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

Dieter Haban and Kang G. Shin, *Senior Member, IEEE*

***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 execution time. Thus, scheduling tasks based on WET could result in a severe under-utilization of CPU cycles and under-estimation of the system's 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 processors* (TMP's), and used to measure accurately, with minimal interference, the true execution time which consists of 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. A set of example tasks are experimentally measured in a simulated environment while varying their characteristics and the measured data are analyzed, demonstrating the utility and power of the proposed real-time monitor.

***Index Terms*—Deadline, nonintrusiveness, real-time monitoring, real-time scheduling, resource sharing delay, task execution time.**

I. 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 [9]. Since deadlines must be met even for the worst-case, real-time tasks are usually scheduled based on their worst-case execution times (WET's[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 of data about the activities of a compuer system. In [3], we gave a detailed account of the software and hardware of a monitoring tool with emphasis on measuring the performance, understanding the behavior, and presenting the monitored results. Such a monitoring tool was built as part of the INCAS multicomputer project [4]. The main goal of this project was to develop a comprehensive methodology for the design of locally distributed systems. The experimental environment consists of 11 nodes interconnected via a local area network.

This paper focuses on one major application of the monitoring tool: feeding the monitored information back to the monitored real-time system to achieve an adaptive behavior. Specifically, the analyzed results about task execution behavior are funnelled back to the host's operating system and 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.* Nonintrusiveness 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: 1) *pure execution time* (PET), and 2) *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 nonintrusiveness, accurate measurements, and feedback. Since use of monitored data is not limited to any particular

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

Manuscript received October 11, 1989; revised June 4, 1990. Recommended by E. Gelenbe. This work was supported in part by the International Computer Science Institute, Berkeley, CA, under a postdoctoral grant, and by the Office of Naval Research under Contract N00014-85-K-0122.

D. Haban is with Daimler Benz AG, Research Center Ulm (FUZ), Postfach 80 02 30, 7000 Stuttgart 80, West Germany.

K. G. Shin is with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 9039333.

0098-5589/90/1200-1374$01.00 © 1990 IEEE

scheduling policy, we focus only on how the accuracy and the dynamic adjustment of APET can improve scheduling, rather than developing a 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 III details our approach to improve conventional scheduling methods and presents a demonstrative example. The real-time monitor is briefly described in Section IV. In Section V, we give an example of measuring the task execution behavior and using it for the dynamic scheduling of tasks. In Section VI, various measurements with a set of tasks are presented to show improvements in the utilization of CPU cycles and the ratio of tasks finished in time to those missing deadlines when the proposed real-time monitor is used as an aid in scheduling tasks. The paper concludes with Section VII.

II. Problems with Existing Scheduling Methods

Before discussing problems with existing scheduling methods, we define the terminology to be used in this paper (see also the Appendix). 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 until 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 redetermined. 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 sometimes 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.

III. Real-Time System Management

A. Measuring Elapsed Pure Execution Times

Given WPET, let RPET(t) represent the remaining pure computation of a task at time t, and EPET(t) 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 RPET(t) = WPET $-$ EPET(t). Then, given deadline D at any time $t' > t$ when the task is ready to start/resume execution, one calculates $T = D -$ 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* [8] 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.

B. 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 large 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 nonoverlapping 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.

Instead of considering the WPET for the entire task, we use a vector $\mathbf{WPETV} = (WPET_1, WPET_2, \cdots, WPET_n)$, 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 usually used to schedule the task, i.e., $\mathrm{WPET} = WPET_1 + WPET_2 + \cdots + WPET_n$.

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 V.) If part 1 of the task is completed, the EPET of part 1, denoted by $EPET_1$, replaces $WPET_1$ to form $\mathbf{APETV} = (EPET_1, WPET_2, \cdots, WPET_n)$. 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.

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.

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.

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.

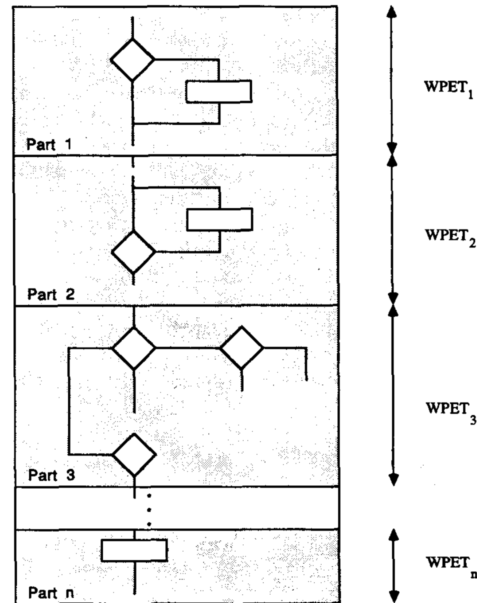

Fig. 1. Parts of a task.

