A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software *

Balaji Srinivasan, Shyamalan Pather, Robert Hill, Furquan Ansari, Douglas Niehaus
Electrical Engineering and Computer Science Department
University of Kansas
Lawrence, KS 66045

Abstract

The emergence of multimedia and high-speed networks has expanded the class of applications that combine the timing requirements of hard real-time applications with the need for operating system services typically available only on soft-real time or timesharing systems. These applications, which we describe as firm real-time, currently have no widely-available, low-cost operating system to support them. In this paper we discuss modifications we have made to the popular Linux operating system that give it the ability to support the comparatively stringent timing requirements of these applications, while still giving them access to the full range of Linux services. Using our firm real-time system as a basis, we have developed the ATM Reference Traffic System (ARTS) that is capable of recording and accurately reproducing packet-level ATM traffic streams with timing resolution in microseconds. The effectiveness of this application, as well as the comparative ease with which it was developed, illustrate the performance and utility of our system.

1 Introduction

Until the fairly recent past, most real-time applications fit into two broad categories. The first consists of those applications with *soft* real-time constraints which require timely execution of tasks at coarse temporal resolution, but do not produce catastrophic consequences if their deadlines are violated. The second group consists of *hard* real-time applications. These applications impose stringent timing demands on their operating systems, with disastrous, and sometimes fatal, consequences resulting from temporal errors.

Soft real-time applications can usually be supported by generic desktop operating systems with slightly extended timing and scheduling capabilities. These extensions seldom preclude the soft real-time applications from accessing the services offered by these operating systems. Hard real-time applications, in contrast, typically require specialized operating systems that run on very specific hardware and come at a significant cost. In order to meet the strict timing requirements of hard real-time applications, these operating systems can often provide only a very austere execution environment, offering few application services.

Recent developments in many areas, including multimedia and ATM networking, have spawned several applications that defy the binary hard/soft classification system. Multimedia video applications, for example, exhibit comparatively strict timing requirements typical of hard real-time applications, since they need to maintain very precise display refresh rates. However, most hard real-time operating systems cannot, or can only partially, support such multimedia applications because they require a wide variety of system services. Our own research in the development and performance evaluation of ATM networks has included several applications with similar combinations of fine grain timing requirements (typical of hard real-time systems) with system service requirements typical of soft real-time systems. These applications, which fall neither into the hard nor soft real-time classes, might reasonably be called *firm*. Furthermore, many of these applications exhibit stringent cost constraints. We could find no system which adequately satisfied the combination of temporal, service, and cost constraints exhibited by this class of applications.

Another important characteristic of firm real-time applications is that they are seldom designed to be run on dedicated real-time hardware. For example, a video-conferencing application would typically be targeted for use on a standard desktop system, in conjunction with conventional applications such as word processors or web browsers. Therefore, a crucial requirement of any operating system claiming to support firm real-time applications is that it allow real-time and non-real-time tasks to coexist.

We have created the KU Real-Time (*KURT*) system

*This work was supported by grants from Sprint Corporation.

which satisfies the constraints of many applications in the firm real-time category. KURT is based on the freely available Linux operating system, modified in several important ways. First, by running the hardware timer as an aperiodic device (as described in Section 3) we have increased the system's temporal resolution without significantly increasing the overhead of the software clock. We call this the *UTIME* extension to Linux, which alone increases the utility of Linux for soft real-time applications.

The KURT system builds on top of UTIME, adding a number of features. KURT enables the system to switch between three modes: **normal mode**, in which it functions as a normal Linux system, **mixed real-time mode**, in which it executes designated real-time processes according to an explicit schedule, serving non-real-time processes when the schedule allows, and **focused real-time mode**, in which only real-time processes are run. Focused real-time mode is useful when KURT is being used as a dedicated real-time system. When real-time applications share a generic desktop workstation, KURT's mixed real-time mode is most often used, since it allows both real-time and non-real-time processes to run. When in either real-time mode, real-time processes can access any of the system services that are normally available to non-real-time processes. However, as discussed in Section 4.2, the use of different subsystems introduces different levels of scheduling distortion. While many approaches to scheduling can be easily implemented, we have found explicit scheduling to be the most appropriate for our current applications.

