Presented in part at the 6th IEEE Workshop on Real-Time Operating Systems and Software, May 1989.

Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times

Dieter Haban[1] and Kang Shin[2]

TR-89-028

May 1989

Abstract

A real-time monitor is employed to aid in scheduling tasks with random execution times in a real-time computing system. Scheduling algorithms are usually based on the worst-case execution time (WET) of each task. Due to data-dependent loops and conditional branches in each program and resource sharing delay during execution, this WET is usually difficult to obtain and could be several orders of magnitude larger than the true exception time. Thus, scheduling tasks based on WET could result in a severe underutilization of CPU cycles and under-estimation of the systems schedulability.

To alleviate the above problem, we propose to use a real-time monitor as a scheduling aid. The real-time monitor is composed of dedicated hardware, called *Test and Measurement Processor* (TMP), and used to measure accurately, with minimal interference, the true execution time which consists of the pure execution time and resource sharing delay. The monitor is a permanent and transparent part of a real-time system, degrades system performance by less than 0.1%, and does not interfere with the host system's execution.

Using the measured pure execution time and resource sharing delay for each task, we have developed a mechanism which reduces the discrepancy between the WET and the estimated execution time. This result is then used to decide at an earliest possible time whether or not a task can meet its deadline.

1 International Computer Science Institute, Berkeley, California.

2 Kang G. Shin is on sabbatical leave from the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan 48109-2122.

The work reported in this paper has been supported in part by the Office of Naval Research under Contract No. N00014-85-K-0122 and by a grant from the International Computer Science Institute, Berkeley.

1. INTRODUCTION

Hard real-time systems are mainly characterized by their timing constraints, because they are responsible for safety- and time- critical control systems such as aircraft, nuclear reactors, and life-support instruments. The software of hard real-time systems features a large number of tasks which differ in priority, timing constraints and execution time. Ideally, every task should meet all of its timing constraints: start at a required time and produce results before a certain deadline. Since missing a task deadline might cause catastrophic consequences, one of the most important design issues in a real-time system is to schedule tasks to meet their deadlines. Most of the scheduling work known to date is based on a common assumption that execution times of all tasks to be scheduled are known *a priori*. However, determination of task (program) execution time is very difficult due mainly to data-dependent branches and loops in each program, and unpredictable delays associated with resource sharing [Woo87]. Since deadlines must be met even for the worst-case, real-time tasks are usually scheduled based on their worst-case execution times (WETs).[1] Since WET could be several orders of magnitude larger than the true execution time, scheduling tasks based on WET may lead to a severe underutilization of CPU cycles and/or incorrect decision on the schedulability of tasks, i.e., some tasks are declared to be unschedulable even if they can be completed in time. We propose that a real-time monitor be used to alleviate this problem.

A monitor can aid in verifying their timing behavior, detecting and locating abnormal behavior such as performance bottlenecks, and exploiting the monitored information for resource management, such as task scheduling and fault handling. Monitoring is defined as the extraction, processing and presentation of data about the activities of a computer system and has long been viewed as a fundamental means of evaluating computer systems. We shall focus on the feedback of monitored information to the monitored real-time system to achieve an adaptive behavior. In particular, the analyzed results about task execution behavior which are funnelled back to the

[1]Since the execution times of real-time tasks must be finite, loop counts are limited to be finite.

host's operating system are used for the dynamic scheduling of tasks. In order to use a monitor for this purpose, it must provide accurate, timely information about task execution behavior. We refer to monitoring under timing constraints as *real-time monitoring*. Non-intrusiveness is an important requirement of any monitoring tool, especially when it is used during normal operation of a real-time system.

As mentioned earlier, the execution time of a task consists of two components: (i) *pure execution time* (PET), and (ii) *resource sharing delay* (RSD). The execution time of a task will become identical to its PET if there is no delay in accessing shared resources during its execution. Note that RSD is unavoidable and varies randomly with the random fluctuation of system workload. Our proposed real-time monitor can measure both PET and RSD accurately, and, as we shall see, their separate measurements are very useful for the on-line check of schedulability of tasks.

This paper describes how a real-time monitor is used to measure the *elapsed pure execution time* (EPET), which is then used on-line to calculate the *anticipated pure execution time* (APET). Also presented is an architectural support consisting of hardware and software to satisfy the requirements of monitor's non-intrusiveness, accurate measurements, and feedback. Since use of monitored data is not limited to any particular scheduling policy, we focus only on how the accuracy and the dynamic adjustment of APET can improve scheduling, rather than developing any scheduling algorithm.

In the next section, we discuss problems associated with existing scheduling methods that do not use any feedback, and list the objectives of this paper. Section 3 details our approach to improve conventional scheduling methods and presents a demonstrative example. The real-time monitor that was built as part of the INCAS [Neh87] multicomputer project is described in Section 4. In Section 5, we give an example of measuring the task execution behavior and using it for the dynamic scheduling of tasks. The paper concludes with Section 6.

2. PROBLEMS WITH EXISTING SCHEDULING METHODS

Before discussing problems with existing scheduling methods, we define the terminology to be used in this paper (see also Appendix A). Since we abstract from an arbitrary real machine, the time is expressed in basic CPU cycles. The *worst-case pure execution time* (WPET) is the time the task will use to execute the longest path of the program (e.g., executing each loop a maximum number of times) without accounting for resource sharing delay (RSD). (Since we are concerned with real-time systems, we can assume all loop counts are finite, and so is the WPET.) The *elapsed pure execution time* (EPET) of a task at any given time t is the amount of time spent on executing the task since its invocation till time t excluding RSD. EPET is in general unknown *a priori* and can be determined only upon task completion.

WPET is usually determined off-line prior to the execution of a task and does not change during its lifetime. Any changes to the task will require this time to be re-determined. The determination of WPET is difficult due to data-dependent branches and loops, and is often several orders of magnitude larger than the true PET. The deviation of WPET from the true PET, however, plays an important role in scheduling tasks. Recall that WET = WPET + RSD. Tasks are scheduled based on WPET, RSD, their deadline, the *remaining pure execution time* (RPET) and the current time. Since the proposed monitor can extract RSD from WET and it is easy to express the dynamic scheduling of tasks with EPET and RPET, we shall not use RSD in the rest of the paper. (We shall, however, show how to measure RSD with our monitor.) Although there exist many interesting and good solutions to the scheduling problem, there remain two major problems associated with the determination and the accuracy of WPET and RPET as outlined below.

The more accurate WPET approximates the true PET, the better tasks will be scheduled. For example, based on the current EPET, the given deadline and WPET, a task can be aborted when it is determined to miss its deadline, thus saving valuable CPU cycles for other time-critical tasks. However, since WPET may be significantly larger than the true PET, a task may some-times be aborted even if it could meet its deadline.