$\mathrm{APETV}(t_0) = (WPET_1, WPET_2, WPET_3, ..., WPET_n) = \mathrm{WPETV}$ t_0: initial

↓

$\mathrm{APETV}(t_1) = (EPET_1, WPET_2, WPET_3, ..., WPET_n) \leq \mathrm{WPETV}$ t_1: part 1 is completed

↓

$\mathrm{APETV}(t_2) = (EPET_1, EPET_2, WPET_3, ..., WPET_n) \leq \mathrm{WPETV}$ t_2: part 2 is completed

↓

$\mathrm{APETV}(t_3) = (EPET_1, EPET_2, EPET_3, ..., WPET_n) \leq \mathrm{WPETV}$ t_3: part 3 is completed

↓

$\mathrm{APETV}(t_n) = (EPET_1, EPET_2, EPET_3, ..., EPET_n) \leq \mathrm{WPETV}$ t_n: part n is completed

Fig. 2. Transformation of the anticipated execution time vector.

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 **APETV's**.) Note that in Fig. 4, RPET = 158 μs after t = 13 μs when APET is used, as compared to RPET = 735 μs based when WPET is used. In Fig. 5, RPET = 444 μs after t = 6 μs based on APET, as compared to RPET = 742 μs based on WPET.

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.

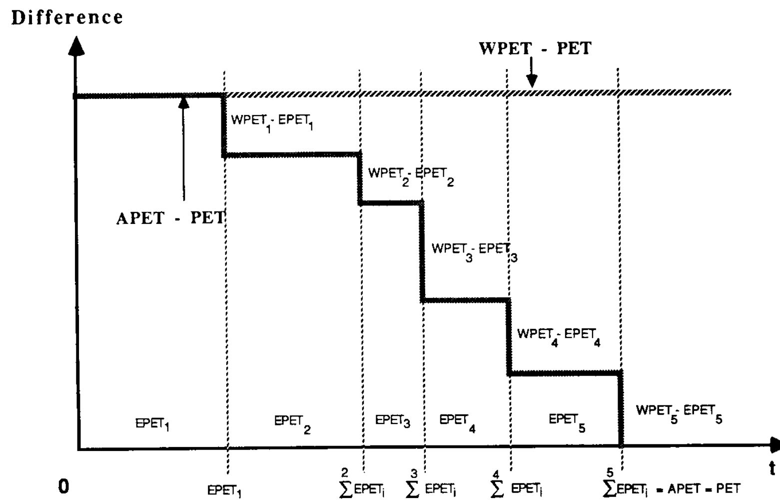

Fig. 3. Differences between the cases of using WPET and APET.

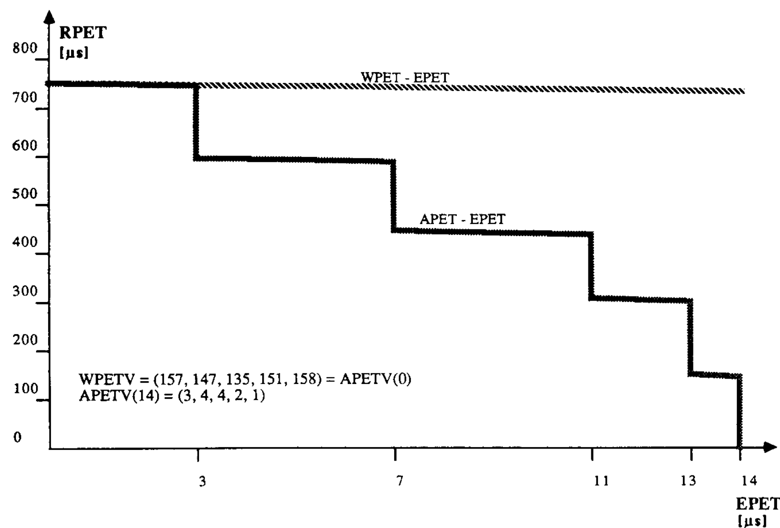

Fig. 4. Computation of RPET based on WPET and APET: example 1.