One such application is the ATM Reference Traffic System (ARTS), described further in Section 3. This application can record and generate packet-level ATM traffic streams with microsecond accuracy. ARTS uses KURT's scheduling abilities in its traffic generation module by placing packet transmission events in an explicit KURT schedule. KURT and closely related approaches are also being applied in our lab to a range of problems, including synchronized distributed real-time computation, the implementation of a software ATM switch with precise rate control, and support for multimedia applications.

We believe KURT fills an important gap in the existing spectrum of real-time systems by providing a low cost system with fine-grain temporal resolution, capable of satisfying firm real-time performance constraints. However, the generic Linux services have not been specifically adapted to real-time execution and are thus sources of scheduling distortion. Such adaptations are part of the future work we intend to do. In spite of this limitation, KURT has provided excellent support to our firm real-time applications for which no other system was as appropriate. In addition, it provides a well structured environment within which to investigate and resolve the sources of scheduling distortion.

In the next section we discuss related work in real-time operating systems. In Section 3 we describe the implementation of the UTIME Linux extension, the KURT system on top of it, and the implementation of ARTS. The experimental results from our tests of UTIME, KURT, and ARTS are presented in Section 4 and in Section 5 we discuss our conclusions and possible future work. We have not included a detailed description of the KURT application programming model, as this has been discussed fully elsewhere [8].

2 Related Work

The recent proliferation of firm real-time applications has motivated several efforts to produce a suitable operating system for them. In this section, we discuss a few of the systems which share some of the characteristics of our system.

The Rialto [5] operating system, developed at Microsoft Research, was designed from the beginning to support the type of applications that we classify as firm real-time. Rialto real-time applications specify their time constraints and interact dynamically with the system to achieve their desired scheduling properties. Rialto provides a wide array of operating system services to its real-time tasks and these are accessed through a per-machine resource planner. In a manner similar to the one we have employed, Rialto runs the hardware timer as an aperiodic device, allowing for much finer timing resolution than is typically available in generic operating systems. While Rialto has been successful in its development context, it suffers from two main drawbacks. First, it is an entirely new operating system and hence has no existing base of application software. This limits its applicability in production environments. Also, it is a proprietary system, making it unsuitable for use in many research environments, where low cost and access to source code are essential.

Stanford's SMART system [7] is a real-time scheduler for multimedia applications, implemented within the Solaris$^{(TM)}$ operating system. It uses a weighted fair queuing scheme to ensure that real-time tasks meet their deadlines, while at the same time providing acceptable levels of responsiveness to non-real-time tasks. It shares with our system the advantage of being implemented inside a full-featured, widely used operating system and thus can be easily deployed in production environments. However, it uses a periodic clock, limiting the timing resolution it can offer.

Real-Time Linux (RT-Linux), developed at New Mexico Tech [3], is another system based on the Linux. Instead of improving the abilities of Linux directly as KURT does, RT-Linux implements a small real-time executive which runs a non-real-time Linux kernel as a completely preemptable, low-priority task. This approach is similar to the one used at the University of North Carolina to allow the IBM Microkernel to coexist with a simple real-time kernel on a sin-

gle system [4]. The RT-Linux model requires that real-time applications be split into real-time and non-real-time parts. The real-time parts run under the real-time executive and the non-real-time parts under the low-priority Linux kernel. Communication between the two environments is supported by lock-free queues and shared memory, but the parts running under the real-time executive cannot access any of the Linux services.

