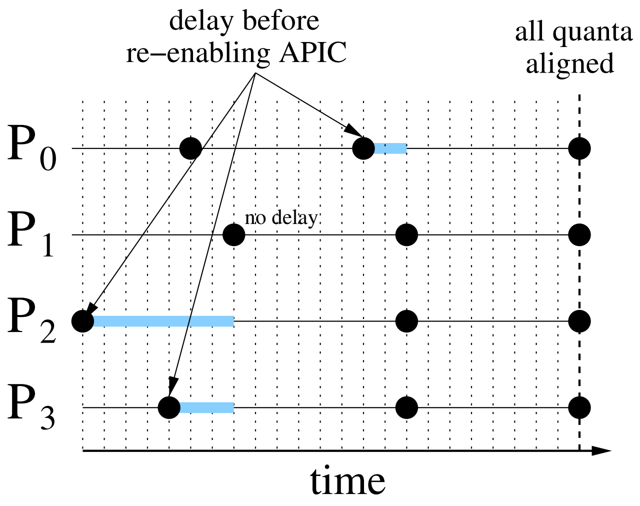


Figure 3: Example of the proposed modification to support aligned quanta in Linux, described in Section 4.1, assuming all processors have already seen their $100,000^{th}$ local interrupt, and recorded at least 50 interrupt invocation times from the TSC.

(full) quantum [4, 6]. This amount, however, is still considerably less than the amount by which tasks can miss deadlines with other global multiprocessor scheduling approaches, such as global EDF [5].

4.1 Supporting Aligned Quanta

We supported aligned quanta by employing the following method, shown in Figure 3.

- Allow the system to boot, and both global and local timer interrupts to initialize normally. To allow enough time for this to occur, we wait until the $100,000^{th}$ local interrupt is generated at every processor.

- When each processor sees its $100,000^{th}$ local interrupt, it begins recording the times at which its interrupt handler is being invoked by recording the value of the Time Stamp Counter (TSC), a cycle-based 64-bit counter that records system uptime in nanoseconds. Each processor keeps a record of the time at which the last 100 local timer interrupts have been invoked.

- All processors except processor 0 wait until processor 0 and themselves have recorded at least 50 interrupt invocation times from the TSC. When this occurs, they use the TSC measurements to calculate how misaligned they are with respect to processor 0, and by how much they need to delay to align themselves.

- Each processor except processor 0 then disables and resets its local APIC timer, so that when it is re-enabled, it will generate its next interrupt after a full timer period.

- Finally, each processor except processor 0 delays appropriately, and then re-enables its local APIC timer.

- If necessary, after all other processors have aligned themselves, processor 0 makes a small adjustment to its own interrupt generation times, if necessary, to prevent the calculation of negative delay times on the other processors, by delaying for an appropriate time and resetting its APIC timer as described for the other processors.

We determine how much to delay at each of M processors by solving a system of M equations representing the delay and quantum boundary offsets of every processor. Consider as an example a four-processor system, which generates the following system of equations. Assume that a local timer interrupt occurs on processor p when the system has been up for t_p nanoseconds, and we wish to calculate the amount of delay, d_p, required on processor p to bring all quanta into alignment.

$$\forall p : 0 \leq p < M - 1 :: (t_p + d_p = t_{p+1} + d_{p+1}) \quad (1)$$

Solving the system of equations for d_0, we get

$$\begin{aligned} d_0 &= d_0 \quad (2) \\ d_1 &= d_0 + t_0 - t_1 \\ d_2 &= d_0 + t_0 - t_2 \\ d_3 &= d_0 + t_0 - t_3. \end{aligned}$$

Since we still want all delays to be non-negative, we calculate d_0 as follows. Note that d_0 may be non-zero—this will allow some other processor to have a zero delay instead of a negative delay, as seen in Figure 3 for P1.

$$d_0 = max(0, t_1 - t_0, t_2 - t_0, t_3 - t_0) \quad (3)$$

Such delays were realized using a non-timer-based kernel delay function called *udelay*, which is implemented using a software loop with μs granularity.