IV. Real-Time Monitoring

A. 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.

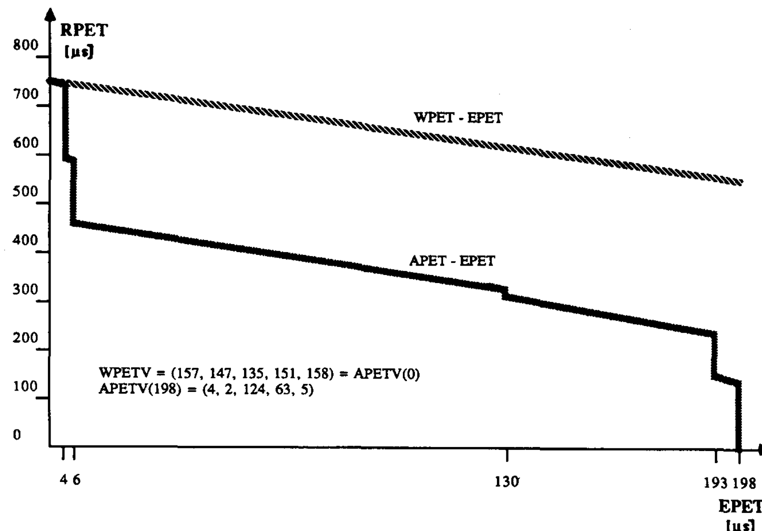

Fig. 5. Computation of RPET based on WPET and APET: example 2.

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.

B. Realization

The insufficiency of current software and hardware monitors has led to the design of the test and measurement processor (TMP) [3]. The hybrid approach of TMP combines the advantages of software and hardware monitors while overcoming their deficiencies. Hybrid monitors typically consist of 1) an independent hardware device which can perform the low level monitoring, i.e., information gathering, and 2) 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 briefly on the use of the TMP for scheduling tasks and refer the interested reader to [3].

1) TMP Principles: Efficient monitoring of task execution times is accomplished by using events generated by the monitored 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 TMP's, 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, WPET's, task names, and task priorities. Fig. 6 illustrates the principles of the TMP-based approach.

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. For applications in a distributed environment, the TMP's exchange data via a separate TMP 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 addi-

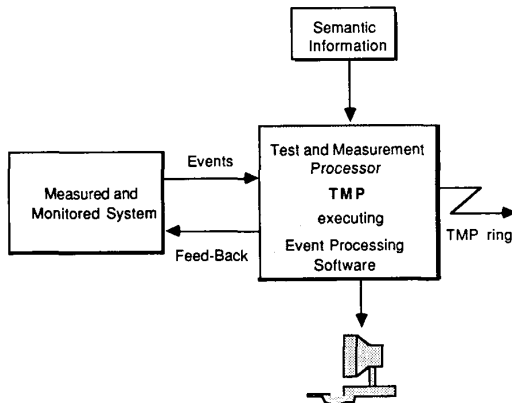

Fig. 6. TMP principles.

TABLE I
LIST OF EVENTS NECESSARY FOR TASK SCHEDULING

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

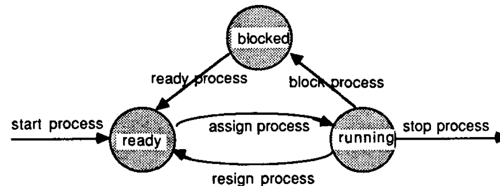

Fig. 7. Events representing the operations of the dispatcher.

tion, 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.

The next subsections present the principle of encoding and decoding information via events and then focus on their detection using the TMP hardware, followed by a description of the evaluation software used for the scheduling purpose.

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 I gives a list of the corresponding events, each with the event class followed by a list of parameters.

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. The parameter *procID* (see Table I) 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.

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.

3) Event Generation: The mechanism for generating an execution-time event consists of a store instruction that is inserted at a specific, well-chosen point in the program code. Since the store instruction writes through the local processor cache, each event is immediately visible on the system bus. The format of the instruction is: *STORE ADDR, VALUE.* Each address represents one event class (e.g., *assign process*), and therefore, the range of the address field is bound to 256 addresses. The VALUE of the store instruction serves as a parameter to specify one event within each event class. The seven event classes listed in Table I are sufficient for scheduling real-time tasks.

