International Journal of Electronics and Communication Engineering (IJECE)
ISSN(P):2278-9901; ISSN(E): 2278-991X
Vol. 5, Issue 1, Dec - Jan 2016, 67-80
© IASET

International Academy of Science, Engineering and Technology
Connecting Researchers; Nurturing Innovations

DESIGN AND REALIZATION OF A REAL-TIME CO-OPERATING SYSTEM FOR MULTIPROCESSOR WORKSTATIONS

AASHISH A. GADGIL
Department of Electronics and Communication Engineering, Gogte Institute of Technology, Belgaum, Karnataka, India

ABSTRACT

I have designed a Real-Time Co-Operating System (RTCS) for simultaneously supporting real-time and non-real-time activities on a workstation with two or more processors. The RTCS is the software equivalent of a co-processor, with software architecture analogous to the hardware architecture that has been used in many workstations and personal computers. In this paper, I discuss a software prototype of the RTCS, which co-exists with Solaris 7 on a four-processor Sun SPARC 20. I summarize the feasibility of the approach through an experimental characterization of Solaris 7. I address the various technical issues involved and present the details of my design. The RTCS is targeted towards real-time applications in the signal processing, sensor- based control, process control, multimedia, and manufacturing domains.

KEYWORDS: Designed a Real-Time Co-Operating System

1. INTRODUCTION

A Real-Time Co-Operating System (RTCS) has been designed for simultaneously supporting real-time and non real- time activities on a workstation with two or more processors.

In this paper, the design issues are discussed, solutions are presented. I provide details and preliminary performance results from first software prototype of the RTCS. The RTCS is targeted towards real-time applications in the signal processing, sensor- based control, process control, multimedia, and manufacturing domains.

The RTCS is the software equivalent of a co-processor, with software architecture analogous to the hardware architecture that has been used in many workstations and personal computers. These computers have a general purpose processor, such as a SPARC, i486, or M68040. In addition, the computer has one or more co-processors which provide a faster and more efficient means for performing specific tasks. Examples of hardware co-processors include floating point units, graphic accelerators, and digital signal processors.

New multiprocessor workstations, such as the SPARC station 20 Model 514MP [15] and the Compaq Pentium-based ProLiant 4000 Multiprocessor [1], have several general purpose processors with a distributed shared memory architecture. By default, Solaris treats all processors as equal.

The goal of the RTCS is to transform some of the general purpose processors into real-time processing units (RTPUs) which can support both hard and soft real-time applications which require a time resolution of less than one millisecond. Processors not assigned as RTPUs continue to execute timesharing UNIX applications, without interfering with execution of real-time tasks on the RTPUs.

1.1 Motivation

The primary objective of the RTCS is to create a real-time operating system environment for desktop workstations and personal computers (both of which are herein referred to as just *workstations*) which will support dynamically reconfigurable and reusable software [12].

Although a workstation is not usually used as the computer of choice for embedded real-time systems, there are many situations for which a workstation is the preferred solution for real-time applications:

Cost: Non-embedded systems, such as many robots, process controllers, and production lines, workstations provide a low cost alternative to expensive real-time hardware.

Performance: Workstations represent the largest market for computing, and hence have the most investment into increasing the speed of the computers. As a result, these computers provide the best performance-to-cost ratio.

Education: Workstations are widely available, often in clusters at a university. This provides the opportunity for improved education through hands-on experimentation along-with real-time systems at the college level, thus better preparing students for the work force.

Software Base: Workstations have a large, evolving software base which uses state-of-the-art technology for such things as editors, debuggers, compilers, software libraries, graphical user interfaces, and CASE tools. They also host distributed file systems, provide security, and offer a variety of networking capabilities. It is highly desirable to leverage as much of this technology as possible for creating equivalent state-of-the-art programming environments for real-time applications, building upon this software base of non-real-time software.

Ubiquitous Computing: There is a technological strive for obtaining a single computer system that can perform all the needs of the users, ranging from non-real-time information system applications to real-time signal processing for multimedia applications such as video conferencing, interactive television, and virtual laboratories. Workstations are the prime candidates for providing the ubiquitous platform, if they can provide the necessary predictability and performance for real-time applications.

There have been several efforts to use off-the-shelf workstation hardware for real-time systems. For example, real-time operating systems (RTOS) such as LynxOS [5] and QNX [8] were designed to use general purpose hardware such as the Intel i486-based computers. However, these systems require that the RTOS take full control of the computer, and completely replace the popular DOS, Windows, or UNIX operating systems. Therefore the workstation can be used for either real-time or non real-time not for both simultaneously.

Other efforts, such as Real-Time Mach [16] and Solaris 5 [15] are UNIX-based operating systems with real-time extensions. There is an attempt to merge real-time tasks with non-real-time tasks in a UNIX environment, by giving the real-time tasks higher priority and performing such procedures as locking pages in memory and reducing the interrupt response time by moving towards microkernel software architecture.