Another problem is that the lack of accurate measurement tools makes it very difficult to determine EPET in the presence of unpredictable delays caused by resource contention, such as communication, synchronization and I/O. Therefore, the calculation of RPET at any time during task execution is usually inaccurate. Since whether a task will miss its deadline or not is decided on the basis of RPET, the decision has to be based on inaccurate values and unpredictable conditions. Moreover, the scheduling algorithm itself uses CPU cycles to determine an optimal schedule for tasks. This could be very complex and time-consuming if many parameters have to be considered in order to achieve the optimality, as is usually the case.

In order to solve and/or alleviate the above problems, we propose to use a real-time monitor as a scheduling aid. Feeding the monitored information about task execution back to the operating system will enable the system to dynamically respond to changing needs during the execution. Moreover, since the actual behavior of a real-time system is very difficult and expensive to simulate in an artificial environment, it is desirable to make decisions based on the actual monitored data and update information about the system behavior dynamically. Specifically, we shall in the rest of the paper focus on meeting the following requirements.

- At each stage of task execution, the real-time monitor must be designed to accurately measure EPET and RSD.
- Based on measurements, the WPET must be adjusted dynamically to approximate the true PET.
- The measurement and the processing of monitored data should cause as little interference in time and space with the host system as possible.
- The scheduling algorithm itself should cause as little overhead as possible in computing an optimal schedule for tasks.

3. REAL-TIME SYSTEM MANAGEMENT

3.1. Measuring Elapsed Pure Execution Times

Given WPET, let RPET(t) represent the remaining pure computation of a task at time t, and EPET be the amount of computation done by time t (all measured in basic CPU cycles). The accuracy in calculating the remaining execution time is very sensitive to the correctness of the scheduling decisions. EPET can be calculated by using the times measured by the proposed real-time monitor (more on this will be discussed later). The real-time monitor will begin counting up upon the start of a task, will stop counting as soon as the task gets blocked to wait for a resource, and will resume counting when the task gets unblocked and starts execution. In other words, the real-time monitor keeps track of the pure computation time of each task without considering delays due to resource contention and/or precedence constraints. If a task gets blocked at time t, the remaining computation at time t is calculated by $\text{RPET}(t) = \text{WPET} - \text{EPET}$. Then, given deadline D at any time $t' > t$ when the task is ready to start/resume execution, one calculates $T = D - \text{RPET}(t) - t'$. If $T > 0$, then the task is still schedulable, or may meet the deadline. If $T < 0$, then the task cannot be completed in time, or a *dynamic failure* [SKL85] will occur; the operator will be informed of this first, and some form of recovery measures will be invoked.

More specifically, the real-time monitor can be used to check the deadline of each task continuously while the task is running, or blocked to wait for a resource to be available. Thus, whether or not a task will miss its deadline is detected at an earliest possible time via the continuous monitoring of task execution. This earliest detection of a dynamic failure will give the system and/or the operator enough time — which would not be available without real-time monitoring — to take actions against the failure.

3.2. Anticipated Pure Execution Time

The calculation of RPET is based on the given WPET and the measured EPET. Therefore, WPET is an important parameter in determining RPET, and thus, whether a task will miss its deadline or not. However, as mentioned earlier, WPET might be several orders of magnitude larger than the true PET, thus making the scheduling decisions based on WPET inaccurate. With a more accurate estimate of PET (or APET) than WPET, the error caused by the significant gap between the WPET and true PET can be reduced, and thus, used for better scheduling of tasks.

Our main idea to counter the above problem is to take the past execution behavior of a task into consideration for the on-line calculation of APET as follows. A task is divided into *n* disjoint *parts* on the basis of its structure. For example, a task may be divided based on loops, i.e., loops and codes between loops become parts. Fig. 1 shows an example task division into non-overlapping parts. The WPET of each part is determined in the same way as for the entire task, i.e., using the longest possible paths within the part.

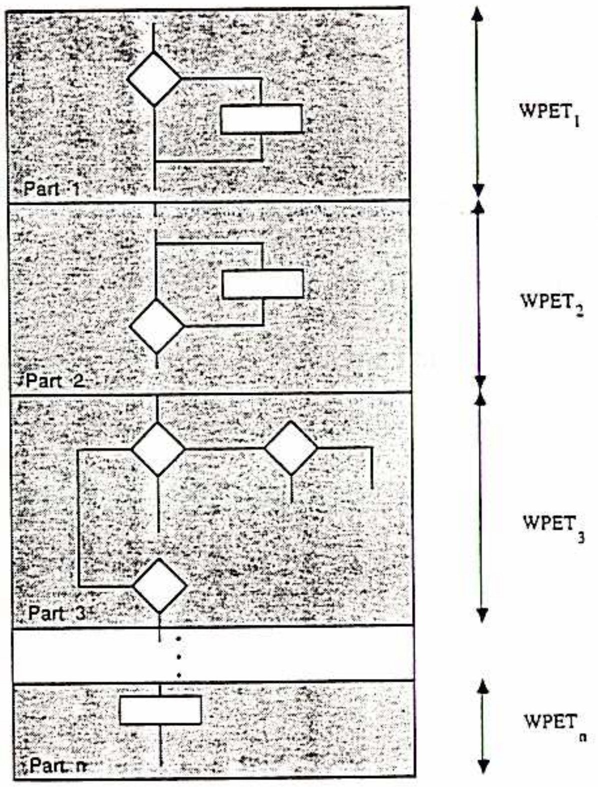

Figure 1: Parts of a task.

Instead of considering the WPET for the entire task, we use a vector **WPETV** = $(WPET_1$, $WPET_2,..., WPET_n)$, where each component is the WPET for the corresponding part of the task. In other words, $WPET_1$ is the worst-case pure execution time for part 1, $WPET_2$ for part 2, and so on. The sum of all components of **WPETV** is equal to the WPET of the entire task, which is used to schedule the task, i.e., WPET $= WPET_1 + WPET_2 + \cdots + WPET_n$.