4) Hardware Implementation: The TMP hardware 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 computer node. 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 TMP's, and an event processing unit (EPU). The processor is used for the execution of monitoring software including low-level monitoring and evaluation routines as well as operating system functions. The monitoring software running on the TMP can collect and process up to 13 000 events per second per node, which is ten times more than the average number of collected events. We leave out the details of the TMP hardware features but emphasize only the collection of the events, since the latter is essential for scheduling 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

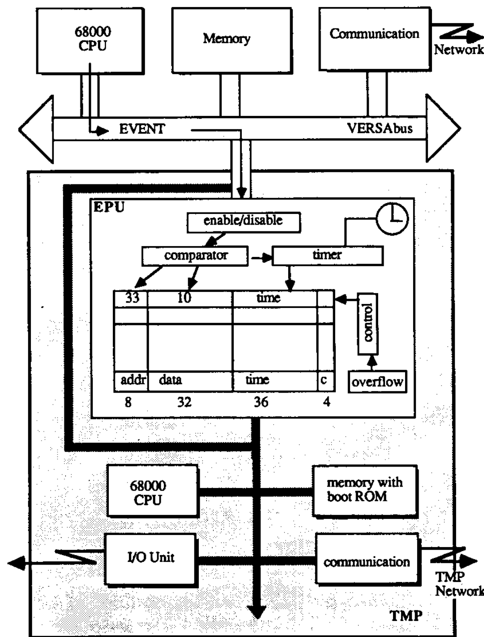

Fig. 8. The TMP hardware.

event buffer consists of the event class, the event parameter, the timestamp (in μs), and control information (CPU mode, overflow marker). We never experienced any overflow of the TMP buffers. 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. 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 1 μs is used, thereby allowing up to 19 hours of measurements before the counter overflows.

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.

V. 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 scheduling tasks. The TMP hardware is responsible for the nonintrusive collection of run-time data based on the standard and optional events mentioned above. Fig. 9 shows the software components residing in the TMP. The TMP software allows for flexibility and comprises the *monitoring software,* which processes and analyzes incoming events, and system management functions.

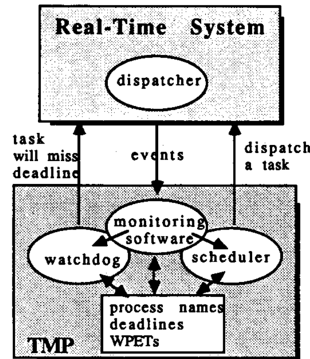

Fig. 9. Software components residing in the TMP.

In particular, the scheduler itself consumes a significant amount of time to "optimally" determine which task to be scheduled next [5]. 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. 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.

A. 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} by 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 time parameters. 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.)

B. 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 algorithm shown in Fig. 10 is devised by looking at the actions of the essential event sequence.

C. 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

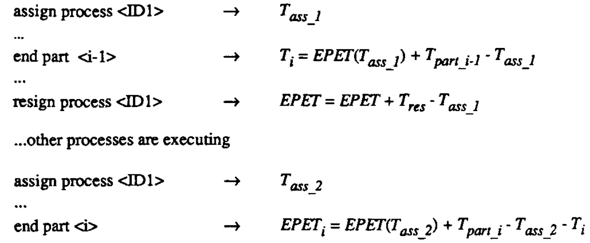

Fig. 10. Algorithm to determine $EPET_i$.

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, RPET(t) = WPET − 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 cycle of the host system.

VI. Experimental Measurements and Analysis

Experiments are conducted on a system with and without the proposed real-time monitor (RTM), showing improvements by the RTM in both the utilization of CPU cycles and the ratio of tasks completed in time to those missing deadlines. Measurements are taken in a simulated environment to provide the same condition for every experiment: the same sequence of scheduled tasks, no RSD except waiting in the dispatcher's ready queue, and the same APETV of a task during a given sequence of measurements. Task data are taken from real measurements with the TMP on an M68000 system [3]. Since the TMP-based real-time monitor measures both EPET and RSD accurately, the unpredictable RSD is extracted away from WPET. Thus, the dynamic scheduling of tasks can easily be expressed with EPET and RPET only.