UNIX-based RTOS with real-time extensions, however, have failed to reach the performance or robustness level of RTOS which have been designed specifically for real-time hardware. Successful RTOS include commercial software such as VxWorks [17], OS-9 [7], and VRTX [6], as well as research projects including Chimera [13] and the Spring Kernel [9]. These systems all require dedicated computers, generally in the form of single-board-computers in an open architecture backplane. Due to the complexity of these setups, their availability is low and cost is relatively high as compared to the cost of workstations with similar computational power. In some cases, such as with the Spring Kernel, custom hardware is recommended in order to handle the high operating system overhead to provide guaranteed hard real-time performance [9].

The approach taken here, is distinctly different from any of the approaches listed above, as it does not require taking over the complete workstation (such as QNX, VxWorks, Chimera, and Spring), nor is it obtained by modifying or providing extensions to an existing non-real-time operating system (NRTOS).

The design proposed is based on the co-existence of two separate operating systems that share the same architectural platform and co-operate in order to provide simultaneous real-time and non-real-time support.

The expected performance and predictability of the RTCS more closely resembles that obtained by RTOS with dedicated hardware, such as QNX, VxWorks, Chimera, and the Spring Kernel. However, the RTCS is designed within the constraints imposed by the NRTOS, and thus performance is achieved without the same flexibility as available for the dedicated RTOS. A preliminary performance analysis of the RTCS convinces me that it can support real-time tasks on a Sun SPARC station 20 with frequencies over 1000 cycles per second and with accuracy in the tens of microseconds.

An additional major challenge for supporting real-time on workstations is to provide a software programming environment targeted towards the inexperienced users, rather than requiring highly specialized real-time system engineers to be the only programmers of the system. To address this issue, the RTCS is being designed to support software assembly through visual programming which is based on the underlying design of dynamically reconfigurable and reusable software components.

In the next section, many of the design issues and the constraints imposed by the NRTOS are presented and I introduce some of the solutions. The technical rationale for these design decisions and solutions is in Section 3.

2. DESIGN ISSUES AND APPROACH

My research objective is to provide the necessary set of RTOS services required to support the predictable execution of reconfigurable real-time tasks on a workstation, without sacrificing any of the workstation’s non-real-time functionality.

Depending on the application domain, some real-time tasks may have frequencies greater than 1000 cycles per second, other tasks may require steady communication streams of several megabytes per second, while yet other tasks may interact with advanced graphical interfaces and have soft real-time requirements. Every type of real-time tasks has been considered.

Here, I demonstrate the feasibility of the RTCS approach, then present many of the technical issues that have been addressed in the design of the RTCS.

2.1 Feasibility of Workstation Solution

To determine the feasibility of using the co-operating system approach to obtain improved real-time performance and predictability on a workstation, I performed a detailed experimental characterization of the Solaris 7 kernel on a 4-processor SPARC station 20, Model 514MP.

The results are summarized in the graphs shown in Figures 1 through 8, and demonstrate the potential of the co-operating system approach. The graphs show the execution time of each cycle of a task τ which computes 100 whetstones per cycle for 500 cycles, under various conditions.

Figure 1 shows the execution time for τ when executing as a non-real-time process on a single processor, in the presence of a typical load in the system. A typical load includes running X windows and a variety of other I/O and CPU-bound jobs. There occurs a large swing in execution time.

Figure 2 shows the same workload, but with τ scheduled as a real-time Solaris thread. There is a drastic improvement in the execution time of the task. Execution time, however still varies that can be seen more clearly in the same but rescaled graph shown in Figure 3. In some cases, the task takes an additional millisecond or more to execute its cycle, due to interrupts and other operating system functions executing at higher priority than the real-time tasks. These interrupts are often side effects of *lower-priority* tasks. For example, a non real- time process performs a file system *write*, and the processor receives a high-priority interrupt when the *write* is complete, thus interrupting execution of the real-time thread.

For comparison, the ideal execution of task τ is shown in Figure 4. This theoretical best case represents a mathematically generated result if task τ executed in exactly 3.7 msec every cycle, without any processor contention from interrupts, the system clock, or other threads. Figure 4 is *not* an experimental result.

When all four processors are enabled with the Solaris kernel running on each one, real-time execution is not improved, as shown in Figure 5. Solaris load-balances its high-priority interrupt handling and scheduling operations and the system clock continues to interrupt all processors. This graph shows that simply adding processors to a workstation does not improve real-time performance and predictability.

A significant improvement can be obtained by binding the real-time task τ onto one of the processors, and preventing all other processes from using that processor. The result of this binding is shown in Figure 6. The drawback of doing this in Solaris is that you can then only place one process per additional processor, and thus only run three real-time tasks on a four-processor system. Nevertheless, this graph demonstrates a result that can be applied to the design of my RTCS, as described later.