This approach is interesting and useful for applications fitting the RT-Linux model. It is not, however, appropriate for the class of applications KURT addresses, precisely because these applications require a combination of access to system services and support for real-time constraints, though these constraints are often less stringent than those RT-Linux can support. For example, if ARTS were implemented under RT-Linux, then its real-time module would have to be able to send an ATM packet into the network. This is not possible, however, because access to the networking subsystem of Linux is not supported by RT-Linux.

There are also several commercially available systems, including LynxOS [1] and QNX [2], which offer real-time performance and a number of services to the applications they run. While these systems are attractive from the point of view of features, they are too costly to be used in research institutions operating on limited budgets.

3 Implementation

KURT is based on the Linux operating system, a freely available, popular Unix clone. Linux was chosen because it enabled us to meet two important design goals. First, we wanted our system to be easy to integrate into existing computing environments. In order to achieve this goal, we had to target an operating system that people already use. Over the past few years, Linux has grown exponentially in popularity, leading to the development of a broad software and user base. Also, since low cost was a primary design goal, Linux was an ideal choice as it runs on commercial off-the-shelf hardware and is distributed without charge.

In order to implement KURT, several changes had to be made to Linux. First, its temporal granularity had to be refined, since many firm real-time applications need to track time at a resolution much higher than that conventionally used. In the ARTS application, for example, scheduling resolution on the order of 10 microseconds is required. This makes the standard Linux software clock resolution of 10 milliseconds (1 millisecond on the DEC Alpha architecture), and that of most other systems, unacceptable. In addition, the standard Linux timesharing scheduler was unsuitable for firm real-time applications, so we had to implement a scheduler that would give firm real-time tasks the preferential treatment required to satisfy their timing constraints.

3.1 Implementation of UTIME

In order to increase the resolution of the Linux software clock, we had to alter the basic mechanism by which it is implemented. In standard Linux, a hardware timer chip is programmed to interrupt the CPU at a fixed rate. Each time the CPU is interrupted, the kernel updates its software clock to indicate the passing of another tick and checks if there are any scheduled events that are due for processing. Thus the software clock is nothing more than a running count of the number of ticks, or hardware timer interrupts, that have passed since the kernel was started. The length of time between each software clock tick is referred to as a *jiffy*. Since the kernel only checks its list of pending scheduled events at each timer interrupt, the length of time between interrupts (or the length of each jiffy) is the smallest meaningful unit of time with which events can be scheduled. Thus the length of a jiffy determines the kernel's timing resolution.

A naive approach to increasing the timing resolution would be to simply program the timer chip to interrupt the CPU at a higher frequency. This would reduce the length of a jiffy and therefore increase the software clock resolution. However, since the timer chip would regularly interrupt the CPU, irrespective of whether any events were scheduled to occur at the time of each interrupt, this approach leads to unnecessary overhead.

Our solution to the problem of increasing the kernel's timer resolution is based on the following observation: there is a crucial difference between the temporal resolution and the frequency of events. In other words, even though firm real-time applications schedule events with microsecond level deadlines, events are rarely scheduled to occur *every* microsecond. To support such applications, what is needed is a mechanism by which timer interrupts are allowed to occur at *any* microsecond, not necessarily *every* microsecond.

Our system departs from the common practice of using the timer chip to interrupt the CPU at a fixed rate. Instead, the timer chip is programmed to interrupt the CPU in time to process the *earliest* scheduled event. When an interrupt is serviced, the kernel checks its list of pending events to see when the next event is due for processing and programs the timer chip to interrupt it in time to service that event. Thus the CPU is interrupted only when it needs to be and not at regular intervals. In this simple form, our scheme eliminates the timer interrupts at which no events are due for processing. Since the timer chip can be programmed with microsecond accuracy, we effectively achieve microsecond resolution.