By delaying for the times specified by these equations, all local timer interrupts are brought into alignment with each other; in experiments conducted by us and discussed later in Section 5, the resulting interrupt invocations on each processor differed by approximately twenty-five microseconds at most. Using this method, we can get aligned quanta even if quanta were misaligned by hundreds of microseconds before delaying. We cannot make such a statement about standard Linux—if all processors do not come online and initialize their APICs at approximately the same time, quanta can be substantially misaligned. Additionally, we note that these delays are typically small, and therefore do not cause other interrupts to be missed, nor do other interrupts have the capability to interfere significantly with delay times. As we perform these delays during boot time, there are few interrupts generated besides those associated with the timers, since neither the network nor most I/O devices are yet initialized, and the user is not yet interacting with the machine. This is particularly true on processors 1 and above, which receive only their local timer interrupts until the *irqbalance* service is initialized, which distributes interrupts across processors rather than always sending them to processor 0.

Note that our method is reasonably straightforward and quite accurate. Our earlier attempts to implement aligned quanta relied on substantially more complex approaches, involving broadcasting the global timer interrupt to all processors, or using interprocessor interrupts (IPIs) to synchronize all local timer interrupts. Both approaches were difficult to implement correctly, due to the sensitivity of some components of Linux to any changes in how timer interrupts are distributed, and also were not very effective. Finally, as we will see in the following sections, our method is easily extensible to support other quantum alignments and durations.

4.2 Supporting Staggered Quanta

Supporting staggered quanta, in an effort to prevent high levels of bus contention when all processors execute scheduling code and perform other scheduling activities at the same time, can be slightly more complicated. Two methods can be employed to support staggered quanta:

- Modify when the scheduler is invoked: provide support for aligned quanta, but instead of invoking the scheduler every interrupt, processor 0 would invoke the scheduler during the first interrupt, followed by processor 1 during the second interrupt, etc.

- Add additional delay at each processor: after achieving aligned quanta, delay for an additional amount at each processor to get the stagger we desire.

We will now discuss each method in detail. Figure 4 shows each approach assuming a four-processor machine.

4.2.1 Modify When the Scheduler is Invoked

The simplest way to provide staggered quanta is to leverage support for aligned quanta, but instead of invoking the scheduler on every processor at every (aligned) timer interrupt, we invoke it every x timer interrupts, where x is a multiple of the number of processors in the system. Staggered quanta are then achieved by equally spacing these scheduler invocations across the processors. For example, on a four-processor system where the quantum is 4 timer interrupts long (*i.e.*, $x = 4$), and processor 0 invokes its scheduler at timer interrupt t, processors 1, 2, and 3 should invoke their schedulers at timer interrupts $t + 1$, $t + 2$, and $t + 3$, respectively.

While this is the most straightforward way to implement staggered quanta, as it is both intuitive and relies on an implementation of aligned quanta (so we do not need to further modify when timer interrupts are generated), it results in larger quantum sizes than aligned quanta. On a machine with M processors, our quantum duration would have to be at least M times as large for staggered quanta as for aligned quanta, using this approach. Therefore, an approach that allows the same minimum quantum duration as aligned quanta is desirable.

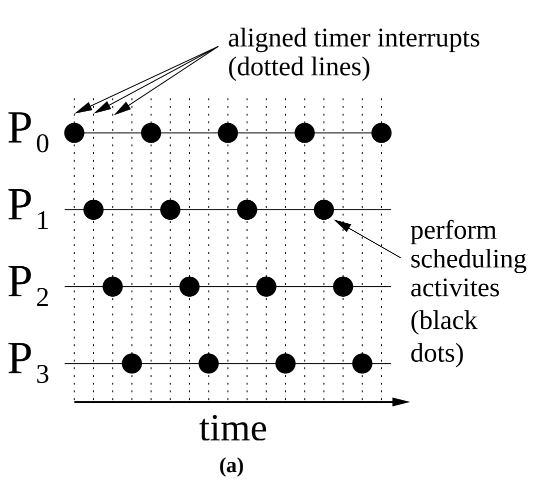
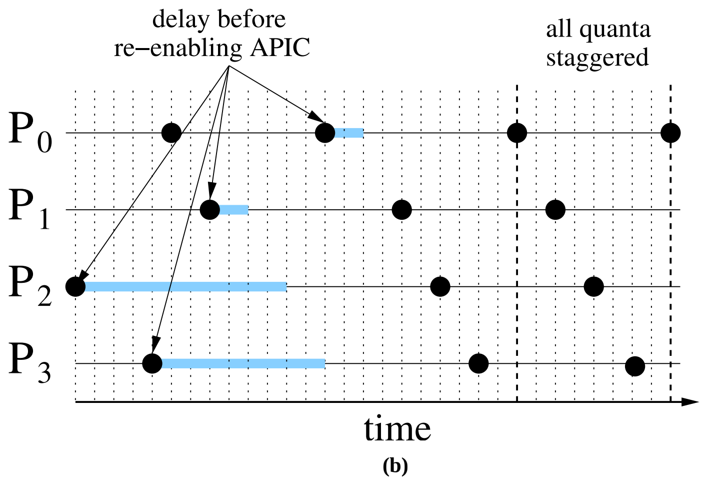

