Policy-Independent Real-Time Operating System Mechanisms for Timing Error Detection, Handling, and Monitoring*

David B. Stewart† and P. K. Khosla‡

†Dept. of Electrical Engineering and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742
Email: *dstewart@eng.umd.edu;* Web*: http://www.ee.umd.edu/~dstewart*

‡Dept. of Electrical and Computer Engineering and The Robotics Institute,
Carnegie Mellon University, Pittsburgh, PA 15213
Email: *pkk@cmu.edu*

Abstract*: Most research focusing on timing errors deals with scheduling policies that avoid the errors. Since many of the policies are based on estimated worst-case execution times for each task, reliability is a function of the accuracy of the estimates. As a result, many hard real-time systems are implemented with the dangerous assumption that due to correct design and testing, a missed deadline will never occur. We have designed novel policy-independent mechanisms for detecting and handling timing errors, and for monitoring real-time tasks. The detection and handling requires less than 1 microsecond overhead per reschedule operation, and has a latency approximately the length of one context switch for handling an error. The monitoring mechanism uses 6 microsecond per context switch, and requires only 1 Kbyte of memory per 32 processes in the system.*

1. Introduction

Design and analysis of real-time systems is heavily based on knowing worst-case execution times (WCET) of periodic tasks and aperiodic servers.

Accurately measuring WCET, however, is often difficult and sometimes impossible, for several reasons:

- Interrupts in the system, which either execute longer than expected or occur more frequently than anticipated may steal critical execution time from the highest priority tasks.
- Variations in processing speed due to caching, pipelining, and bus arbitration may alter WCET.
- There is no easy way to accurately measure execution times of embedded code.

As long as scheduling policies are based on WCET, these difficulties in measuring WCET inevitably lead to timing errors in the system. Many of these errors go undetected until more catastrophic failures occur, and others result in the system failing to meet its specifications, but with non-obvious reasons as to the cause of such failures.

We have created low-overhead policy-independent real-time operating system (RTOS) mechanisms, which can detect timing errors, invoke policy-specific handlers, and continually monitor the execution of tasks. The mechanisms can be used with a variety of common scheduling algorithms, and serve as the basis for easily extending these policies to incorporate aperiodic servers, soft real-time tasks, imprecise computations, and adaptive real-time scheduling.

1.1 Objective and Requirements

The objective of the research was to create an RTOS mechanism that can detect and handle timing errors, and provide continuous monitoring of real-time activities, even after the system is deployed. The maximum CPU overhead that can be incurred by the mechanisms is 1% of available processing bandwidth. Beyond that, system designers may complain that the mechanisms are too intrusive.

We feel that the 1% of additional CPU overhead can easily be reclaimed by obtaining more accurate WCET, and using those values to form more liberal critical task sets, in order to operate the CPU closer to its full capacity. Currently, WCET is often overestimated in order to have a safety margin, resulting in an over-designed CPU.

In addition, we had the following constraints:

- Additional hardware tools, such as logic analyzers, *cannot* be used, as they prevent the mechanisms from working after deployment of a real-time system.
- The only internal timer available is the system clock. Although newer computers have high-resolution microsecond clocks, low-cost embedded processors and older computer systems do not have such a capability.
- Maximum available memory is 8 Kbytes of RAM. Beyond that, it may be too costly to use in embedded systems.

Since the focus of our research lab is in designing advanced RTOS technology, we did not limit ourselves to the capabilities of commercially available RTOS. Rather, we considered all design options, such that if necessary, we could recommend how current RTOS can be improved. The Chimera RTOS [14] was used as the testbed.

1.2 Background

There has been extensive research into detection and handling of hardware, software, and state errors. A hardware error is a result of failing hardware or a processor-generated exception. A software error is a mistake either in the design or implementation of software. A state error is a discrepancy between the current state of the system, and the internal data representation of that state.

*Research presented in this paper was primarily performed as part of the Chimera Project. Chimera has been funded in part by NASA, ARPA, Sandia National Laboratories, and the Electrical and Computer Engineering Department and the Robotics Institute at Carnegie Mellon University. Continuing research on this topic is supported by the Electrical Engineering Department and the Institute for Advanced Computer Studies at University of Maryland, College Park.

However, there has been little research into the detection and handling of timing errors. These types of errors, which occur in real-time systems, are failures to meet the timing specifications of the system.

Much research in real-time systems focuses on developing scheduling policies that can be used to guarantee that timing errors will not occur [3].

Unfortunately, these hard real-time systems are often implemented ***with the dangerous assumption that due to correct design, analysis, and testing, a timing error will never occur***. Since many of these policies are based on estimates of the WCET of each task, the reliability of the system is often a function of the accuracy of the estimates. These systems rely on the guarantees of the scheduling policies to meet the timing requirements, and cannot detect missed deadlines, let alone take emergency action in response. Yet errors in these systems do occur, primarily due to the difficulties in accurately measuring WCET of the code.

The most common timing error is the missed deadline. That is, a real-time process has a specific time by which it must finish executing, but it does not finish in time.