In this simple version of our system, the count of ten millisecond jiffies is no longer updated, since there are no regular ten millisecond interrupts at which to do so. This presents a problem since there are several kernel subsystems that assume, both implicitly and explicitly, that the

jiffy counter is being updated at regular intervals. To keep these systems working, we had to devise a way to reproduce this periodic "heartbeat" in the absence of regular, periodic interrupts. To do this, we determine the correct jiffy count value at each timer interrupt by computing the number of CPU cycles elapsed since boot time. This computation is performed using the *time stamp counter* (TSC), a 64-bit register that increments at the clock rate of the 200 MHz Intel Pentium $^{(R)}$ Pro processor used in our test system.

When the computed jiffy counter increments, we perform the various activities that are usually done by the standard kernel during each timer interrupt. Since our interrupts occur with little regularity, we could miss one or more jiffy boundaries. To overcome this problem, when reprogramming the timer chip to trigger the next interrupt in time to process the next event, we check to see if that event occurs after the next jiffy counter increment would have occurred. If this is found to be the case, then we schedule an interrupt to occur just in time to increment the jiffy counter. This introduces some interrupts at which no events are due for processing, but keeps all kernel subsystems, including the software clock, working as normal. The implications of this scheme, in terms of system overhead and scheduling distortion, are discussed in Section 4.

3.2 KURT Scheduler

As discussed earlier, in addition to increasing the temporal resolution of Linux, we also had to implement a new scheduling algorithm for firm real-time applications. To the FIFO, round-robin, and normal timesharing scheduling algorithms available in standard Linux, we have added the KURT scheduler. This is an explicit plan scheduler, requiring real-time applications to state explicitly the times at which events are to occur. We have also introduced the concept of *real-time modes*, in which those processes that are marked as using the KURT algorithm are scheduled, either exclusively (focused real-time mode) or in conjunction with non-real-time processes (mixed real-time mode). When operating in focused real-time mode, normal non-real-time processes cannot interfere with the timely execution of real-time processes, but the real-time processes still have full access to all Linux services such as network protocol implementations and hardware device drivers.

The KURT system consists of real-time kernel modules which perform certain application-specific activities and a base system which invokes these modules at scheduled times. Applications pass the KURT base system a *schedule file* that lists the times at which certain real-time kernel modules are to be invoked. These modules execute in kernel mode and can therefore access devices, as well as other parts of the kernel, in ways that would not be permitted to normal user-level processes. Furthermore, new application-specific modules can be added to a running kernel at any time.

The module-centric approach used by the KURT scheduler lends itself well to many types of real-time applications. For example, in the ARTS system described earlier, we implemented a real-time module that could transmit an ATM packet. The ARTS traffic generator then simply consisted of a program that would generate a KURT schedule file specifying the times at which to invoke this ATM transmit module, together with a program that would pass this schedule file to KURT and start the scheduler. The fact that the transmit module could run in kernel mode added greatly to the effectiveness of our system, because packets could be written directly to the network interface card. If this application were written as a user-level process, then unpredictable delays would be incurred while copying data between user and kernel space and while performing context switches.

Although applications such as ARTS are well-suited to a real-time system based on kernel modules, there are other applications that work better as a group of coordinated user processes that are scheduled in real-time. To facilitate such applications, the KURT system includes the *process module*. This is a built-in real-time kernel module whose function is to switch context to a specified user process. A schedule file for an application that consists of real-time user-level processes simply specifies the times at which the built-in process module should be invoked and to which user-level process context should be switched. Since, from the perspective of the KURT scheduler, the process module is the same as any other real-time module, the schedule file for such an application is written in exactly the same format as that used for an application such as ARTS.

Explicit scheduling using schedule files as described above works well for many real-time applications. However, some applications have periodic execution flows, wherein a certain section of code is executed repeatedly at a fixed time interval. For these applications, explicit event times can be specified, but this approach is not natural. A more attractive approach is to allow these applications to specify the length of one period and then be scheduled by KURT to run once every period. To accomplish this, a periodic application can switch KURT into *periodic mode* and use a system call to specify its period. The application can then use a KURT system call to *suspend* its execution at the top of its periodic loop. Once the scheduling is begun, KURT will allow the application to run and cause the periodic loop to execute. Once an iteration is complete, the application should suspend itself once again and wait for KURT to re-activate it for the next iteration.