The following experiments are conducted on 10 tasks of the same type, each with **WPETV** = (361, 459, 259, 359, 263), and thus, WPET = 1701. All time parameters are measured in μs. Three sequences of measurements with different **APETV**'s are considered to evaluate the efficiency of our method when WPET − PET is large, medium, and small. During a given sequence of measurements, a particular task's **APETV** remains unchanged, but each task has a different **APETV**. In the first sequence, WPET is on the average one order of magnitude greater than PET. In the second sequence, WPET is about five times larger than PET. In the third sequence, WPET is only twice larger than PET. During a sequence of measurements, the task deadline D is the only variable. For simplicity, but without loss of generality, we assume all tasks to start at the same time, say 0, and have the same deadline D. Since all ten tasks have the same priority and deadline during an experiment, tasks are switched based on a round-robin policy. Two different systems, one with the proposed RTM and the other without the RTM, are used to execute the program formed by these ten tasks. The first system is aided by the RTM in scheduling tasks and

continuously checking schedulability, whereas the second system always uses a constant WPET and checks schedulability only when a task is ready to resume. Unlike the first system, the second system cannot take the inaccuracy of EPET and RSD, and thus, the inaccuracy of RPET, into account when testing for schedulability. With this setup, our experimental results will indicate the degree of schedulability improvement via real-time monitoring. Another source of improvement via the RTM—that is not reflected in the following measurements—is to save the time required for the system without the RTM to execute the scheduling algorithm. We will refer to the system with the RTM as the *RTM case* and to the system without the RTM as the *NO-RTM case.*

The CPU utilization is computed as the CPU cycles consumed by the ten tasks during a sequence of measurements divided by the CPU cycles the ten tasks will consume in case their execution is completed in time. The CPU cycles consumed by those tasks which will eventually miss their deadlines form the wasted time. In the following measurements, WPET is always kept constant at 1701 for the NO-RTM case, and therefore, is not presented. Since the RTM's ability of measuring RSD accurately eliminates the need of considering RSD for scheduling tasks, the deadline D can be made not to depend on the unpredictable RSD, and thus, is not set to the number of tasks multiplied by WPET which would always be 17 010. With D = 17 010, all tasks will always finish in time. Tasks which missed their deadlines are marked with **XXX** in the tables given below.

A. First Sequence of Measurements

In the first sequence of measurements, WPET is ten times larger than PET. **APETV**'s for the ten tasks are computed as follows:

Task	**APETV** (1)	**APETV** (2)	**APETV** (3)	**APETV** (4)	**APETV** (5)
0	11	65	97	85	19
1	5	178	9	34	88
2	23	6	50	20	53
3	145	2	5	21	7
4	3	26	15	9	44
5	2	84	0	6	83
6	7	49	1	10	14
7	50	113	2	96	109
8	27	5	35	23	1
9	9	113	4	9	199

During this sequence, 6 experiments are performed while varying deadlines to be 2000, 2200, 2500, 2800, 3300, and 3500, and keeping **APETV** and WPET unchanged. These deadlines are chosen based on the following observation: the first deadline is close to the one before which all tasks will not be completed, the last deadline is approximately equal to the time before which all tasks will finish. All the other deadlines represent significant states between these two extremes.

Deadline = 2000										
RTM	CPU_util = 0.50 wasted 4.69 %					**NO_RTM**	CPU_util = 0.25 wasted 47.43 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
6 XXX	1701	1701	81	0	300	6 XXX	1701	81	0	300
7 XXX	1701	1701	370	0	300	7 XXX	1701	370	0	300
8 XXX	1701	1701	91	0	300	8 XXX	1701	91	0	300
9 XXX	1701	1701	334	0	300	9 XXX	1701	334	0	300
3 XXX	1701	1651	210	50	350	1 XXX	1651	314	50	350
4 finished	575	479	97	97	496	2 XXX	1651	152	50	350
2 finished	621	470	152	152	848	3 XXX	1651	210	50	350
5 finished	967	793	175	175	873	4 XXX	1651	97	50	350
0 finished	780	504	277	277	1000	5 XXX	1651	175	50	350
1 finished	748	435	314	314	1064	0 finished	1424	277	277	526