Another timing error, which is perhaps just as common but not necessarily as obvious, is the incorrect estimate of execution time. The guarantees provided by a scheduler are only valid if the WCET times used in the schedulability analysis are accurate or over-estimated. What if those estimates are wrong? Executing a real-time process for longer than originally anticipated usually leads to unexpected overloading of a CPU, such that even processes that were guaranteed to meet all deadlines eventually miss a deadline!

To our knowledge, there has not been any prior solutions for policy-independent mechanisms at the operating system level that can be used to detect and handle timing errors.

Several real-time programming languages such as Real-Time Euclid [8], RTC [22], and RTC++ [6], describe policy-independent mechanisms for timing errors. Although the mechanism is primarily part of the language, they still depend on a real-time kernel to detect and notify the language of the timing errors. They also require that all code must be written in that language. An operating-system based mechanism can be compatible with any language, and can be used as the basis for notifying these languages which require kernel support.

Flex [7] is a language extension to C++ that includes built-in timing error detection and handling for imprecise computations, without relying on the operating system kernel, as do the other language-based methods. The mechanism uses a combination of delay calls and alarm functions to implement the detection and handling. This mechanism, however, is tied to the imprecise computation model [11], and, as stated by the authors, slows down typical C++ code 2 to 10 times.

Other efforts to identify timing errors have been limited to using monitoring methods, which are typically only available during the development phase of a real-time system.

Monitoring, also called *profiling*, involves the collection of performance data while the system is running, then analyzing the data either in parallel or afterwards. *Hardware monitoring* is achieved by attaching probes of a logic analyzer to the processor, system bus, or to input/output ports in order to observe the activity and collect the measurement data without disturbing the system [12] [21]. *Software monitoring* is accomplished by adding measurement routines that record important events which can later be evaluated by the software application on the target system [2] [19]. *Hybrid monitoring* is a combined hardware/software solution, in which a logic analyzer with a computer interface is used [5] [18].

Hardware methods have the advantage of accuracy and are less intrusive on a real-time system. They have the disadvantage of being difficult to setup and use, and require special hardware that is only available during system development. Software methods are more flexible and can more easily be automated, but generally provide much less accuracy.

Existing software methods use a process to collect data about executing tasks at regular intervals. This process incurs significant amounts of overhead, as it requires the time for at least two context switches each time it executes. In a real-time system, if this process runs with low priority to only use extra CPU time, then much data about the system is lost if the system is overloaded. If the process runs with the highest priority, it can easily lead to over 25 percent overhead. Consequently, software monitoring has primarily been limited to the performance analysis of non-real-time systems.

We designed a real-time profiling mechanism that uses about 1% CPU bandwidth to provide information at least as accurate as other software monitors. Instead of creating a separate process, the monitoring is built-in to the microkernel.

Chodrow et al. extended their software monitoring technique to support timing error detection and handling [4]. They incorporated monitoring tools into the real-time system as a separate process, with a message queue connection to every other task in the system. A real-time task sends a message to this monitor whenever it meets one of its timing constraints. The monitor maintains a database of these timing constraints, and expects to receive the messages. If it does not see the message, then it sends a signal to the task indicating the timing constraint violation. The signal handler of the task is a timing error handler.

The operating system overhead required to implement this technique is enormous. The monitoring task must be executing in order to detect a timing failure of another task. In a uniprocessor environment, it means there is always a significant delay between the time the error occurs, and the time the failing task is signalled, considering that at least one signal and two context switches are required between the detection and handling of the error. When no error occurs, there is still one message transferred per task per cycle, indicating that a timing constraint was met. The authors of this work did not provide any performance benchmarks on their method, therefore we cannot quantify the overhead.

The authors also attest that there is significant complexity of initializing and running this setup, primarily due to the need for a large numbers of message queues and setting up of a hash table in order to optimize the code which continually checks the timing constraints of every task.

We took a different approach than Chodrow, by creating separate mechanisms for timing error detection and handling, and monitoring. As compared to Chodrow's technique, our mechanisms can produce more accurate results with significantly less operating system overhead, thus less intrusion on the real-time system, and with minimal effort on the part of the programmer, since the mechanism is part of the RTOS.

Our mechanism for detecting and handling timing errors is described in Section 2. The mechanism for automatically monitoring real-time processes is described in Section 3. Sample policies, which demonstrate the use of the mechanisms, are presented in Section 4. Finally, in Section 5, we summarize our work and discuss future directions.

2. Detection and Handling of Timing Errors

The most common timing error that must be detected is a missed deadline. In many cases, the deadline is the start of a periodic task's next period. With some scheduling policies, such as the deadline monotonic algorithm [1], the deadline may be earlier than the start of the next period. The mechanism we developed is flexible to handle either of these cases.