4 Evaluation

To evaluate our system, we chose to test each piece individually and then test the system as a whole by means of an application. In this section, we will present the results of the tests we ran to measure the overhead of the UTIME system, the associated clock drift that it introduces, and the accuracy of the KURT scheduler. In addition, we will show that these systems worked together to lend the ARTS traffic generator a very high degree of accuracy, as evidenced by tests performed using an external network analyzer.

4.1 UTIME Tests

As stated in section 3.1, running the hardware timer as an aperiodic device allows the UTIME system to achieve significant gains in temporal resolution without dramatically affecting the system overhead. Since most of the UTIME implementation centers around changes to the Linux timer interrupt handler, comparing the processing time of a timer interrupt under UTIME with that of a timer interrupt in standard Linux gives a quantitative measure of the overhead resulting from UTIME.

We measured the processing time of timer interrupts using the TSC, described in Section 3.1. Approximately 8000 timer interrupts were measured, first on standard Linux and then on Linux with the UTIME modifications applied. Table 1 shows the results. Without UTIME, timer interrupts take 1 microsecond to process on average and virtually all take less than 8 microseconds. With UTIME in place, the average timer interrupt requires just over 7 microseconds of processing time and 99% of all timer interrupts require 22.5 or fewer microseconds. It should be noted that the times described here do not include the time required to process any scheduled events: the timer interrupt handler simply checks which events are due for processing and marks them as such.

The exact value of the timer overhead is less important than the fact that it is greater under UTIME than under standard Linux. This increase can be attributed to the fact that timer interrupts in UTIME do not occur at regular intervals and therefore some amount of extra computation is required in the handler to determine the new value to load into the timer chip to trigger the next interrupt. This is not the case in standard Linux because the timer chip is simply programmed, during system boot-up, to interrupt the CPU at a fixed rate. Even though the actual time required to process timer interrupts using UTIME increases, this penalty is incurred at most once every 10 milliseconds if no microsecond events are scheduled. Thus, the overhead presented by the UTIME modification is, on average, about 0.073%.

In addition to affecting the system overhead, the UTIME modifications also affect the software clock drift. In standard Linux, the count of ten millisecond jiffies is the primary means by which the kernel tracks time. Unfortunately, the assumed jiffy length is rarely correct, because the hardware timer chip has a finite resolution and the smallest value by which it can be incremented usually does not divide evenly into ten milliseconds. The accumulation of error due to the imprecision in the jiffy length causes the software clock to drift over time. Since timer interrupts do not occur at regular intervals in UTIME, a different method is employed to update the jiffy counter. At each interrupt, a TSC reading is taken and used to compute the number of cycles elapsed since boot time. Using the pre-computed number of cycles per second, the number of elapsed cycles is converted into time and used to update the jiffy counter appropriately.

	Mean	Std. Dev.	Min	99%	Max
Standard	1.0μs	1.2μs	0.5μs	7.9μs	13.1μs
UTIME	7.3μs	2.8μs	6.3μs	22.5μs	31.7μs

Table 1. Timer Interrupt Processing Times for Standard Linux and Linux with UTIME

It would seem that this method would be more precise and hence lead to less clock drift than is observed in standard Linux, but this is not the case. As the lines labelled "UTIME" and "Standard Linux" in Figure 1 show, the two systems show similar clock drift. This is explained by the fact that even though UTIME uses the more accurate TSC to track time, it still uses the timer chip at boot time to determine the number of CPU cycles per second and thus calibrate the TSC.