Deadline = 2200										
RTM	CPU_util = 0.58 wasted 28.61 %					**NO_RTM**	CPU_util = 0.34 wasted 61.89 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
3 XXX	1701	1651	210	50	550	1 XXX	1651	314	50	550
4 finished	575	479	97	97	696	2 XXX	1651	152	50	550
6 finished	589	509	81	81	777	3 XXX	1651	210	50	550
8 finished	612	522	91	91	868	4 XXX	1651	97	50	550
7 XXX	1390	1290	370	100	911	5 XXX	1651	175	50	550
9 XXX	1349	1249	334	100	952	6 XXX	1651	81	50	550
1 XXX	1345	1245	314	100	956	7 XXX	1651	370	50	550
2 finished	621	470	152	152	1120	8 XXX	1651	91	50	550
5 finished	967	793	175	175	1145	9 XXX	1651	334	50	550
0 finished	780	504	277	277	1222	0 finished	1424	277	277	726

Deadline = 2500										
RTM	CPU_util = 0.76 wasted 24.89 %					**NO_RTM**	CPU_util = 0.50 wasted 56.87 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
4 finished	575	479	97	97	746	4 finished	1604	97	97	746
6 finished	589	509	81	81	827	6 finished	1620	81	81	827
3 XXX	1701	1601	210	100	900	8 XXX	1651	91	50	878
8 finished	612	522	91	91	918	9 XXX	1651	334	50	878
7 XXX	1390	1240	370	150	1261	1 XXX	1601	314	100	928
1 XXX	1345	1195	314	150	1306	2 XXX	1601	152	100	928
2 finished	621	470	152	152	1320	3 XXX	1601	210	100	928
5 finished	967	793	175	175	1345	5 XXX	1601	175	100	928
0 finished	780	504	277	277	1522	7 XXX	1601	370	100	928
9 finished	657	324	334	334	1606	0 finished	1424	277	277	1054

Deadline = 2800										
RTM	CPU_util = 1.00 wasted 0.00 %					**NO_RTM**	CPU_util = 0.64 wasted 59.43 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
4 finished	575	479	97	97	746	4 finished	1604	97	97	746
6 finished	589	509	81	81	827	6 finished	1620	81	81	827
8 finished	612	522	91	91	918	8 finished	1610	91	91	918
2 finished	621	470	152	152	1420	7 XXX	1601	370	100	1219
5 finished	967	793	175	175	1495	9 XXX	1601	334	100	1219
3 finished	725	516	210	210	1705	1 XXX	1551	314	150	1269
0 finished	780	504	277	277	1832	2 XXX	1551	152	150	1269
1 finished	748	435	314	314	1996	3 XXX	1551	210	150	1269
9 finished	657	324	334	334	2080	5 XXX	1551	175	150	1269
7 finished	783	414	370	370	2100	0 finished	1424	277	277	1345

Deadline = 3300										
RTM	CPU_util = 1.00 wasted 0.00 %					**NO_RTM**	CPU_util = 0.90 wasted 26.35 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
4 finished	575	479	97	97	746	4 finished	1604	97	97	746
6 finished	589	509	81	81	827	6 finished	1620	81	81	827
8 finished	612	522	91	91	918	8 finished	1610	91	91	918
2 finished	621	470	152	152	1420	2 finished	1549	152	152	1420
5 finished	967	793	175	175	1495	5 finished	1526	175	175	1495
3 finished	725	516	210	210	1705	3 finished	1491	210	210	1705
0 finished	780	504	277	277	1832	0 finished	1424	277	277	1832
1 finished	748	435	314	314	1996	7 XXX	1451	370	250	1883
9 finished	657	324	334	334	2080	9 XXX	1451	334	250	1883
7 finished	783	414	370	370	2100	1 finished	1387	314	314	1896

Deadline = 3500										
RTM	CPU_util = 1.00 wasted 0.00 %					**NO_RTM**	CPU_util = 1.00 wasted 0.00 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
4 finished	575	479	97	97	746	4 finished	1604	97	97	746
6 finished	589	509	81	81	827	6 finished	1620	81	81	827
8 finished	612	522	91	91	918	8 finished	1610	91	91	918
2 finished	621	470	152	152	1420	2 finished	1549	152	152	1420
5 finished	967	793	175	175	1495	5 finished	1526	175	175	1495
3 finished	725	516	210	210	1705	3 finished	1491	210	210	1705
0 finished	780	504	277	277	1832	0 finished	1424	277	277	1832
1 finished	748	435	314	314	1996	1 finished	1387	314	314	1996
9 finished	657	324	334	334	2080	9 finished	1367	334	334	2080
7 finished	783	414	370	370	2100	7 finished	1331	370	370	2100