The real-time monitor is used to measure accurately the start and completion of each part by placing triggering points into the code. (The detailed implementation and mechanisms to achieve this are described in Section 5.) If part 1 of the task is completed, the EPET of part 1, denoted by $EPET_1$, replaces $WPET_1$ to form **APETV** $= (EPET_1, WPET_2,...,WPET_n)$. Whenever part i of the task is completed, the measured $EPET_i$ replaces $WPET_i$ in **APETV**. Then, the sum of **APETV**'s components is used as a new APET for on-line scheduling of tasks. In other words, the measured and dynamically adjusted **APETV** in place of **WPTEV** is used for scheduling. In general, the new APET is much lower than the previous APET, since the WPET of one part is replaced by the true PET of that part. The number (n) of parts is variable and depends on the task structure and also on the desired accuracy towards the end of task completion. Initially, APET = WPET, meaning that $APET_i = WPET_i$ for all i. When the task is completed, APET = PET. Fig. 2 shows the transformation of the APET starting with WPET and ending with PET.

$$\begin{array}{ll}
\text{APETV}\ (t_0) = (\ \text{WPET}_1, \text{WPET}_2, \text{WPET}_3, ..., \text{WPET}_n) = \text{WPETV} & t_0\text{: initial} \\
\downarrow & \\
\text{APETV}\ (t_1) = (\ \text{EPET}_1, \text{WPET}_2, \text{WPET}_3, ..., \text{WPET}_n) \leq \text{WPETV} & t_1\text{: part 1 is completed} \\
\downarrow & \\
\text{APETV}\ (t_2) = (\ \text{EPET}_1, \text{EPET}_2, \text{WPET}_3, ..., \text{WPET}_n) \leq \text{WPETV} & t_2\text{: part 2 is completed} \\
\downarrow & \\
\text{APETV}\ (t_3) = (\ \text{EPET}_1, \text{EPET}_2, \text{EPET}_3, ..., \text{WPET}_n) \leq \text{WPETV} & t_3\text{: part 3 is completed} \\
\downarrow & \\
\text{APETV}\ (t_n) = (\ \text{EPET}_1, \text{EPET}_2, \text{EPET}_3, ..., \text{EPET}_n) \leq \text{WPETV} & t_n\text{: part n is completed}
\end{array}$$

Figure 2: Transformation of the anticipated execution time vector.

To demonstrate that APET approximates the true PET much better than WPET, we plotted in Fig. 3 the difference between WPET and the true PET as well as the difference between APET and the true PET.

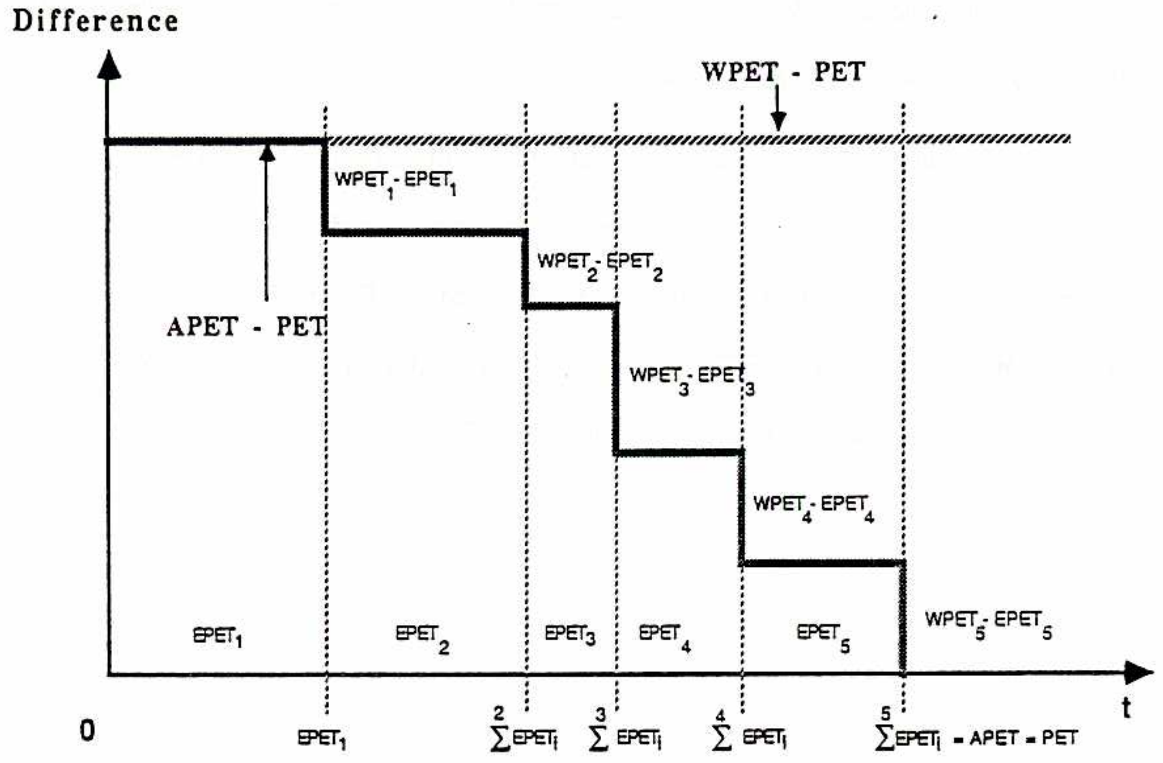

Figure 3: Differences between the cases of using WPET and APET.

The plots in Fig. 3 are not based on the actual numbers; even if the plots would be based on real measurements, they will vary every time the task is executed due to random input and environmental changes. The difference between APET and the true PET is shown to decrease monotonically as task execution progresses toward its completion, whereas the difference between WPET and the true PET remains constant. The difference between the true (unknown) PET and the APET is decreased by $WPET_i - EPET_i$ whenever part i is completed. In general, this decrease is significant, because $WPET_i$ represents the largest possible PET for part i for all i. Although the actual difference $WPET_i - APET_i$ depends on a particular task and input data, the graph indicates that APET approaches the true PET as its execution progresses to completion.

Since WPET is usually much larger than the true PET, the value of APET – PET is getting smaller as the *WPET*$_i$'s in **APETV** are being replaced by *EPET*$_i$'s. Another result is that the more parts we introduce, the faster APET approximates the true PET.

Obviously, the accuracy of RPET is important for the scheduling decision on each task. The following two examples illustrate that the RPET based on APET is much smaller than the RPET computed with WPET. (Thus, use of APET will reduce the probability of throwing out tasks as a result of incorrect decisions on whether or not the tasks will meet their deadline.) These two examples are based on an artificial workload generator which is used to compute *EPET*$_i$'s. The plots in Figs. 4 and 5 are obtained from using two different sets of *EPET*$_i$'s for the same task. (Therefore, both figures used the same **WPETV** but different **APETVs**.) Note that in Fig. 4, RPET = 158μs after t=13μs when APET is used, as compared to RPET = 735μs based when WPET is used. In Fig. 5, RPET = 444μs after t=6μs based on APET, as compared to RPET = 742μs based on WPET.