The small spikes seen in Figure 6, which are approximately 80 τ sec each, are a result of the system clock still interrupting the processor every 10 msec, and load balancing the Solaris dispatcher and scheduler on every fourth system clock interrupt.

The interrupt overhead shown in the graph in Figure 6 can be eliminated by removing this processor from the pool of processors that can be used for scheduling tasks on the workstation. The result, which is very close to the ideal case shown in Figure 4, is shown in Figure 7. The main difficulty encountered with this case is that it requires a loadable kernel module to implement this feature, and requires super-user permission for installing the module. As a result, it is not a viable option for most Solaris users. This approach, however, can be used to install an RTCS microkernel, and forms part of my design.

To obtain Figure 7, only one Solaris process was bound to the processor and all remaining processes prevented from using this processor as a result of which, there is never a need to call the local scheduler. If lower-priority processes are allowed to use the processor, then even though this processor is not used for running the Solaris dispatcher, the local scheduler can be called, only to realize that there is a real-time process executing. As a result, the real-time process may get interrupted, with the result shown in Figure 8. It is therefore desirable to bind and execute only one real-time Solaris process per processor.

These graphs reveal some of the capabilities and limitations of using the workstation for real-time execution, and also show what must be done to obtain predictable execution.

The design of this RTCS is based on the conditions that produce the predictability shown in Figure 7. The RTCS microkernel executes as the only process on each RTPU. It performs its own thread management and uses in-process scheduling, similar to the methods used in creating the kernel for dedicated RTOS such as Chimera, VxWorks and spring. The technical details of this approach are given later in Section 3.

2.2 Technical Issues

I have addressed many issues relating to the various aspects of design of any operating system. The remainder of this section presents these issues and an overview of my approach. Technical details of the solution for many of these issues are given in Section 3.

Memory Management: There are several memory-related issues which have been addressed, including dealing with the cache, virtual memory, and address spaces in a multiprocessor environment. The RTCS deals with the SPARCstation's 1MByte cache by treating it as a local memory, and limiting the size of real-time tasks which collectively execute on a processor to less than 1MByte. Virtual memory is circumvented by locking all cached and shared memory pages into real memory. Multiprocessor pointer addressing in the RTCS is simplified by using a uniform 32-bit address space for all RTPUs.

Communication: Solaris 7 only has non-real-time inter process communication (IPC) mechanisms. Although POSIX-compliant interfaces for real-time mechanisms are defined in the Solaris documentation, there currently is no implementation.

To obtain predictable IPC, the Chimera IPC package was ported to my RTCS. It includes mechanisms such as spinlocks, remote semaphores, prioritized typed message passing, and global state variable tables. Express mail is also implemented as the underlying IPC mechanism between the RTCS micro kernels executing on each RTPU. The RTCS IPC mechanisms are further discussed in Section 3.2.

Resource Sharing: The file system, display, network, and other workstation hardware must all be shared by both the NRTOS and RTCS. However, it is not acceptable to have real-time tasks block and wait for non-real-time tasks. I address this issue by adopting the global/local operating system separation as designed into the Chimera RTOS [13], and using express mail and remote procedure calls to implement system calls and an extended file system. I also improve upon the Chimera design by implementing the servers as multiple real-time Solaris threads on the NRTOS processor, as described in Section 3.3.

Scheduling: Real-time scheduling is always an issue for obtaining predictable execution. My novel design used a loadable Solaris scheduling module; but due to the reasons described in Section 2.1, I could not obtain the desired predictability, nor obtain a resolution better than the Solaris 10 msec system clock. In my revised design, I opted for in-process user-level scheduling with a policy/mechanism separation, which provides us with many advantages, as discussed in detail Section 3.5. A maximum-urgency-first scheduler [10] is used as the default RTCS scheduler, as it provides the maximum flexibility by supporting static, dynamic, and mixed priority scheduling, as well as support for guaranteed scheduling of both hard and soft real-time tasks.

The RTCS scheduler also provides handling and deadline failure detection. Continuous monitoring of real-time tasks is provided through an automatic task profiling mechanism with better than 10 $_\tau$ sec resolution [11], as described in Section 3.6.

Process Management: Solaris provides process management through lightweight processes (LWP). Each process allocates a set of LWPs used to schedule the threads it creates. The Solaris operating system only manages the LWPs and not the threads within a process. A LWP, upon being scheduled, chooses the highest priority thread from the set of runnable threads and executes that thread. Solaris only guarantees that enough threads will be active so that the process can continue to its progress, which is why many Solaris applications are designed with a one-to-one mapping between LWP and threads. These *bound threads* guarantee that a LWP will be available to execute the thread as soon as it becomes runnable.

As highlighted in Section 3.5, more than one LWP on a processor reduces the predictability of the real-time tasks. Therefore, the RTCS microkernel defines only a single LWP per processor, and manages its own threads internally, independent of Solaris. Solaris is instructed never to disturb the RTCS LWP running on the processor except when the RTCS is directly affected by an event, such as timer interrupt signaling a missed deadline.