aligned timer interrupts (dotted lines)
P 0
P 1
P 2
P 3
perform scheduling activites (black dots)
time
(a)


Figure 4: Modifications to support staggered quanta in Linux. **(a)** Use aligned quanta, invoke the scheduler only at certain interrupts to create staggered quanta (Section 4.2.1); **(b)** delay each processor appropriately in its local interrupt handler before re-enabling its APIC timer (Section 4.2.2), in a similar manner to the method described in Section 4.1.

4.2.2 Add Additional Stagger Delay

Our final approach requires that we first achieve aligned quanta, and then add the necessary delay at each processor that will allow us to get staggered quanta. We can easily calculate this extra delay as follows. Let s be the duration of a quantum divided by the number of processors in the system. For example, with a quantum duration of one millisecond and four processors, s is 250 microseconds. For processor P, the stagger delay is $P \cdot s$. For processors 0 through 3 in a four-processor system, these delays would be 0, 250, 500, and 750 microseconds, respectively. By delaying for the times specified, all local timer interrupts are staggered appropriately.

4.3 Supporting Desynchronized Quanta

Desynchronized quanta are supported by disabling, resetting, and re-enabling the local APIC timer whenever intra-quanta scheduling decisions are made, thereby readjusting the quantum duration and alignment. When supporting desynchronized quanta, we can begin with either aligned or staggered quanta. However, starting with staggered quanta would likely eliminate any chance of high bus contention (due to scheduler invocations) in the system.

4.4 Reducing Quantum Size

We next attempt to reduce the *duration* of a quantum in the Linux operating system. Smaller quanta are desirable, as their use in Pfair scheduling algorithms may result in less processor idling. In particular, in such algorithms, the overall utilization of the system is highly dependent on quantum duration, as this determines the time granularity at which scheduling decisions are made, and therefore the units by which the execution costs of tasks are effectively determined. If the duration of a quantum is too large, then the WCET of a task may force it to be scheduled for an additional quantum, which is highly underutilized, and this may result in a very low effective system utilization.

As an example, consider a task with an execution cost of 1.1 milliseconds. If the duration of a quantum is one millisecond, then such a task will need two quanta allocated to every job. However, each job will only use 10% of the second allocated quantum, resulting in only a 55% utilization of the time it has been allocated by the scheduler. If the duration of a quantum was 500 microseconds, then the job needs three quanta; however, it only uses 20% of the third quantum, resulting in a utilization of roughly 73%. If, however, the duration of a quantum was 100 microseconds, then there would be no waste at all—each job would require, and fully utilize, 11 quanta (assuming that the WCET of the task is its actual execution time). (Of course, with a smaller quantum, scheduling and preemption costs are greater. We have ignored such issues in this simple example.)