The other timing error is incorrect estimates of a task's worst-case execution time (WCET). The schedulability analysis for many scheduling algorithms requires knowing a task's WCET. However, due to the difficulties of getting accurate measurements, it is possible to over- or under-estimate this value. Over-estimating is not as critical, as it leads to under-utilized systems. Under-estimating the value, however, can lead towards incorrect operation, and any execution for hard real-time tasks are at risk. To increase the reliability of real-time systems, it is therefore necessary to both obtain accurate estimates of execution time for analysis, and to immediately detect an under-estimated WCET.

2.1 Programmer Interface

The programmer specifies the deadline and maximum execution time whenever a task pauses to wait until the start time of its next cycle arrives. The Chimera framework for a periodic task is shown in Listing 1.

The system call which interfaces with the Chimera microkernel is *pause(float restart, float exectime, float deadline)*. It program's one of Chimera's virtual timers to wake up the task at time *restart*. Time is specified in seconds. When the task begins executing its next cycle, it will be allowed to use a maximum of *exectime* seconds of the CPU. It must also complete its execution before *deadline*, which is specified in seconds relative to the start of the task.

```
float Ta = 0.1;/* period of task, in seconds */
float Ea = 0.04/* estimated WCET of task, in seconds */
float nextstart;/* start of next cycle */

main(args) {
     /* initialization stuff goes here */
   nextstart = clock();/* store 'now' into nextstart */
   while (1) {  /* begin periodic loop */
      nextstart += Ta;/* compute next restart time */
      pause(nextstart,Ea,Ta);
      {execute one cycle of task here;}
   }
}
```

Listing 1: Framework for periodic tasks in Chimera.

In the example in Listing 1, the maximum execution time E_a is 40 msec (note, *msec*=millisecond, μ*sec*=microsecond), and the deadline in this case is the start of the next cycle, thus the deadline is set to the task's period T_a. Although in the example the period and execution time are hard-coded, it is more typical to pass these values as arguments to *main()*. In addition, the static priority of the task is set when the task is created by a parent task. However, as will be discussed in Section 4.1, the static priority can be temporarily modified for failure handling and execution of aperiodic servers.

Timing constraints in Chimera are specified using a policy-independent interface, so that it can be used with both static and dynamic scheduling algorithms.

2.2 Timing Error Detection

Timing errors are detected jointly by the scheduler and microkernel. In this section, the implementation of the timing error detection in the Chimera RTOS is described.

The scheduler tracks missed deadlines by monitoring the earliest deadline among all tasks on the READY and BLOCKED queues. The running task is always on the READY queue. The BLOCKED queue is for tasks waiting on a semaphore or for a message. Tasks awaiting their next re-start time are on a separate PAUSE queue and do not yet have deadlines.

The scheduler programs the earliest deadline into one of the microkernel's virtual timers, used to monitor missed deadlines. If the current time reaches this programmed time, then the microkernel calls the scheduler using the missed-deadline entry point, and retrieves which task(s) missed a deadline. It then invokes a handler for those failing task(s).

In order to control maximum execution time, the microkernel maintains a software down-counter. When a task begins to execute, the maximum execution time it requires is copied into this counter. On every clock tick, the counter is decremented. If it reaches zero, then the task has used up its allotted CPU time, and a timing error has occurred.

If a task blocks or is preempted during execution, then the value in the down-counter is saved as part of the context of the task. When the task is swapped in again at a later time, the down-counter value is restored to its saved value.

A task's cycle is considered complete when it calls the *pause()* routine again. At this time a new deadline for that task is specified, and the maximum execution time for the task's next cycle is replenished. If the task's old deadline was programmed into the virtual timer, then the scheduler finds the next earliest deadline and re-programs the timer. Otherwise, the timer continues to run, as the task with the earliest deadline is not the executing task (this situation should not occur if the EDF algorithm is used, though).

2.3 Handling Timing Errors

When a timing error is detected, a user-defined *timing failure handler* (TFH) is invoked. The TFH executes within the context of the failing task. Failure handlers can perform any kind of recovery operations, such as aborting the task and preparing it to restart the next period, continuing the task and forcing it to skip its next period, sending a message to some other part of a distributed system, performing emergency

handling such as a graceful shutdown of the system or sounding an alarm, or, in the case of iterative algorithms, returning the current approximate value regardless of precision.

Since a TFH can be called at any time, it must be designed as *re-entrant* code, similar to that used for interrupt and signal handling. It saves part of the task's context such as scratch registers, so that the code can be called at any time, without compromising the integrity of the main body code.

High level languages such as C do not provide the necessary functions for creating re-entrant code. As a result, this code must be written in assembly language, and it is usually non-portable. In Chimera, a generic assembly language timing error handler was created. This code then performs a jump-table lookup, and calls a C-language subroutine corresponding to the task which has the timing error.

The user creates a TFH by defining a subroutine, then installing it using the *tfhInstall(funcptr handler,int hpriority)* routine. The *handler* argument is the name of the subroutine, while the *hpriority* is the static priority at which the failure handler is to be executed.

The *hpriority* parameter is used to temporarily modify the priority of the task during execution of the handler. This feature allows the failure handling code to have a priority that is different from the failing task. Thus critical failure handling can have high priority and can be called immediately, while failure handlers for soft real-time tasks have lower priority, and do not use up execution time of other more critical tasks. If *hpriority* is 0, then the handler is executed with the same priority as the failing task.