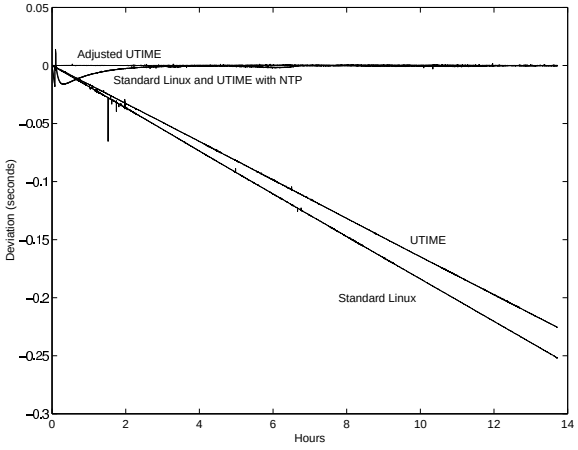

Figure 1. Clock Drift

In our lab we use XNTP, an implementation of the Network Time Protocol (NTP) and associated tools to main-

	Accumulated Deviation
Machine A	145.0ms
Machine B	-137.8ms
Machine C	-235.4ms

Table 2. 14-Day System Clock Deviation Using Adjusted UTIME

tain clock synchronization among workstations [6]. This works equally well for both unmodified Linux and UTIME as shown in Figure 1. These lines lie essentially on top of one another and appear to be a single line labeled "Standard Linux and UTIME with NTP". As the figure shows, the clock drifts initially, but once NTP begins to take effect, the deviation is soon corrected. The line labelled "Adjusted UTIME" in Figure 1 shows the clock drift in a UTIME system where the number of cycles per second is adjusted to the correct value using an external reference instead of the timer chip. The figure shows that in this case, the clock drift is minimal over the 14-hour testing period. This illustrates that with proper calibration, our method of updating the jiffy count using the TSC introduces very little accumulated clock deviation.

To further investigate this point, we monitored the amount of deviation in the software clocks of three different UTIME machines over a 14-day period. These machines all ran the version of UTIME in which the number of cycles per second was calibrated accurately against an external reference, as described above. The results shown in Table 2 indicate that the accumulated deviation is minimal, amounting to less than one-half second per month.

4.2 KURT Scheduler Tests

To test the effectiveness of the new firm real-time scheduling modes, we measured the difference between the times at which real-time kernel modules were scheduled to be invoked and when they were actually invoked. Figure 2 summarizes the results of 10,000 such measurements. As mentioned in Section 3.2, some real-time applications use the built-in process module to schedule user processes. As a reflection of the effectiveness of KURT in scheduling these processes, Figure 2 also shows the distribution of differences between scheduled process module invocation times and the times at which the appropriate user processes began running.

The tests were performed in both focused and mixed real-time modes. Table 3 shows the values below which 99.5% of the differences between actual and scheduled times fell. The maximum value in each category indicates

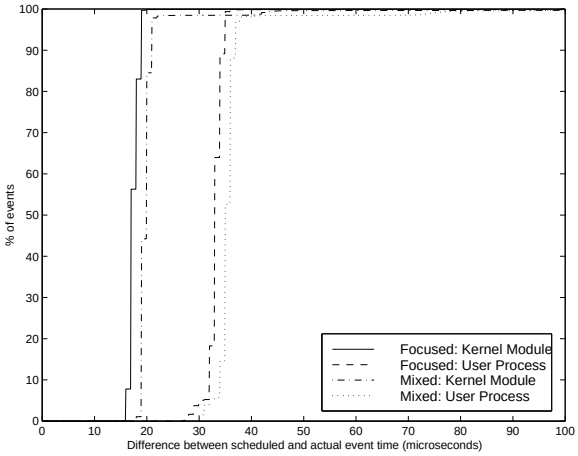

Figure 2. Distribution of Differences between Scheduled and Actual Event Times

that a small percentage of events are still substantially late. This is most likely caused by interrupt service routines executed on behalf of non-real-time system services and processes in mixed real-time mode, which block interrupts for various periods of time. The fact that the distortion is increased under the mixed real-time mode supports this hypothesis.