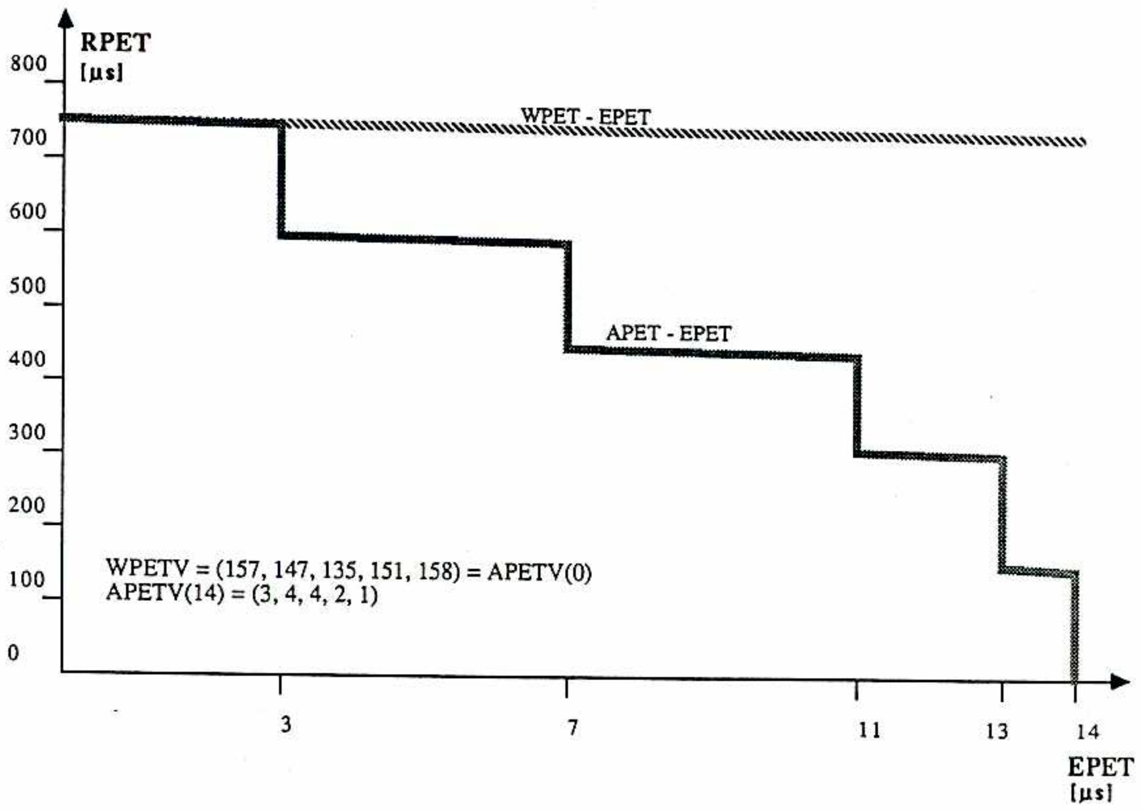

Figure 4: Computation of RPET based on WPET and APET: Example 1.

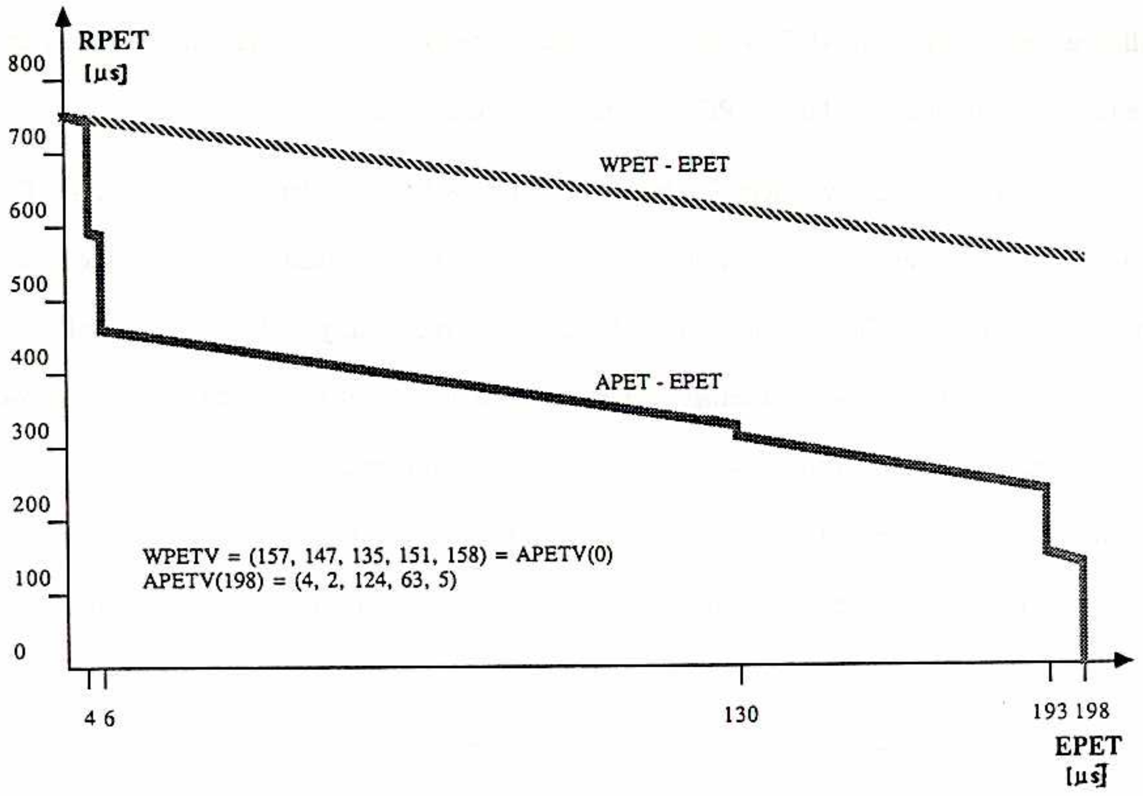

Figure 5: Computation of RPET based on WPET and APET: Example 2.

It is important to observe that using an accurate PET becomes more important as the task is getting closer to its completion/deadline. At the beginning of task execution, the task should be schedulable for the most of times, but as the task approaches its completion/deadline, the more urgent the monitoring of task execution time and deadline will become. This can be taken into account by dividing the task into larger (coarser) parts in the beginning and into smaller (finer) parts near the end of the task.

4. REAL-TIME MONITORING

4.1. Design Requirements

We list here the requirements of real-time monitoring necessary to support the scheduling method discussed thus far.