An example of defining and installing a TFH is shown in Listing 2. The argument passed to the failure handler is the type of failure, which is either DEADLINE or MAXEXEC. This allows each task to have separate handling for the *missed deadline* and *maximum execution time used up* timing errors.

In this example, the TFH is designed to send a message to another task and prepare the task to restart execution if a missed deadline occurs, and gracefully shuts down the system if the task uses more than it's allotted WCET.

An issue in designing the mechanisms for supporting TFHs is to do so within the scope of the failing task. Using the same mechanism as interrupt handlers is not sufficient. Interrupt handlers execute within the scope of the kernel, and not of any task. As a result, they do not have access to most of a task's data. Also, the failing task may not necessarily be the task that is currently running. In particular, for the missed deadline timing error, it is likely that the task is not executing, but rather is on the READY or BLOCKED queue.

In the Chimera implementation of the TFH, the microkernel modifies the stack and program counter of the stored context. The current program counter is added to the stack, and the stack pointer adjusted accordingly. The program counter is then modified to contain the start address of the generic TFH re-entrant code. As a result, the next instruction executed by the task will be the TFH. The priority of the task is also modified to *hpriority*, and a re-schedule is performed if necessary. If the failing task now has the highest priority in the system, it will be the next task selected. Otherwise, the TFH will be executed only when it is that task's turn to execute, based on the real-time scheduling algorithm in use at the time.

Modifying a task's stored context works for all cases except when the running task fails. In this case, the running task is swapped out first, then the above modifications are made. If the task still has highest priority, then it is swapped back in, otherwise, the next highest priority task is selected.

After executing a TFH, there are three possibilities to resume normal processing: restart, continue, or exit. The restart option causes the task to return to the beginning of its code, using a UNIX-like *setjmp()/longjmp()* mechanism. The *continue* option allows the task to keep executing from before its code was interrupted by the TFH. This is accomplished by simply doing a *return()*, since the old PC was saved when the task's stored context was modified. The *exit* option is used to kill the task altogether.

Since a failure handler can be called at any time, it might be called when the failing task is writing data in a critical section. In such cases, execution of the TFH must be delayed until the end of the critical section, to ensure the integrity of the data. This can be accomplished by locking out the TFH during such critical sections. The task continues to execute at the same priority to the end of the critical section, unless the failure handler is defined to have a higher handler priority, in which case the task immediately inherits the higher priority. This in effect extends the definition of the TFH to implicitly include a "finish writing data first". Chimera provides a facility for locking out a TFH, or such code can be called by an RTOS's semaphore or condition variable mechanisms.

If a task is on the BLOCKED queue, it is moved to the READY queue, with the TFH being the next instruction to execute. In this case, if the *continue* option for the task is selected, the task returns to the BLOCKED queue.

The latency to call a TFH is the length of one context switch, plus 12 μsec (on a 25MHz MC68030) for updating the stored

```
float Ea = 0.05;            /* in seconds */
float Ta = 0.1, nextstart;
int Ph=100;/* highest priority in system */
extern MSG *msgq;/* message queue for IPC */
jmp_buf restart;

tfhandler(int type) {
   switch (type){
   case DEADLINE:  /* missed deadline */
      msgSend(msgq,"no more time,no data this cycle");
      longjmp(restart);
      return;}
   case MAXEXEC:   /* effective CPU time used up */
      shutdown(); /*emergency shutdown;doesn't return */
      }
   } /* end switch */
}
main(int Pthis) { /* Pthis is priority of this task */
   { initialization stuff goes here }
      /* execute handler with highest priority */
   tfhInstall(tfhandler,Ph);
   nextstart = clock();
   while (1) {       /* begin periodic loop */
      setjmp(restart);
      nextstart += Ta;
      pause(nextstart,Ea,Ta);
      { execute one cycle of task here; }
   }
}
```

Listing 2: Example of a TFH in Chimera.

context and executing the generic assembly language framework. This overhead is incurred *only* when a timing error occurs. The overhead of decrementing the down-counter on each clock tick is less than one microsecond.

3. Automatic Task Profiling

The automatic task profiling (ATP) mechanism is a software monitoring technique built in to the RTOS. It collects run-time data, and eliminates the need for a developer to use other hardware or software monitoring.

The mechanisms described in the previous section are useful for handling timing errors immediately, but they do not aid developers to address the more fundamental issues of why the timing error occurred in the first place.

The ATP that we have designed can measure execution time, maintain a summary of the number of cycles and missed deadlines, and compute the minimum, average and worst-case CPU utilization for every task in the system.

ATP is accomplished without the need for any additional hardware. The philosophy in designing the ATP mechanism is that the RTOS always knows which task is executing; it should keep track of the execution time used by each task, and to monitor that the task has completed execution before the deadline time arrived.