The scheduler's performance, illustrated in Figure 2, can be improved by trading accuracy for event service overhead. For example, consider the statistic from Table 3 stating that 99.5% of kernel module events occur within 19 microseconds of their scheduled times. In order to make these 99.5% of kernel module events occur within a few (3 to 5) microseconds of their scheduled times, we could schedule them 19 microseconds early and then busy-wait, monitoring the TSC, in the case of events requiring less time than 19 seconds to be dispatched. Slightly less waiting will be required if the cutoff point is chosen below 99.5%, but in any case, this method introduces idle cycles into the event service overhead. This might be a worthwhile design tradeoff, however, for applications with scheduling constraints more stringent than their utilization constraints.

In order to characterize the magnitude and source of the distortion introduced by blocked interrupts, we measured the lengths of the time periods during which interrupts were disabled in the various subsystems of a standard Linux kernel running under heavy disk load[1]. The distribution of the lengths of these periods is shown in Table 4. It should be noted that these numbers may vary substantially with the hardware configuration. They also vary with the load and the amount of stress that it places on the various kernel sub-

[1] These tests were performed on an 200 MHz Intel Pentium(R) Pro machine

	Kernel		User	
Mode	**99.5%**	**Max**	**99.5%**	**Max**
Focused	19μs	192μs	35μs	210μs
Mixed	44μs	590μs	81μs	793μs

Table 3. Differences Between Scheduled and Actual Event Times Using KURT

Subsystem	Min	Mean	Max
Disk	0.35μs	6.50μs	407.62μs
Memory	0.34μs	0.60μs	10.31μs
Network	0.33μs	0.37μs	3.26μs
Process	0.33μs	1.11μs	43.66μs
Timer	0.35μs	2.33μs	52.98μs
TTY	0.34μs	0.60μs	8.79μs
Other	0.33μs	0.39μs	82.61μs

Table 4. Distribution of Intervals During Which Interrupts Were Disabled in a Standard Linux Kernel

systems. Also, it is important to observe that the maximum values shown in Table 4 do *not* represent an upper bound on the scheduling distortion introduced by blocked interrupts because several blocking periods can occur one after another with little or no spacing between them. However, they do show that most interrupts are relatively short, and that the disk driver is the most immediate area for further investigation.

As a further test of the scheduling quality, we used a hardware ATM cell analyzer to observe traffic streams generated by ARTS. The observed packet interarrival times closely resembled those that ARTS was told to generate. In fact, with a packet scheduled for transmission every 500 microseconds, 99.7% of all packets arrived within 1% of their scheduled times.

To test the KURT scheduler's periodic mode, we wrote a KURT application that was supposed to run once every 10 milliseconds. We also wrote a version of this application that would use only the timing provided by standard Linux running its FIFO scheduler (which claims to offer soft real-time performance). By comparing the effectiveness of these two applications with respect to the actual length of time between executions, we were able to evaluate the gains in scheduling accuracy provided by KURT.

Tests were performed in the KURT and non-KURT cases using first 10, and then 30 concurrent executions of the application. Beyond 30, new attempts to invoke the periodic application were rejected by the KURT scheduler, since it would not have been able to meet the resulting timing demands. Figure 3 shows the distribution of differences between the desired and actual execution periods. As can be seen, the Linux FIFO scheduler introduces essentially an order of magnitude more scheduling variation than does the KURT scheduler and degrades significantly with increased load. The KURT scheduler shows almost no degradation in performance when the number of concurrent executions is raised from 10 to 30. This is evident from the fact that the lines for these two cases lie almost exactly on top of one another.