The hardware and software in most existing computer systems are not designed to be monitored, although monitoring tools in the form of stand-alone hardware devices, programs, and hybrid tools have been available for many years. Monitoring tools can be classified into pure hardware and pure software. A hardware monitor is a device that is not a part of the host system. Although such devices can be designed to have minimal or no effect on the host system, they generally provide only limited, low-level data about the host system activities and cannot be used to provide any desired data about the system's execution. Especially, these monitors have reached the frontiers of measurability in computer systems with modern hardware, such as cache memories and memory management units. Hardware monitors are not suited for systems with dynamic behavior where processes are created and migrated dynamically. On the other hand, software monitors can extract almost any desired information and can present it in an appropriate, user-oriented manner. These monitors are usually contained within the host system, sharing with it the same execution environment, and thus, producing some degree of interference in both the timing and space of the monitored program. Although simple software monitors, such as counters, are incorporated into the operating system, the accuracy of these monitors and the contents of the processed data are, due to efficiency reasons, not well suited for the measurement and management of real-time systems. Moreover, the interference caused by software monitors is not acceptable in real-time systems due to the resulting increase of system response time and unpredictable changes to system behavior.

For the above reasons, conventional monitors are not adequate for the measurement and management of real-time systems. New methods and tools are thus necessary to meet the following design requirements.

- **Interference**: the monitoring system should not change the host system behavior and have minimal effects on the host system performance.

- **Continuous monitoring during normal operation:** the monitoring system should be able to trace the host system, evaluate and supervise the execution of applications, and make information available for display and feedback in real-time about their progress. This service is particularly important when monitoring those programs that control safety- and time- critical systems, such as a nuclear power plant or an airborne system.
- **Integration:** the instrumentation of the monitoring system has to be incorporated into the host system during its design phase, leading to an integrated approach. Thus, the monitoring system can be permanently used for observation and management functions.
- **Feedback:** in order to allow the host system to dynamically respond to the monitored results, the monitor must be able to funnel its results back to the host system. This provides the host system with up-to-date information about its own activities, and can be used for fine performance tuning, scheduling decisions and error handling.
- **Real-time operating system support:** the monitor should be able to process the monitored data locally. In combination with the ability to execute routine low-level, operating system tasks, such as local load or network management, the host operating system is relieved of processing and management functions which could significantly increase the response time.

4.2. Realization

The insufficiency of current software and hardware monitors has led to the design of the Test and Measurement Processor (TMP). The hybrid approach of TMP combines the advantages of software and hardware monitors while overcoming their deficiencies. Hybrid monitors typically consist of (i) an independent hardware device which can perform the low-level monitoring, i.e., information gathering, and (ii) software programs executing on this device to measure, evaluate and display the host system performance. The TMP meets all the design requirements mentioned above. In this paper, we focus only on the use of the TMP for scheduling tasks. The TMP incorporates additional features for applications in distributed systems, such as debugging, load balancing, program evaluation and animation. The interested readers are referred to [HaW88, HaW89, WyH88].

4.2.1. TMP Principles

Efficient monitoring of task execution times is accomplished by using events generated by the application software. Events represent significant trends in the system behavior, such as assign, resign and block processes. The triggering points for these events are placed in the operating system kernel and the application code which then provide continuous information about the system behavior. These events are then collected, time-stamped, and processed by the separate TMP hardware. Therefore, the TMP is capable of executing local software for the various processing and evaluation needs for its host concurrently with the execution of application tasks. The analyzed results can then be displayed locally, combined with remote results from other TMPs, and fed back to the host system to achieve an adaptive behavior. By using semantic information about the monitored programs provided by the compiler and the programming environment, the monitoring software is able to access any desired information, such as task deadlines, WPETs, task names and task priorities. Fig. 6 illustrates the principles of the TMP-based approach.

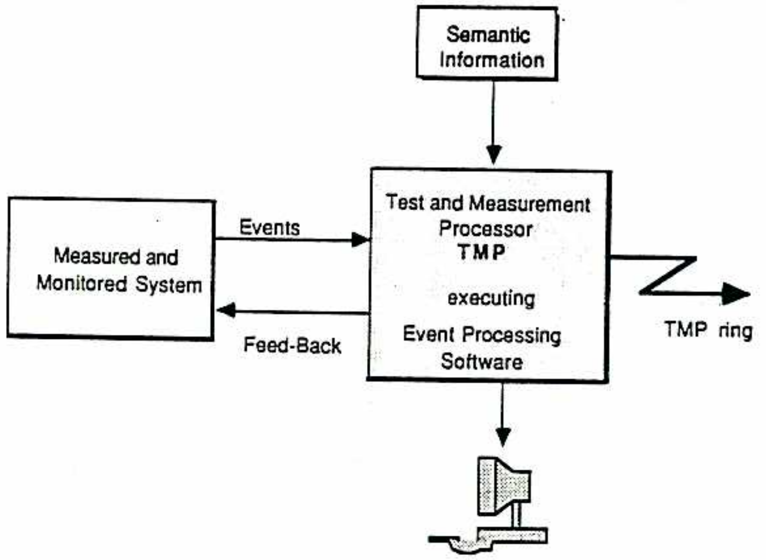
Semantic Information
Measured and Monitored System
Events
Feed-Back
Test and Measurement Processor TMP
executing
Event Processing Software
TMP ring

Figure 6: TMP principles.

The TMP may be viewed as an extra device responsible for monitoring, recording, and evaluating the activities of the host node as well as its communication activities. It was designed to be an integral part of each node in a multicomputer system. In a distributed environment that consists of multiple nodes, each TMP can receive data from any other TMP over a network; symmetrically, each TMP is able to send data to other TMPs. Ideally, the TMPs communicate via a separate network, thereby avoiding disturbances to the application tasks, which is the case for real-time systems. However, inter-TMP communications can also be accomplished by the host's communication facility, if the resulting interference does not cause an unacceptable disturbance to the host network. Experiments with the TMP in the INCAS multicomputer system showed that typically, 600-800 such events were generated every second on each node. The host overhead caused by the TMP is shown to be lower than 0.1%. Since this overhead is negligible, the TMP has become a permanent part of each node. In addition, since the instrumentation is permanent within the host system, and since the TMP can execute its own monitoring software, the host system's behavior is not changed by the monitor.

In the next subsection, we present the principle of encoding and decoding information via events and then describe the low-overhead mechanism which makes the events visible to the TMP hardware. The TMP hardware is outlined in Section 4.2.4. The evaluation software used for the scheduling purpose is described in Section 5.

4.2.2. Instrumentation of the Host System

Events represent the only overhead introduced by the TMP-based monitor. An event is defined as a special condition that occurs during the normal system activity such that it can be made visible to the TMP. There are two kinds of events, *optional* and *standard*. Optional events are associated with the application program. They are generated by the compilers or are placed manually into program code. Standard events are permanent and integral parts of the system. They are intended to support monitoring and measuring during normal system operation.

The minimal monitored activities necessary for the desired scheduling support include the dispatcher and the trace of the parts of a task. Table 1 gives a list of the corresponding events, each with the event class followed by a list of parameters.

start process	<procID> <procNo>
stop process	<procID>
assign process	<procID>
resign process	<procID>
ready process	<procID>
block process	<queueID>
end_part	<number>

Table 1: List of events necessary for task scheduling.