The Chimera microkernel was modified to maintain statistics every time a context switch is performed. The statistics are made available immediately via shared memory to soft real-time tasks executing on the same or remote processors, or to GUI's with shared memory access to the system.

The key to minimizing the run-time overhead is to identify and store only a finite amount of raw data during each context switch. From this data, a task that reads the shared data can create a more detailed profile of the task's execution. The granularity of the raw data is the same as the system clock, which by default in Chimera is 1 msec.

The following data is collected for each task τ_k:

$n_c[k]$	total number of cycles executed (for periodic tasks) or events processed (for aperiodic tasks).
$n_m[k]$	total number of missed cycles.
$q_c[k]$	cumulative clock ticks for this cycle or event
$q_t[k]$	total clock ticks executing
$q_{max}[k]$	maximum ticks used for one cycle or event
$q_{min}[k]$	minimum ticks used for one cycle or event

The memory required is less than 32 bytes per task (using 32-bit integers to store each value). The mechanism needs only 1 Kbyte of memory per 32 processes in the system.

Instrumentation code is added to the beginning of the context switch code, where task τ_k has just finished executing and is being swapped out.

The pseudo-code for incorporating automatic task profiling into Chimera is shown in Listing 3. It demonstrates the data collection algorithm.

The pseudocode was implemented in assembly language at the beginning of the Chimera context-switch code, and require 6 μsec to execute on a 25MHz MC68030. The mechanism uses less than 1% CPU bandwidth when there is an average of 16667 or less context switches per second.

In line 2, l is assigned the number of clock ticks since the last context switch; t is the current time (in clock-ticks) and t_0 is the time of the last context switch. In line 3 the time of this context switch is stored. In line 4, l is added to the cumulative number of clock ticks used by task τ_k, $q_0[k]$. If this context switch is a result of a periodic task ending its cycle or an aperiodic server finishing the processing of an event (as opposed to a task being pre-empted by a higher-priority task), then there is further information stored. The number of cycles that task τ_k has executed is incremented.

If it is the first cycle of the task (or first event to be processed by an aperiodic server), then the cumulative value is dropped and the minimum and maximum execution times are reset (line 8). The first cycle is ignored because the profiling can be reset at any time by the user, and all profiling is done relative to that time. If this is done while tasks are running, then partial information about that task in its first cycle may be lost, and thus the information obtained when a task completes its first cycle after the profiling is reset it not accurate.

If it is not the first cycle of the task, then a check is made to update the minimum and maximum cycle times of that task (lines 10-12). The total number of clock ticks used by the task is also maintained in the variable $q_t[k]$.

Missed deadlines are also computed as $n_m[k]$, except that they are counted separately in the kernel by the timing error detection code described above.

The profiling can be set to execute for any arbitrary time of Q_{total} clock ticks, such that $Q_{total} = Q_{end} - Q_{start}$. Given these three values, the system clock rate in seconds T_{sys}, and the raw data obtained during each context switch given above, the following information can be computed on demand for each task in the system:

$ptime = T_{sys} \cdot Q_{total}$	The profiling time (in seconds), which is the amount of time elapsed from beginning to end of profile.
$cycle = n_c[k]$	Number of cycles started.
$miss = n_m[k]$	Number of missed deadlines.
$mezF = \frac{n_c[k]}{ptime}$	Measured frequency: number of cycles per second. This value should be the same as the desired frequency, unless there are missed deadlines, in which case it will be lower.

```
1:  Local l
2:  l = t-t0
3:  t0 = t
4:  qc[k] += l
5:  if (end_of_cycle_or_event) {
6:     nc[k]++
7:     if (nc[k] == 1) {
8:        qt[k]=0, qmin[k]=infinity, qmax[k]=0
9:     } else {
10:       if (qmin[k] > qc[k]) qmin[k] = qc[k]
11:       if (qmax[k] < qc[k]) qmax[k] = qc[k]
12:       qt[k] += qc[k]
13:    }
14:    qc[k] = 0
15: }
```

Listing 3: Pseudo-code for incorporating automatic task profiling into Chimera microkernel.

$mezT = \frac{ptime}{n_c[k] + n_m[k]}$	Measured period of the task, as detected by the automatic profiling. This equation implicitly rounds to the nearest multiple of the clock rate.
$totC = q_t[k] \cdot T_{sys}$	Total execution time (in seconds) used by task within the profiling time
$minC = q_{min}[k] \cdot T_{sys}$	The minimum amount of CPU time used by the task in one cycle.
$maxC = q_{min}[k] \cdot T_{sys}$	The maximum amount of CPU time used by the task in one cycle.
$avgC = \frac{q_t[k]}{n_c[k]} \cdot T_{sys}$	The average amount of CPU time used by the task in one cycle.
$minU = \frac{minC}{Q_{total}} \cdot 100$	The minimum percentage of CPU time used by the task in any one cycle.
$maxU = \frac{maxC}{Q_{total}} \cdot 100$	The maximum percentage of CPU time used by the task in one cycle.
$avgU = \frac{avgC}{Q_{total}} \cdot 100$	The average percentage of CPU time used by the task.
$resolution = T_{sys}$	The resolution of the times in the profile is the same as the system clock.