Clocks and Timers: The RTCS must be designed within the constraints of not only the hardware timers and clocks available, but also within the constraint of how the NRTOS has them programmed. In Section 3.5, I present a solution that maximizes timer resolution while minimizing overhead resulting from timer interrupts.

Global Error Handling: The RTCS uses the Chimera *global error handling* facility for fault detection and handling.

Solaris uses the standard UNIX mechanisms for error handling. However, as discussed in [11], it is preferable to support global error handling, such that exception handling code can be separated from the main code. Global error handling also removes much of the redundant error checking typical in almost all software systems, hence is required to support the design of component-based software. The Chimera RTOS already supports global error handling, which ensures that return codes from system calls are always checked and errors handled appropriately. This improves application development and debugging times by ensuring proper error handling and reducing the amount of code needed to handle errors. If the default error handling is not sufficient, the application developer can specify the handler for a specific error or class of errors on a per module or per-scope basis. This results in error handling code being separate from the main thread of execution, and allows error handling to be modularized and reused.

Portability: One of the major issues in developing the RTCS is to create a programming environment that has minimal hardware dependence, and does not require custom versions of the NRTOS, that for my software prototype is Solaris 7. To maximize portability, I constrained the design space by not allowing any modifications to the NRTOS kernel or operating system modules. Firstly, this ensures that all applications which run on the NRTOS can continue to run even when the RTCS is installed. Secondly, it ensures that the RTCS can run with any version of the NRTOS without recompilation of the NRTOS. Third, the RTCS never directly access any workstation hardware. Any hardware dependent accesses are implemented using a policy/mechanism separation, allowing the minimum amount of code to be modified if porting to a different hardware platform.

Security: Security is generally not an issue in RTOSs because real-time systems are generally single user applications. In Solaris, creating a real-time process requires super user privileges, which limits the availability of real-time processing to a very few users. The requirement for super user permissions is used to prevent one user from completely taking over all available processing time on the machine. The RTCS package contains only one process which is run with super user permissions. This process promotes itself to the highest priority Solaris real-time task in the system, allocates itself an infinite time quantum and then executes all user tasks with normal user priority. The RTCS also prevents real-time tasks from being created on all of the processors in the system, which ensures that the system is always in an operational state.

Reconfigurable and Reusable Software: the primary objective of the RTCS is to provide a computing environment to support dynamically reconfigurable and predictable real-time applications. First aspect of reusing software is to allow the RTCS to use the same software libraries that are available to Solaris real-time threads. Reconfigurable software at the operating system level is obtained by supporting loadable modules and dynamically bound policy/mechanism separation for operating system communication and scheduling functions. Reconfigurable software at the user level is obtained by supporting the port-based object abstraction defined in the Chimera Methodology [12]. Chimera provides a set of library functions for supporting reconfigurable software, such as the *cfig()* utility, which allows for rapid development of code for reading configuration files. The set of libraries available to the RTCS will be a union of the Solaris MT-Safe libraries and the Chimera libraries.

Graphical User Interfaces: Solaris provides process priority inheritance (PPI) to prevent priority inversion from occurring when a high-priority process and a low-priority process are competing for the same resource. The PPI system works by monitoring the system for cases when a low-priority process is blocking a high priority process by holding a critical resource and automatically boosting the priority of the low priority process to that of the blocked process, hopefully freeing the critical resource sooner than it would normally be freed. This allows, for example, an X-Windows based RTCS real-time process to perform updates ahead of other non-real time processes. The use of the PPI system in the RTCS allows for practical real-time displays and other graphical interfaces.

The RTCS provides access to the standard X-Windows interfaces, allowing applications developers full access to the Solaris platform, without giving up real-time advantages. These real-time windowing facilities are used by Onika to provide a user-friendly interface to developing and executing real-time applications.

Programming Environment: One of the goals of the RTCS is to minimize replication of the programming environment available for software development on the NRTOS. In that respect, the RTCS uses all the same tools, including compilers, debuggers, window systems, and graphical tools as the NRTOS. The RTCS is designed with a message passing interface to the user interface, so that command-line, program, and graphical user interfaces can be interchanged. The support for reconfigurable software and loadable modules also allows the use of software assembly-based interfaces for rapid visual programming of real-time applications [2].

Compilers/Linkers/Debuggers: Since Solaris uses a standard dynamically linked executable format generated by all Solaris compilers, the application developer can use their compiler of choice, whether it is the standard Sun C/C++ compiler, the GNU C/C++ compiler, or another third party compiler, to create software components. External commercial or freeware libraries can be used to develop applications and simply included in the link stage. The use of dynamic linking results in smaller executable files and lower memory usage for running an executable.

2.3 Section Summary

In this section, I described the many issues that were addressed in the design of the RTCS, and gave an overview of the solutions employed in the design and implementation of the RTCS. In the next section, the technical details of my solutions are presented.