Dispatcher events trace the operations of the operating system dispatcher. The general model of a dispatcher is depicted in Fig. 7. It includes events to represent the states of a process: ready, blocked, running, killed. The parameter *procID* (see Table 1) identifies the process object, and the parameter *procNo* identifies its type. Note that dispatcher events are standard events and, therefore, a permanent part of the system. Programs need not be recompiled or relinked to be monitored. In the INCAS environment, additional standard events were included to reflect the activities of a distributed system, such as communication traffic.

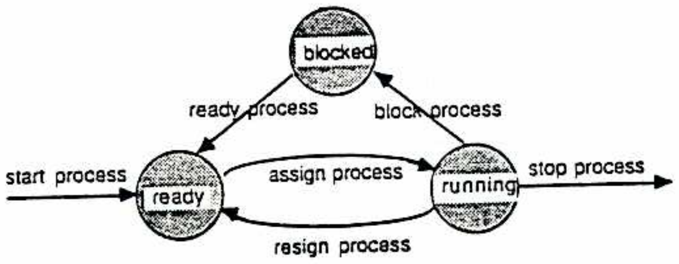

Figure 7: Events representing the operations of the dispatcher.

Events signaling the completion of parts of a task are inserted by the programmer into the real-time task by using a procedure call: EVENT (end_part, Number). The insertion of the event *end_part* is not necessary for the exact measurement of task execution times, since it can be accomplished with the standard instrumentation set.

Optional events are inserted by the programmer at the boundaries of task partitioning, and APET is updated whenever a part of each task is completed. If no optional event is inserted into the task's code, the task is considered to consist of only one part and, therefore, APET is not adjusted during the entire execution of the task. However, the insertion of the optional events after the end of each loop can be done automatically by the compiler.

The list of events can be expanded to keep track of other activities, such as interrupts, change of priority, change of deadline and process migration, which is not addressed in this paper.

4.2.3. Event Generation

The mechanism for generating an execution-time event consists of an instruction that is inserted at a specific, well-chosen point in the program code. Each event is marked by one store instruction which writes through the local processor cache. Therefore, each event is immediately visible on the system bus. If a memory management unit (MMU) is used to map logical addresses to physical addresses, one MMU register is programmed to ensure that events are visible within the same physical address range representing event classes. The format of the instruction is: *STORE ADDR, VALUE*. Each address represents one event class. In the current implementation, there are 256 different event classes, and therefore, the range of the address field is bound to 256 addresses. Examples of event class are *assign process* and *block process*. The VALUE of the store instruction serves as a parameter to specify one event within each class.

Since VALUE occupies 4 bytes of storage in our system, $256 * 2^{32}$ different events can be represented. In some cases, a standard event may require two parameters. For example, the event *start process <procID> <procNo>* is triggered by two instructions, one for each parameter. In such a case, the operating system kernel ensures that these events are treated as one atomic operation. Although in the current implementation, the TMP is implemented to distinguish between 256 event classes, only a small fraction is used: 32 standard and 26 optional event classes.

4.2.4. Hardware Implementation

The TMP hardware was designed to allow implementation on any computer system. It is connected to a system bus and monitors events on this bus with negligible impacts on the measured system. Fig. 8 shows the TMP hardware and its integration into a node of the INCAS multi-computer system.

The specific parts of the current TMP hardware consist of a M68000-based processor with 1 MByte of local memory, a dual RS-232 port for local interface, a network interface to other TMPs, and an Event Processing Unit (EPU). The bus interface is a separate module which does not affect the other parts of the TMP. The processor is used for the execution of monitoring software including low-level monitoring and evaluation routines which are resident in the memory. Higher-level evaluation software as well as operating system functions can be loaded into the TMP's memory. This software makes use of the data which are preprocessed and condensed by the lower-level monitoring software. We leave out the details of the TMP hardware features but emphasize only the collection of the events, since the latter is essential for the scheduling of tasks.

The EPU consists of a local event buffer, a comparator, a clock and an overflow counter. The local event buffer of the EPU is used as a FIFO for collecting sequences of events. The depth of the FIFO is 16 entries. This depth was determined to cope with a high arrival rate of events, and is based on experiences gained with an earlier prototype. Each entry in the event buffer consists of 80 bits: 8 for the event class, 32 for the event parameter, 36 for the timestamp (in μs), and 4 bits for control (CPU mode, overflow marker).

The comparator of the EPU is responsible for checking the addresses on the host bus. If an address falls within the range which represents event classes, the matched address and the next data on the bus are stored in the event buffer, along with the local time. The last byte of the address determines the event class; thus, it is the only byte stored by the EPU. The low-level implementation of the TMP ensures that the address range representing events does not interfere with the main memory.