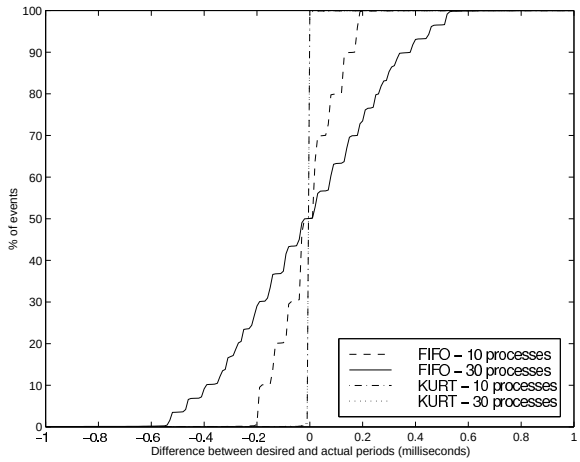

Figure 3. Performance of the FIFO and KURT Schedulers Executing Periodic Processes

5 Conclusions and Future Work

With the increasing popularity and availability of multimedia and high-speed networking, a new class of application software has emerged. The applications in this class require many of the services offered by generic timesharing or soft real-time operating systems, but have timing constraints characteristic of hard real-time applications. Applications of this nature cannot be described as either hard or soft real-time, so we have chosen to call them firm real-time applications. In addition to the unique computing requirements presented by firm-real time applications, many are developed and used under significant budgetary constraints, precluding the use of expensive commercial hard real-time operating systems to support them.

We have developed the KURT system, based on the well-known and widely-used Linux operating system that offers a broad range of services to the applications it runs. We were able to increase the temporal resolution of Linux without causing a significant increase in system overhead. The

new scheduler that KURT adds to Linux is able to achieve high levels of scheduling accuracy and can coexist with the other currently available schedulers. As evidenced by the ARTS application, which makes use of existing ATM networking services offered by Linux, the changes we made do not prevent firm real-time applications from using the existing operating system services. For these reasons, KURT successfully transforms Linux into a firm real-time operating system. In addition to the ability to support firm real-time applications, our system has the added advantage of low cost, since it runs on commercial off-the-shelf hardware and is based on free software.

As the evaluation shows, KURT was able to demonstrate firm real-time scheduling capabilities, although limitations remain, in the form of sources of scheduling distortion. It should be noted, however, that the frequency of distortions in our system is low and falls well within the limits appropriate to a number of interesting applications. Since most distortions occur because the generic operating system services made available to applications by KURT often introduce unpredictable delays, developers need to experiment and discover which kernel subsystems are suitable for use in their applications.

Future work will include investigations into sources of scheduling distortion that degrade real-time scheduling accuracy. The kernel sub-systems that are found to generate this latency might have to be re-designed to work more predictably. The disk driver subsystem is the obvious first choice for this. We will explore the use of alternate scheduling methods within the KURT framework and intend to use the system as the foundation for development of a real-time ORB. It is also interesting to consider how the methods used to implement KURT might be combined with those used for RT-Linux to produce a system capable of supporting and even wider range of applications.

References

[1] LynxOS - Hard Real-Time OS Features and Capabilities. WWW: *http://www.lynx.com/products/ds_lynxos.html*. Obtained November 28, 1997.

[2] QNX Realtime OS. WWW: *http://www.qnx.com/product/qnxrtos.html*. Obtained November 28, 1997.

[3] M. Barabanov and V. Yodaiken. Introducing Real-Time Linux. *Linux Journal*, Issue 34, February 1997.

[4] G. Bollella and K. Jeffay. Support for Real-Time Computing Within General Purpose Operating Systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 4–14, May 1995.

[5] M. Jones, J. Barrera III, A. Forin, P. Leach, D. Rosu, and M. Rosu. An Overview of the Rialto Real-Time Architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 249–256, September 1996.

[6] D. L. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Transactions on Communications*, COM-39:1482–1493, October 1991.

[7] J. Nieh and M. Lam. The Design of SMART: A Scheduler for Multimedia Applications. Technical Report CSL-TR-96-697, Computer Systems Laboratory, Stanford University, June 1996.

[8] B. Srinivasan. A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software. Technical Report 11510-02, Information and Telecommunication Technology Center, University of Kansas, February 1998.