3. TECHNICAL DISCUSSIONS

Here, I discuss the technical details of my design, and the advantages and repercussions of my approach, for various aspects of the RTCS.

3.1 Memory Management

Caches: Making effective use of the 1 MB level 2 caches associated with each processor requires preloading each real-time task into the cache and keeping the process size to under 1 MB. Each text and data page in a task running under the RTCS is accessed before execution to ensure that the page is in the cache. Care is taken to prevent pages from being removed from the cache. Proper use of the cache is highly dependent upon the underlying architecture and the RTCS uses a method which provides high performance using the Sparcstation20 architecture.

Virtual Memory: The RTCS does not use virtual memory. The *plock(2)* system call is used to disable virtual memory and page swapping for the RTCS and all associated real-time tasks. Effective cache use limits the RTCS to using 1megabyte of memory, so disabling virtual memory for the RTCS does not unduly affect the memory requirements of the rest of the system. All RTCS memory pages are locked into physical memory and cannot be swapped to disk during the lifetime of the RTCS kernel. All other tasks in the system can continue to use virtual memory as usual. This increases the predictability of the system as well as the overall speed.

Memory Map: All RTCS processes have both a private memory area used to for internal data storage and a shared memory area, common to all RTCS processes. The shared memory area is at the same base address in all of the RTCS processes and is reserved at RTCS start-up by the “mmap” system call. This allows pointers to data in the shared memory areas to be passed between RTCS processes without translation, which improves the speed and flexibility of the system.

3.2 IPC Mechanisms

The RTCS has implemented the Chimera IPC suite, including remote semaphores, dynamically a locatable shared memory, prioritized message queues, and the global state variable table [13]. This decision is based on the fact that, although Solaris has real-time IPCs defined in the manual, in practice, the functions currently return *ENOSYS* ( ). Hence the Solaris IPCs still cannot be used. I therefore had to port the Chimera IPCs to the RTCS. To make the RTCS IPCs more compatible with the POSIX.4 specification for real-time operations, a translation layer has been added to provide compatibility with the IPCs defined in POSIX.4. The Chimera IPCs are a superset of those specified by the POSIX.4 API, which simplified the process of implementing the translation layer. The RTCS IPC mechanisms are implemented on top of express mail.

The IPC implementation uses a policy/mechanism separation that is based on the Chimera reconfigurable device drivers [11]. This separation allows for each of the various IPC mechanisms to be replaced while the system is running. For example, the replacement of the semaphores requires shifting the system to a *safe* state where there is no contention between RTCS tasks for the resources (no blocked *P* operations on semaphores). As long as the system is in this safe state, the IPC mechanism can be replaced or upgraded. This implementation serves as the basis for evolutionary design, where a system that cannot be taken off-line can be upgraded and dynamically modified while on-line.

The generic IPC framework also allows the use of other IPC mechanisms not currently defined or implemented. This mechanism can be used to add user-defined functionality to the RTCS kernel. Note that user-level dynamically reconfigurable modules which are not accessed through the RTCS kernel do not use this mechanism. Instead, they use the support for reconfigurable software, based on the Chimera methodology.

3.3 Resource Sharing

The Solaris kernel is reentrant and thus can have multiple processes (and processors) executing kernel code simultaneously. This allows multiple applications to be inside the file system and networking code (and also other areas of the kernel) while avoiding race conditions. However, this can affect the predictability of real-time tasks. To avoid potential problems with using the Solaris method of permitting direct file system access, the RTCS uses the same global/local operating system separation used by Chimera. Remote procedure calls based on express mail are used to provide networking, file system, and user I/O support for real-time applications. Express mail is a unique message passing mechanism for distributed shared memory systems, which allow the non-blocking communication between real-time and non-real-time operating systems. The mechanism passes all service requests for network or file system access to the NRTOS.

The RTCS concentrates exclusively on providing real-time performance and predictability, while the NRTOS handles those aspects which can compromise the predictability of real-time tasks, such as security related issues and hardware interrupts.

For example, if the RTCS allowed direct file system access, a low priority real-time task could begin a disk access and then be preempted by a higher priority task. Then, when the disk is ready and issues an interrupt, the higher priority real-time process will be preempted to handle the interrupt, even though the interrupt was caused by a lower priority task. The remote procedure calls to the NRTOS prevent this type of priority inversion.

3.4 Interrupt Handling

The RTCS controls the interrupt generation and handling on each of its processors, so that only interrupts which directly affect the real-time application are handled by RTPUs.