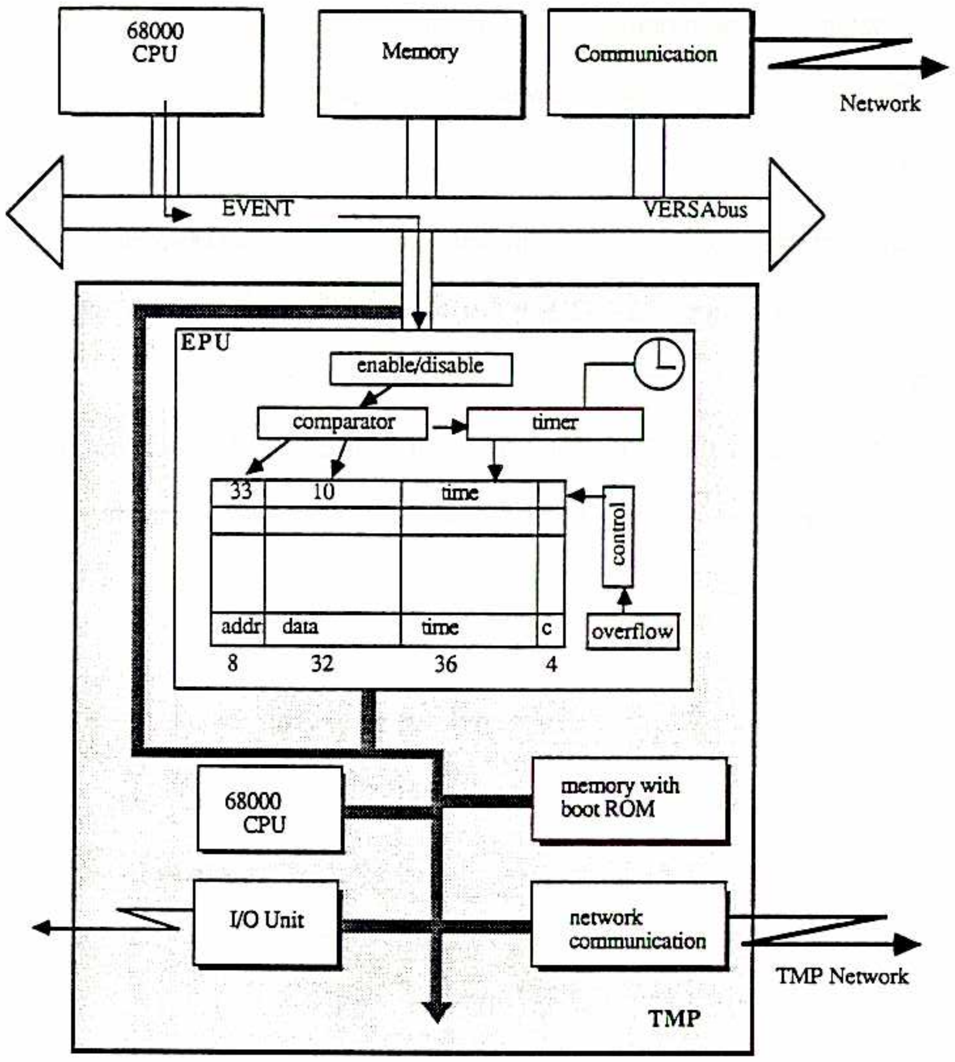

Figure 8: The TMP hardware.

The timer of the EPU is to measure the time difference between events. The resolution of this timer guarantees that no two successive events will have the same time. For example, in the current implementation, a time quantum of 1μs is used, thereby allowing up to 19 hours of measurements before the counter overflows. The accuracy of the timer can be interactively programmed. The timer quantum in no way affects the host system's timer.

The overflow counter of the EPU is used to count the number of events lost due to a buffer overflow. Although the specific type of each event is lost, the TMP software is aware of this loss. In order to detect the occurrence of an overflow, the first event placed in the buffer after such an overflow is marked in its control field. The monitoring software running on the TMPs can collect and process up to 13,000 events per second per node, which is ten times more than the average amount of collected events. We never experienced any overflow of the TMP buffers.

Finally, we note that the TMP has access to the memory of the host processor. This property is used to feed the results processed by the TMP back to the host operating system.

For fault-tolerance and reliability purposes, it is possible to add more than one TMP to each node without disturbing others. Thus, each TMP runs the same event processing software, and funnels results back to the host system which then extracts the necessary data.

5. ANALYSIS OF MONITORED DATA FOR TASK SCHEDULING

After collecting the information about a program's execution, raw data must be processed and analyzed to assist in task scheduling. The TMP hardware is responsible for the non-intrusive collection of run-time data based on the standard and optional events mentioned above. The TMP software allows for flexibility and comprises the *monitoring software*, which processes and analyzes incoming events, and system management functions. In particular, the scheduler itself consumes a significant amount of time to ''optimally'' determine which task to be scheduled next [PeS89]. Thus, the scheduler is made to run on the TMP while keeping the dispatcher to run on the host system. The scheduler supplies the dispatcher with the task identifier of the next task to be scheduled. A *watchdog* process running on the TMP is responsible for checking whether or not a task will meet the deadline at an earliest possible time during task execution. Another component which is loaded into the TMP memory is the information provided by the compiler and the programming environment about the real-time system to be monitored. This information is structured as a database with a hash function for fast access to the information stored therein, such as task deadlines and WPET for each task. Fig. 9 shows the components residing in the TMP. Note that the scheduler does not always have to be executed by the TMP. In such a case, the TMP can feed information about the execution of each task back to the host's scheduler which, in turn, determines the next task to be scheduled by using the CPU cycles of the host system.

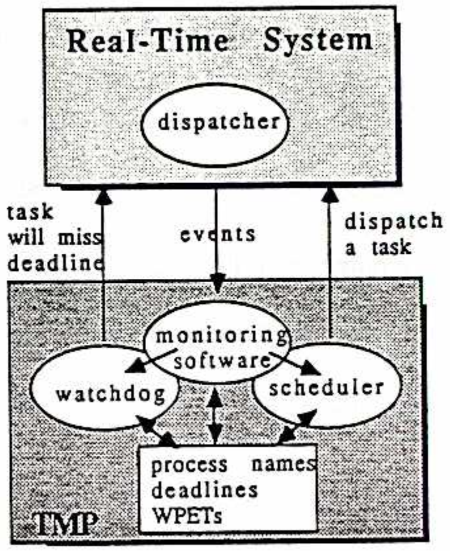

Figure 9: Software components residing in the TMP.

5.1. Measuring EPET and RSD of Each Task

In this subsection, we focus on the measurement of EPET and RSD. Based on the foregoing event instrumentation, events signal state transitions of a task from running to blocked, blocked to ready, and ready to running. Since each event is stored with a local timestamp, the TMP software can accurately measure elapsed times between events.

Upon occurrence of the event *assign process <procID>*, the monitoring software is informed that the process with the identifier <procID> has been activated (running). Thus, all the subsequent events (*end_part, block*) will follow from the activities of that process until a next *assign process* event.

The time difference between *assign process* and *resign process* or *block process* determines the EPET of a process. The time difference between *block process* and *ready process* determines the blocked time of a process. Cumulative execution and blocked times are stored in information tables for each process. Let T_{ass} be the timestamp of event *assign process*, T_{res} be the timestamp of event *resign process*, T_{block} be the timestamp of event *block process*, T_{ready} be the timestamp of event *ready process*, T_{start} be the timestamp of event *start process*, T_{stop} be the timestamp of event *stop process*, T_{part} be the timestamp of event *end part*, RSD_{block} be the time a process spent in the blocked queue, and RSD_{ready} be the time a process spent in the ready queue. Then, we can compute the following. Initially, EPET = 0, $RSD_{block} = 0$ and $RSD_{ready} = 0$. Upon occurrence of an event *block process*, one can compute:

$$\text{EPET} = \text{EPET} + (T_{block} - T_{ass}),$$

upon occurrence of an event *resign process*:

$$\text{EPET} = \text{EPET} + (T_{res} - T_{ass}),$$

upon occurrence of an event *ready process*:

$$RSD_{block} = RSD_{block} + (T_{ready} - T_{block}),$$

upon occurrence of an event *assign process*:

$$RSD_{ready} = RSD_{ready} + (T_{ass} - T_{ready}),$$

or if the task has been invoked and T_{ready} is undefined:

$$RSD_{ready} = RSD_{ready} + (T_{ass} - T_{start}).$$

Since the processor itself is a resource, the time a process spends in the ready queue waiting to be assigned to the processor is included in RSD. Thus, RSD is computed as RSD = RSD_{block} + RSD_{ready}.