Although the resolution of performing such timing in software through the kernel is much lower than the timings that can be done using specialized hardware, it is often sufficient because the resolution is the same as what is available to the scheduler. For example, if the system clock is set to 1 msec, then the real-time scheduler cannot differentiate between a task that takes 2.3 msec to execute or 2.7 msec. Therefore, the scheduler views each of those as 3 msec tasks, which is the same resolution provided by the current automatic profiling.

4. Timing Error Handling Policies

Given mechanisms for timing error detection and handling and task profiling, it is reasonable to ask what is the best way to use them. In many cases, if a system is designed such that some timing errors are expected, but that proper definition of the handlers can take care of them, the implementation of many scheduling policies becomes straightforward.

The mechanisms are compatible with popular real-time scheduling policies, such as the rate monotonic (RM) [10] and deadline monotonic [1] static priority algorithms; and the maximum-urgency-first (MUF) [15], and earliest-deadline-first [10] dynamic priority algorithms.

In this section, however, we discuss how the mechanisms can be used for more advanced scheduling, often using the above policies as the basis. This includes scheduling aperiodic servers, soft real-time tasks, imprecise computations, and adaptive systems.

4.1 Aperiodic Servers

Aperiodic servers are designed to handle events in a hard real-time system which arrive at random times. Initial implementation of two types of aperiodic servers, the deferable server [9] and sporadic server [13] into the Real-Time Mach operating system [20] required that the scheduler be modified to explicitly support them.

In this section, we show how the timing error detection and handling mechanisms can be strategically used to control the capacity and period of aperiodic servers. As a result, any system which supports these timing error mechanisms can also support the aperiodic servers *without the need to modify any part of the real-time scheduler*.

The aperiodic servers were initially designed for the RM algorithm, and later extended to be used with the MUF algorithm [16]. The implementation of these aperiodic servers is basically the same.

Deferable Server: The basic concept behind the deferable server (DS) is that a periodic task with highest priority is created with a fixed period T_{ds} and maximum capacity C_{ds}. The *capacity* represents the maximum CPU time that the server can use within one period. If more than that amount of CPU time is required, then the server's priority is lowered, until the beginning of the next cycle.

The DS executes with the highest priority of P_h. In comparison, all the periodic tasks which form the critical set would have a priority less than P_h. T_{ds} is equal to that of the highest frequency periodic task that may execute. Its capacity C_{ds} is equal to U_{ds}/T_{ds}, where U_{ds} is the DS size. The server size represents the highest average execution time per cycle that the server can sustain without jeopardizing the execution of any of the periodic tasks in the critical set. The server size can be analytically derived, as described in [13] when using the RM algorithm, and [16] when using the MUF algorithm.

The structure of the main body of a DS is similar to that of a periodic task, and is shown in Listing 4. The maximum execution time of the task is set to the capacity, C_{ds}, and the deadline is set to the task's period T_{ds}. The routine *dshandler()* is the TFH which is called when either the server's capacity has expired, or the server's replenishment time has arrived.

If a timing error occurs as a result of the task using up all it's allotted CPU time (i.e. the error is of type MAXEXEC), then the server has used up its capacity, and must let the other critical real-time tasks execute so that they do not miss their deadline. As a result of using up it's time, the microkernel signals a timing error, and the TFH for the DS is called. The TFH lowers the priority of the task to P_l, where P_l is a lower priority than any periodic tasks in the critical set, but still greater than tasks that are not in the critical set. The server continues to execute only if there is leftover CPU time after the critical set tasks have executed, or if its replenishment time arrives.

The replenishment time of the server is its deadline. When the task's deadline time arrives, the microkernel detects this as a timing error, and again calls the TFH. This time, however, the error is of type DEADLINE, which implies a missed deadline. For the server, however, this is an indication that it's capacity can be replenished. Therefore, the TFH resets the criticality of the server back to P_h, and sets up new deadline and resets the maximum execution time back to C_{ds}. Note that the routine *set_deadline()* is similar to *pause()* in terms of setting the deadline and maximum execution time, but it leaves the task on the ready queue, rather than pausing the task until a new start time arrives.

If using the MUF scheduling algorithm instead of RM, the only difference is that the criticality of a task is adjusted rather than the static priority [16].

Sporadic Server: The sporadic server (SS) is similar to the DS, except that the method in which the task's capacity is replenished is different. In order to improve on CPU utilization and server size, the SS replenishes the capacity only when necessary, and sometimes it is only partially replenished. In contrast, the DS always replenishes its capacity fully at its deadline time. This change results in a more complex aperiodic server, but the gain in CPU utilization and server size might be worth the additional complexity.

The design of an SS is similar to that of the DS, except that instead of just specifying a server's capacity and its replenishment time, a list of execution start times and the amount of execution used must be maintained. The framework for implementing an SS in Chimera is shown in Listing 5.