In the Solaris system, interrupts are handled by very high priority non-perceptible threads which are members of a special scheduling class. These *interrupt* threads are higher priority than Solaris real-time tasks, and if a processor has interrupt processing enabled, these interrupt threads can run on a processor and preempt even a real-time task. This was the primary reason for the unpredictable execution times for the task that was shown in Figures 3 through 6, and Figure 8. The RTCS uses the Solaris kernel functions *cpu_enable_intr* and *cpu_disable_intr* to control interrupt processing on the RTCS processors. Access to these functions is provided to the RTCS through a loadable kernel module. The functions prevent Solaris dispatcher from stealing CPU cycles from an RTCS real-time thread. The Solaris dispatcher services are not needed by the RTCS because the RTCS is the only Solaris process on each RTPU. The processing of interrupts from added sources, such as the hard disk or mouse, is also not allowed. Controlled use of these and other interrupt-related functions allows the RTCS to provide low latency, highly predictable services.

3.5 Scheduling and Timing

The original design of the RTCS used the Solaris dispatcher in combination with a MUF scheduler loadable kernel module to control all task switching between RTCS threads. The loadable scheduler implemented a new scheduling class which was given a higher priority than Solaris real-time threads. In other words, threads of the RTCS class could preempt even threads in the Solaris real-time class. This was done to ensure that when an RTCS process was "bound" to a processor, it would never be preempted by another process in any scheduling class. The binding of the RTCS to a particular processor was done using the Solaris *processor bind (2)* system call.

This design suffered from several limitations, namely accuracy and predictability. Solaris 7 provides only a 10msclock for scheduling, that limits the accuracy and increases the latency for tasks running under the RTCS. In other words, a task asking to run at T = 100 msec will actually run at *T* = 100 msec ± 10 msec. Tasks running at frequencies greater than 100Hz will be hurt by tasks which miss their deadlines because the scheduler will not find out about the missed deadline for (worst case) 10 msec. This low timing resolution and accuracy is not acceptable for most control applications. Even if the RTCS defined a custom Solaris scheduling class, the task execution is not predictable; it would result in real-time execution that resembles the output shown in Figure 4.

My current design of the RTCS scheduler uses user level context switching and an alpha-version of the *timer14*device driver [3] developed and supplied to us by Sun Microsystems to provide high-resolution, low latency task switching in the RTCS. The RTCS is still bound to a specific processor, as it was in the previous design. It executes as the highest priority real-time task in the system and has been given an infinite time quantum. This is done using the *priocntl (2)* system call. To Solaris, the RTCS appears to be a single process with a single thread of execution, with one RTCS process running on each RTPU, but never on processor 0 which also executes the NRTOS. The task switching is done using the Solaris *getcontext (2)*, *setcontext (2)*, and *makecontext (2)* system calls. User-level context switching allows for very low overhead context switches and fast response times when a formerly blocked high-priority task is moved onto the ready-queue. Very high accuracy (timings accurate to within 10 microseconds) data collection for automatic task profiling is also made possible with this scheduling mechanism. Automatic task profiling is described more below.

The source code to this device driver, which is to be distributed with Solaris 8, has been modified and provides the RTCS with a loadable kernel module to generate timer interrupts with microsecond resolution. The RTCS scheduler has been designed to program the *timer14* driver only when necessary, and thus the RTCS is interrupted only when process rescheduling is needed. By not having a periodic clock tick, I avoid the periodic noise shown that was shown in Figure 5.

This design allows the RTCS to provide support for tasks running at 1000Hz, which was one of the design criterions for the system. All hardware dependent portions of the system are encapsulated in one small set of files and can easily be replaced when moving the RTCS to other platforms. The usage of this design improves the speed and portability of the system.

The RTCS scheduler is implemented using the policy/mechanism separation which is the basis for Chimera reconfigurable device drivers. The implementation of the scheduler uses Solaris dynamic loading to allow for safe, dynamic replacement of the RTCS scheduler while the system is running.

3.6 Automatic Task Profiling

The RTCS has been designed to provide continuous monitoring of real-time tasks, based on the automatic task profiling (ATP) designed for the Chimera RTOS. The RTCS version of ATP provides 5Ț sec measurement accuracy, as opposed to the 1 millisecond accuracy in the Chimera implementation. The RTCS ATP is implemented as part of the scheduler, since the scheduler always knows which task is executing at any given time and is responsible for switching task contexts. Data points are also taken upon entry to and exit from interrupt handlers, thus providing for accurate readings even in the presence of interrupts.ATP is an extremely useful RTOS feature because one of the assumptions made by most real-time scheduling algorithms is that the execution time of a task is known. In the general case, manual methods of profiling a task are required in order to determine how long executing a cycle of the task takes, which can be a long and tedious task. If changes are made to the code then profiling must be performed again. In the past, task sampling has been used to obtain task profiles. However, the results are not necessarily accurate due to the coarse grain of the sampling and using a finer grain results in too much system overhead for the profiling. In this RTCS, the profiling provides statistical feedback to the user, the executing task, and to the on-line schedulers, so that, if necessary, they may adapt to account for the actual execution times of the system, and not the estimated worst-case execution times.

Implementation of the ATP in the RTCS is based on the Solaris *gettimeofday (3C)* system function, which performs a system trap and returns (on a SPARC station 20) the current time with better than 10 microsecond resolution.