The duration of a quantum is dictated by the frequency of timer interrupts in the system. The frequency of both local and global timer interrupts is governed by a constant called *HZ* in Linux. This value is set to 1000 by default in Linux 2.6.9, allowing for, at best, a one millisecond quantum duration. (The value is set lower in lower versions of the kernel due to overhead concerns on slower processors, and in higher versions of the kernel due to power consumption concerns, neither of which concern us). We have attempted to increase this value as high as possible, and thus far, we have increased it to 3750, providing support for quanta only slightly larger than 250 microseconds. Applying a patch found in the online Linux community (see the post at [8]), which should allow for frequencies as high as 10000, did not help. These difficulties appear to be due to timer calibration issues, and may also be due to the hardware configuration of our system. Higher timer interrupt frequencies have the potential to cause networking and user I/O-related (*e.g.* keyboard input, terminal output) protocols to operate incorrectly, as well, as they appear to be highly dependent on the system timer frequency. Overall, we are uncertain whether other aspects of our system configuration, or a critical path in Linux, are limiting additional fre-

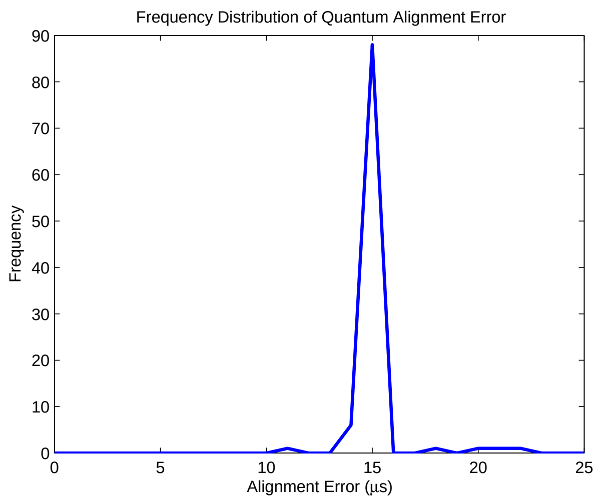


Figure 5: Quantum alignment error using our method.

quency gains.

It is important to note that the use of smaller quanta magnifies any error related to quantum alignment among all processors. For example, we earlier claimed that quantum boundaries across processors differ by about twenty-five microseconds in the worst case. Using a *HZ* value of 1000 results in a quantum duration of one millisecond and an alignment error of $\frac{25}{1000} \times 100 = 2.5\%$. With a *HZ* value of 3750, we have an alignment error of approximately 9.4%, which is substantially larger and might result in serious timing errors.

5 Experimental Evaluation

We implemented and evaluated our approach for generating aligned quanta. Results are shown in Figure 5. These results were obtained by measuring the time between the first and last invocation of a particular timer interrupt across all processors. These times were measured by reading the TSC at the beginning of the local timer interrupt handler. Overall, 100 interrupt invocations were measured after Linux had fully booted and stabilized. More measurements were not taken due to constraints on sizes of files in the *proc* directory, to which data was written. However, we have yet to observe an instance where the quantum alignment error changed significantly over time after the system had booted and stabilized. This is probably related to the fact that all timers are driven by the same external clock signal, as mentioned in Section 3.

It appears that our method of generating aligned quanta has a significant impact. Note that alignment error never exceeds approximately 25 microseconds with our method.

6 Conclusion

This paper discussed ways to support aligned, staggered, and desynchronized quanta in the Linux operating system, in order to provide support for Pfair multiprocessor real-time scheduling algorithms, such as PD^2, in Linux. Additionally, an implementation of aligned quanta was experimentally evaluated. We also determined that the frequency of timer interrupts can be increased as high as 3750 hertz without problems, resulting in a quantum duration of approximately 250 microseconds. The use of a smaller quantum, however, magnifies any quanta alignment error among all processors and could result in timing errors. As Pfair scheduling algorithms are theoretically optimal for multiprocessors, the work in this paper is significant in that brings us one step closer to realizing the full potential of such algorithms in practice.

References

[1] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, February 2004.

[2] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[3] D. Bovet and M. Cesati. Timing measurements. In *Understanding the Linux Kernel, 3rd edition*, pages 227–257. O'Reilly Publishers, 2005.

[4] U. Devi and J. Anderson. Desynchronized Pfair scheduling on multiprocessors. *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, April 2005.

[5] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. *Proc. of the 26th IEEE Real-time Systems Symposium*, Dec. 2005.

[6] P. Holman and J. Anderson. Adapting Pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4), 2005. to appear.

[7] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proc. of the 34th ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[8] J.-M. Valin. Increasing HZ (patch for HZ $>$ 100). http://www.uwsg.iu.edu/hypermail/linux/kernel/0312.1/0858.html, December 2003.