Initially, the maximum execution time of the SS is C_{SS} and its deadline time set to infinity. This means that the TFH will only be called when the WCET down-counter reaches zero, and not as a result of any kind of missed deadline. Replenishment occurs whenever the server exhausts its execution time. The amount of replenishment depends on when the CPU time was used up, which is maintained in a list.

Whenever an event is processed, the task updates the replenishment time of the server based on when the event's processing began and how much execution time was used. The routines *execstart()*, *execend()*, and *execleft()* are calls which return information about the execution time being used by that task. The information is available to the task via the shared memory information generated by Chimera's ATP mechanism that was described in Section 3.

```
float Cmds = 0.05;          /* in seconds */
float Tmds = 0.1, nextstart;
int Ph=100,Pl=50; /*assume that tasks in critical set */
     /* have priority between Ph and Pl. */

dshandler(int type)
{
   switch (type){
   case DEADLINE:  /* type 1 failure; replenish server */
      set_priority(Ph);
      nextstart += Tds;
      set_deadline(nextstart,Cds,Tds);
         return; }
   case MAXEXEC:/* type 2 failure; capacity used up */
      set_priority(Pl);
      return; }
   } /* end switch */
}
task_name()
{
   { initialization stuff goes here }
      /* execute handler with default priority */
   tfhInstall(dshandler,0);
   nextstart = clock();
   set_deadline(nextstart,0,Cmds,Tmds);
   while (1) {       /* begin periodic loop */
      msgReceive(message); /* wait for event */
      { do event handling here; }
   }
}
```

Listing 4: Framework for a deferable server using a TFH to control the server's execution

4.2 Soft Real-Time Tasks

An obvious use of the timing error detection and handling mechanism is to implement soft real-time tasks.

For example, one scheduling policy may be for tasks that miss deadlines to continue to execute, but to use up the time from their next cycle as well. This in effect temporarily halves the frequency of that one task. This method works well especially for tasks which receive input from analog sensors, where missing an occasional sensor reading is acceptable. The schedulability analysis of a system which contains this type of soft real-time task is given in [16].

Implementation of these tasks in Chimera is the same as for the hard real-time task shown in Listing 1, except that it uses a TFH similar to the one shown in Listing 4 for the DS.

When a MAXEXEC error occurs, the priority of the task is lowered such that it is less than the priority of any task in the critical set. For the remainder of that cycle, it only executes with low priority. When the task's deadline time arrives (i.e. a DEADLINE error occurs), the task's priority is restored, and

```
float Css = 0.05;           /* in seconds */
float Tss = 0.1, nextstart;
int Ph=100;/*highest priority of tasks in critical set*/
int Pl=50; /*lowest priority of tasks in critical set*/

struct qlist {
   float t,c;
   struct qlist *next
} Q[QMAX];
   /* Q[0] is header node */
qList *qHead=&Q[0],*qTail=&Q[0],*qFree=&Q[1];

sshandler(int type) {
     /* both DEADLINE and MAXEXEC types of deadlines
handled same way */
   mexec = execleft();
   while (q->t <= clock() {
      mexec += q->c;
         /* delete entry from list */
      qHead = q->next; q->next = qFree; qFree = q;
   }
   if (mexec == 0) {
         /* no exec time left, go to lower criticality */
      set_deadline(clock(),Infinity,q->t-clock());
      set_priority(Pl);
   } else {
         /* some high priority exec time left */
      set_deadline(clock(),mexec,Infinity);
      set_priority(Ph);
   }
}
task_name() {
   { task-dependent initialization stuff goes here }
   { initialization of qLists goes here }
      /* handler should execute with default priority */
   tfhInstall(sshandler,0);
   nextstart = clock();
   set_deadline(nextstart,Css,Infinity);
   while (1) {       /* begin periodic loop */
      msgReceive(message); /* wait for event */
      estart = execstart();
      { do event handling here; }
      eend = execend();
         /* add entry to list */
      q = qFree; qFree = q->next; qTail->next = q;
      q->t=estart+Tss; q->c=eend-estart;
   }
}
```

Listing 5: Framework for a high-priority sporadic server using a TFH to control the server's execution.

the task can continue to execute with a priority that guarantees some execution.

4.3 Imprecise Computations

Another possible use of our mechanisms is for implementing imprecise computations. This policy provides the means to create approximate computations quickly, and to only fully execute a process if time permits.

A task which can be scheduled using an imprecise computations algorithm typically has two parts, an essential part and an optional part. If a missed deadline occurs while the optional part is being executed, then those optional computations are discarded, and the results of the essential part used.

Scheduling theory using an imprecise computation model is summarized in a collection of papers in [11].

4.4 Adaptive Real-Time Scheduling

In dynamically changing systems, it may not be possible to analyze the schedulability of a task set a priori. Streich developed a policy for adaptive scheduling of soft real-time tasks, called TaskPair scheduling [17]. It is based on the notion of optimistic case execution times, in addition to WCET.

The method requires that a built-in monitor constantly measure execution times of tasks, and that a timing error handler be invoked if the optimistic execution time is used up.