4. RESULTS

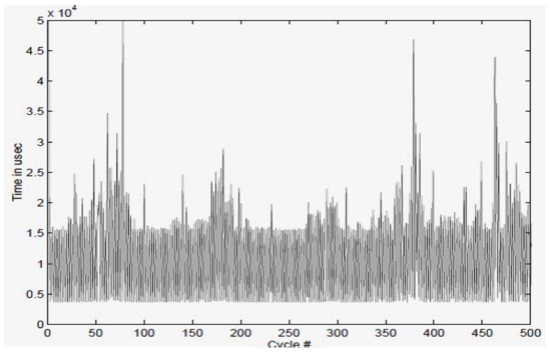

Figure 1: Execution time of each cycle of task τ when scheduled as a Solaris time-sharing thread. Real execution time of task is 3.7 msec.

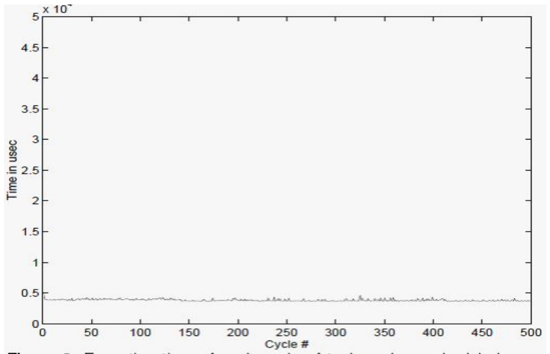

Figure 2: Execution time of each cycle of task τ when scheduled as a Solaris real-time thread on a single processor.

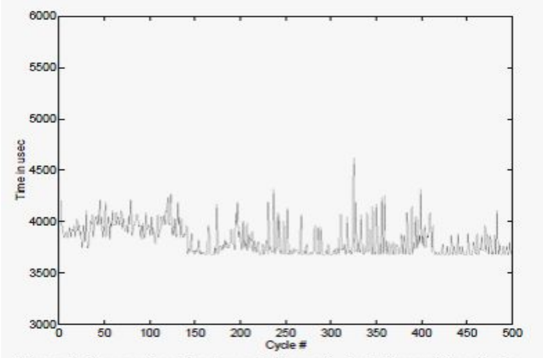

Figure 3: Zoom of graph shown in Figure 2. Note the variation in the execution time when scheduled as a real-time Solaris thread.

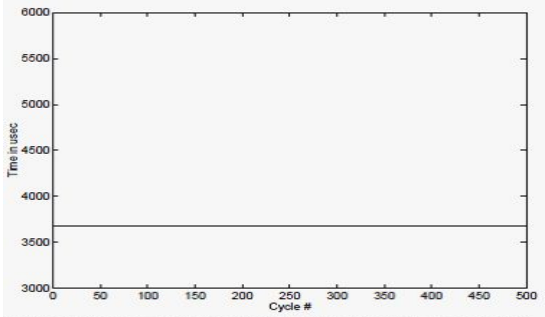

Figure 4: Ideal execution time for task τ. Note this graph was generated based on the real execution time of τ; it was not obtained experimentally.

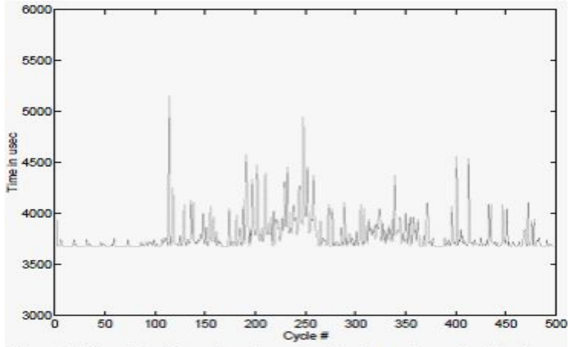

Figure 5: Execution time of each cycle of task τ when scheduled as a Solaris real-time thread in a multi-processor environment.

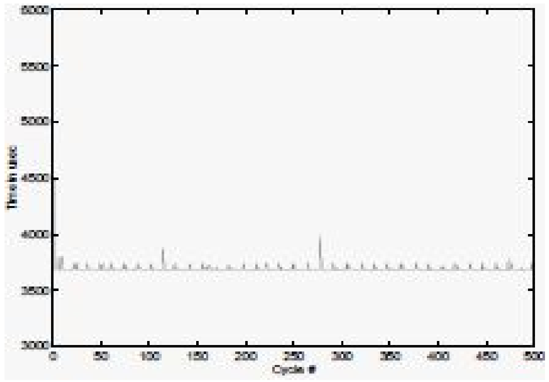

Figure 6: Execution time of each cycle of task τ when executed as only process on a separate processor from the non-real-time threads.