Since we can use the computation of RSD_{ready} and RSD_{block} for other purposes, such as performance tuning and detection of bottlenecks, the TMP keeps track of these activities. Besides, the parameter <queueID> of the event *block process* allows the TMP software to compute RSD_{block} separately for each wait condition. However, it would be sufficient to compute only EPET for each task, since RSD for a particular task at any time can be easily computed by subtracting EPET from its lifetime. (The lifetime of a process is the actual real time minus the timestamp of the event *start time* of this process.)

5.2. Measurement of APET

To determine $EPET_i$ of a task, the TMP software computes the PET between two successive events *end_part <i-1>* and *end_part <i>* triggered by the task. For simplicity, we assume that the occurrence of the event *start (stop) process* determines the start of the first (last) part. Process switching times are taken into account when computing $EPET_i$. Note that the EPET of a particular process is updated upon occurrence of events *resign process* or *block process*. Recall that EPET(t) represents the EPET at time t. T_i is used to store an intermediate result of the computation. The following algorithm is devised by looking at the actions of the essential event sequence:

assign process <ID1>	$\rightarrow$	T_{ass_1}
...		
end part <i-1>	$\rightarrow$	$T_i = EPET(T_{ass\_1}) + T_{part\_i-1} - T_{ass\_1}$
...		
resign process <ID1>	$\rightarrow$	$EPET = EPET + T_{res} - T_{ass\_1}$
...other processes are executing		
assign process <ID1>	$\rightarrow$	T_{ass_2}
...		
end part <i>	$\rightarrow$	$EPET_i = EPET(T_{ass\_2}) + T_{part\_i} - T_{ass\_2} - T_i$

Figure 10: Algorithm to determine $EPET_i$.

5.3. Deadline Check

The watchdog process is responsible for checking at any time t whether a task in progress will miss its deadline or not. Checking of the schedulability of a task is usually performed when a task is ready to start/resume execution. However, since missing a deadline also depends on the time a process is blocked, the deadline check must also be done during the waiting for resources. To deal with this, the watchdog process running on the TMP continuously checks whether or not each active task will miss its deadline even while it is in the ready or blocked queue. A task is assigned to the processor only if its RPET is smaller than the difference between the deadline and the current time. The task may not be completed before its deadline if it is blocked once or more during its remaining execution. Thus, the watchdog process checks whether or not the deadline will be met using the following algorithm which is executed periodically at fixed, but selectable, time intervals.

When a task is blocked at time t, $\text{RPET}(t) = \text{WPET} - \text{EPET}(t)$ does not change with t. Given deadline D, at any time $t' > t$ when the algorithm is invoked, one calculates $T = D - \text{RPET}(t) - t'$. While the task is blocked, t' is the only value that changes. IF $T < 0$, then the task will miss its deadline, which will be detected by the watchdog process. The watchdog process will signal this to the host's operating system which, in turn, can initiate a recovery action. Note that the watchdog process does not consume any CPU cycles of the host system.

6. CONCLUSION

We proposed to use a real-time monitor as an aid in scheduling tasks with random execution times in real-time computing systems. The real-time monitor based on the TMP is transparently integrated into the system, measures and monitors the task execution without altering the system behavior and with negligible interference. One essential feature of the monitor is that the measured results are fed back to the operating system in order to achieve an adaptive behavior. Specifically, the feedback is used to aid in scheduling tasks and checking in real-time whether or not task deadlines can be met.

The TMP measures precisely the EPET of each task without taking RSD into account (although RSD is also measured by the monitor). The measured EPETs are then used to accurately compute the RPET of the task. In addition, measurements about the past execution behavior of a task are used to update its APET. It is shown that this APET approximates PET more accurately than WPET which is fixed over a task's lifetime but could be much larger than the corresponding PET. The TMP's ability to execute arbitrary software is used to perform real-time system management functions, such as the scheduler and the watchdog process, thus making (i) the scheduler not use any CPU cycles of the host system and (ii) the watchdog process not cause any overhead to the host system.

The architectural support with the TMP has made the above results possible. Note that the real-time monitor based on the TMP can also be used in distributed and parallel systems to aid in debugging, program animation, understanding of parallel behavior, load balancing, etc.

REFERENCES

[HaW88] D. Haban and W. Weigel, "Global Events and Global Breakpoints in Distributed Systems," *Proc. of the 21st Hawaii Int. Conf. on System Sciences*, Vol. II, pp. 166-175, Jan. 1988.

[HaW89] D. Haban and D. Wybranietz, "Monitoring and Measuring Parallel Systems," *Proc. 3rd Annual Parallel Processing Symposium*, vol. 2, pp. 499-513, March 1989.

[Neh87] J. Nehmer, *et al.*, "Key Concepts of the INCAS Multicomputer Project," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 8, pp. 913-923, 1987.

[PeS89] D. Peng and K. G. Shin, "Static Allocation of Periodic Tasks with Precedence Constraints in Distributed Real-Time Systems," *Proc. 1989 Int'l Conf. on Distributed Computing Systems*, June 1989 (in press).

[ShL88] K. G. Shin and H. Lee, "Port manipulator for the distributed realization of an integrated manufacturing system", *Int'l J. of Computer Systems Science and Engineering*, vol. 3, no. 1, pp. 21-31, Jan. 1988.

[ShM88] K. G. Shin and Y. K. Muthuswamy, "Message communications in a distributed real-time system with a polled bus," *Proc. 22nd Annual Hawaii Int'l Conf. on System Sciences*, vol.

II, pp. 703-711, Jan. 1989.

[SKL85] K. G. Shin, C. M. Krishna, and Y.- H. Lee, ''A unified method for evaluating real-time computer controllers and its applications,'' *IEEE Trans. on Automatic Control*, vol. AC-30, no. 4, pp. 357-366, Apr. 1985.

[WyH88] D. Wybranietz and D. Haban, ''Monitoring and Performance Measuring Distributed Systems during Operation,'' *ACM SIGMETRICS Conf.*, Santa Fe, May 1988. in: *ACM Performance Evaluation Review*, vol. 16, no. 1, pp. 197-206, May 1988.

[Woo88] M. H. Woodbury, ''Analysis of Execution Time of Real-Time Tasks,'' *Proc. Real-Time Systems Symp.*, pp. 222-231, Dec. 1987.

APPENDIX A: List of Acronyms

WET:	worst-case execution time.
PET:	measured pure execution time.
EPET:	elapsed pure execution time.
APET:	anticipated pure execution time.
APETV:	anticipated pure execution time vector.
WPET:	worst-case pure execution time.
WPETV:	worst-case pure execution time vector.
RSD:	resource sharing delay.
RPET:	remaining pure execution time.