The mechanisms we describe in this paper provide the necessary functionality to implement adaptive real-time scheduling policies such as TaskPair scheduling.

5. Summary

We have designed novel mechanisms for detecting and handling timing errors, and for monitoring real-time tasks. The detection and handling requires less than 1 μsec overhead per reschedule operation, and has a latency of 12 μsec plus the length of one context switch for handling an error. The ATP mechanism uses 6 μsec per context switch, and requires 1 Kbyte of memory per 32 processes in the system.

We are now investigating techniques for integrating the timing error detection and handling with other non-timing error handling techniques.

The ATP is being extended in several directions.

First, some newer computers, including multiprocessor workstations, have μsec-resolution clocks in addition to the timer used for the system clock. An ATP mechanism that uses these clocks instead of the system clock is being designed.

Second, with higher-resolution timing, it now becomes feasible to not only time the execution of a task, but also the amount of time it spends blocked waiting for resources or preempted by an interrupt handler.

Key research issues include determining what raw data should be collected, what kind of derived information can be obtained from it, and minimizing run-time overhead and memory to make the mechanism practical.

6. References

[1] N. C. Audsley, A. Burns, M.F. Richardson, A.J. Wellings, "Deadline monotonic scheduling theory," in *Proc. of IFAC Workshop on Real Time Programming (WRTP '92)*; Bruges, Belgium; pp. 55-60, June 1992.

[2] J. P. Calvez and O. Pasquier, "Real-time behavior monitoring for multi-processor systems" *Microprocessing and Microprogramming, 38*, pp. 213-230. 1993.

[3] H. Chetto and M. Silly, "Scheduling strategies for periodic tasks to avoid timing faults in critical control systems," in *Proc. of IFAC Triennial World Congress*, Australia, pp.725-728, 1993.

[4] S. E. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," in *Proc. of 12th Real-Time Systems Symposium*, San Antonio, TX, pp. 74-83, Dec. 1991.

[5] D. Haban and D. Wybranietz; "A hybrid monitor for behavior and performance analysis of distributed systems"; *IEEE Trans. on Software Engineering*; v.16, n.2; Feb. 1990.

[6] Y. Ishikawa, H. Tokuda and C. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints", in *Proc of ECOOP/OOPSLA.*, Oct. 1990.

[7] K. Kenny and K-J. Lin, "Building Flexible Real-Time Systems Using the Flex Language", *IEEE Computer*, v.24, n.5, May 1991.

[8] E. Kligerman and A. D. Stoyenko, "Real-Time Euclid: A Language for Reliable Real-Time Systems", *IEEE Transactions on Software Engineering*, v. SE-12, n.9, Sep. 1986.

[9] J. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proc. of 8th IEEE Real-Time Systems Symp.*, pp. 261-270, Dec. 1987.

[10] C. L. Liu, and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment," *J. of the ACM*, v.20, n.1, pp. 44-61, Jan. 1973.

[11] S. Natarajan, Ed., *Imprecise and Approximate Computation*, Kluwer Academic Publishers, Boston, 1995.

[12] B. Platter, "Real-time execution monitoring"; *IEEE Trans. on Software Engineering*; v.10, n.6, Nov. 1984, pp 756-764.

[13] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *J. of Real-Time Systems*, v.1, n.1, pp. 27-60, Nov 1989.

[14] 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*, v.22, n.6, pp. 1282-1295, Nov./Dec. 1992.

[15] D. B. Stewart and P. K. Khosla, "Real-time scheduling of sensor-based control systems," in *Real-Time Programming*, ed. by W. Halang and K. Ramamritham, (Tarrytown, New York: Pergamon Press), 1992

[16] D. B. Stewart, *Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems*, Ph.D. Dissertation, Carnegie Mellon University (Pittsburgh, PA) April 1994.

[17] H. Streich, "TaskPair Scheduling: an approach for dynamic real-time systems," *Intl. J. of Mini and Microcomputers*, v.17, n.2, pp. 77-83, 1995.

[18] L. Svobodova, "Online system performance measurement with software and hybrid monitors"; *Operating System Review*, Vol. 7, no. 4, pp. 45-53, Oct. 1973

[19] M. Timmerman, F. Gielen, P. Lambrix, "High level tools for the debugging of real-time multiprocessor systems," in *Proc. of the IEEE Workshop on Real-Time Applications*, New York, NY, pp. 151-156, May 1993.

[20] H. Tokuda, T. Nakajima, and P. Rao, "Real-time Mach: Towards a predictable real-time system," in *Proc. of the USENIX Mach Workshop*, Oct. 1990.

[21] J.P. Tsai et al. "A non-interference monitoring and replay mechanism for real-time software testing and debugging" *IEEE Trans. on Software Engineering*, v.16, n.8, Aug. 1990.

[22] V. Wolfe, S. Davidson and I. Lee, "RTC: Language Support For Real-Time Concurrency", in *Proc. of Real-Time Systems Symposium*, San Antonio, Texas, Dec. 1991.