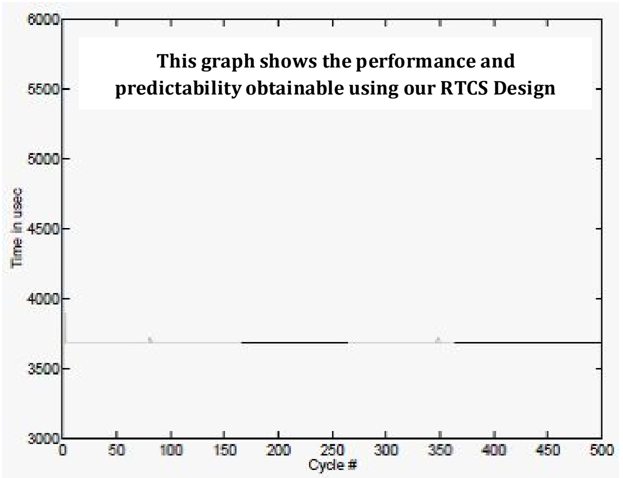

Figure 7: Execution of task τ when executed as only process on a separate processor, and interrupt threads disabled.

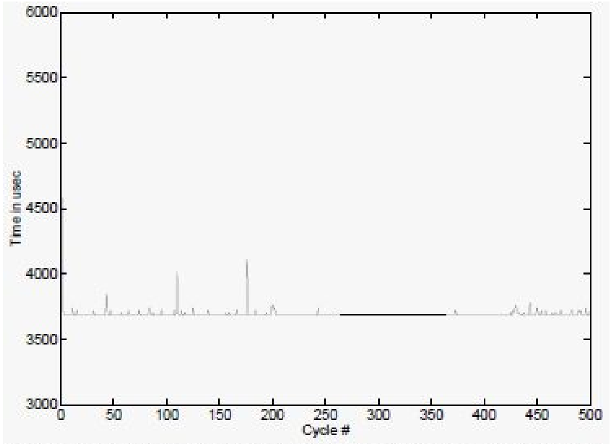

Figure 8: Effect of disabling interrupt threads from processor, but allowing other lower-priority real-time processes to also use the processor.

5. REFERENCES

1. Compaq Computer Corporation, *http://www.compaq.com/.*
2. M. W. Gertz, D. B. Stewart, "A Human- Machine Interface for Distributed Virtual Laboratories," *IEEE Robotics and Automation Magazine*, Vol. 2, No. 4, Dec. 1995.
3. M. Hamilton, "Extending the realtime capabilities of solaris" Sun Microsystems internal working document, April 1996.
4. C. L. Liu, and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment," *J. of the Association for Computing Machinery*, v.20, n.1, pp. 45-61, Jan. 1973.
5. Lynx Real-Time Systems, Incorporation, *http://www.lynx.com/.*
6. Microware Systems Corporation, *http://www.microware.com/.*

7. QNX Software Systems Limited, *http://www.qnx.com/.*
8. K. Ramamritham and J. A. Stankovic, "The design of the spring kernel," in *Proc. of Real-Time Systems Symposium*, pp.146-158, December 1987; *http://www-ccs.cs.umass.edu/.*
9. P. K. Khosla and D. B. Stewart, "Real-time scheduling of sensor-based control systems," in *Real-Time Programming*, ed. by W. Halang and K. Ramamritham, (Tarrytown, New York: Pergamon Press Inc.), 1999.
10. D. B. Stewart, *Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based System*s, Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, PA 15213, April 1996.
11. D. B. Stewart, P. K. Khosla, "The Chimera Methodology: Design of Dynamically reconfigurable real-time software", *Int'l Journal of Software Engineering and Knowledge Engineering*, May 1997.
12. D. B. Stewart, D. E. Schmitz, and P. K. Khosla, "The Chimera II real-time operating system for advanced sensor-based control applications," *IEEE Trans on Systems, Man, and Cybernetics*, vol. 22, no. 7, pp. 1281-1295, November/December 1992.
13. H. Tokuda and C. Mercer, "ARTS: A distributed real-time kernel," *ACM Operating Systems Review,* vol. 22, No. 4, July 1989.
14. Sun Microsystems, Inc., *http://www.sun.com/.*
15. H. Tokuda, P. Rao and T. Nakajima, "Real-time Mach: Towards a predictable real-time system," in *Proc. of the USENIX Mach Workshop*, Nov. 1991.
16. Wind River Systems, Incorporation, *http://www.wrs.com/.*

AUTHOR BIOGRAPHY

Prof. Aashish A. Gadgil received his B.E in Electronics and Communication Engineering and has done M. Tech in Computer Science from Vishvesvaraya Technological University, Belgaum. He is presently working as Assistant Professor at Gogte Institute of Technology. He is well versed in various computer science aspects. He has research interest in Operating Systems, VLSI and Embedded Systems, Image Processing and Artificial Intelligence. He is graduate student member of I.E.E.E, life member I.S.T.E, C.S.I.