It is obvious that if all tasks finish in time, the CPU utilization is optimal since no time will be wasted. This sequence of experiments shows that the RTM case significantly improves the CPU utilization due to the fact that for a given deadline the more tasks finish in time, the less CPU cycles will be wasted. The CPU utilization of the RTM case is approximately twice as large as that of the NO-RTM case while the time wasted on executing tasks missing deadlines is only a half of the NO-RTM case. When the RTM is used, the significant difference between WPET and PET results in a significant increase in the ratio of finished tasks to thrown-out tasks. For example, when $D = 2800$, the system with the RTM can complete all 10 tasks in time, whereas the system without the RTM will throw 6 out of 10 tasks (since WPET is used to determine schedulability).

B. Second Sequence of Experiments

WPET is now set to be five times greater than PET and **APETV** for the ten tasks are computed as follows.

Task	**APETV** (1)	**APETV** (2)	**APETV** (3)	**APETV** (4)	**APETV** (5)
0	82	120	88	60	97
1	74	63	80	74	0
2	15	113	81	86	61
3	114	70	30	2	101
4	4	115	72	119	7
5	68	107	134	104	90
6	4	121	135	91	0
7	62	113	57	4	89
8	136	120	83	52	56
9	76	92	31	91	53

During this sequence, 6 experiments are conducted while varying deadlines to be 3500, 3750, 4000, 4500, 4750, and 5000, and keeping **APETV** and WPET unchanged. These deadlines represent significant states and are larger than those in the first sequence, since the tasks need more CPU cycles to complete.

Deadline = 3500										
RTM	CPU_util = 0.82 wasted 59.01 %					**NO_RTM**	CPU_util = 0.59 wasted 79.65 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
8 XXX	1476	1276	447	200	2225	9 XXX	1551	343	150	1950
1 finished	739	449	291	291	2540	1 XXX	1501	291	200	2000
5 XXX	1056	756	503	300	2745	2 XXX	1501	356	200	2000
0 XXX	872	522	447	350	2979	3 XXX	1501	317	200	2000
9 XXX	812	512	343	300	2989	4 XXX	1501	317	200	2000
3 finished	738	422	317	317	3007	5 XXX	1501	503	200	2000
6 XXX	782	482	351	300	3019	6 XXX	1501	351	200	2000
4 finished	832	516	317	317	3024	7 XXX	1501	325	200	2000
2 XXX	817	467	356	350	3034	8 XXX	1501	447	200	2000
7 finished	758	434	325	325	3049	0 finished	1254	447	447	2196

Deadline = 3750										
RTM	CPU_util = 0.89 wasted 16.68 %					**NO_RTM**	CPU_util = 0.66 wasted 81.73 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
8 XXX	1476	1226	447	250	2525	5 XXX	1501	503	200	2250
1 finished	739	449	291	291	2590	6 XXX	1501	351	200	2250
5 XXX	1056	756	503	300	2995	7 XXX	1501	325	200	2250
3 finished	738	422	317	317	3057	8 XXX	1501	447	200	2250
4 finished	832	516	317	317	3074	9 XXX	1501	343	200	2250
7 finished	758	434	325	325	3149	1 XXX	1451	291	250	2300
9 finished	812	470	343	343	3192	2 XXX	1451	356	250	2300
2 finished	817	462	356	356	3248	3 XXX	1451	317	250	2300
6 finished	782	432	351	351	3249	4 XXX	1451	317	250	2300
0 finished	872	426	447	447	3296	0 finished	1254	447	447	2446

Deadline = 4000										
RTM	CPU_util = 0.94 wasted 7.14 %					**NO_RTM**	CPU_util = 0.72 wasted 83.42 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
1 finished	739	449	291	291	2590	1 XXX	1451	291	250	2550
8 XXX	1476	1226	447	250	2775	2 XXX	1451	356	250	2550
3 finished	738	422	317	317	3057	3 XXX	1451	317	250	2550
4 finished	832	516	317	317	3074	4 XXX	1451	317	250	2550
7 finished	758	434	325	325	3199	5 XXX	1451	503	250	2550
9 finished	812	470	343	343	3242	6 XXX	1451	351	250	2550
2 finished	817	462	356	356	3298	7 XXX	1451	325	250	2550
6 finished	782	432	351	351	3349	8 XXX	1451	447	250	2550
0 finished	872	426	447	447	3396	9 XXX	1451	343	250	2550
5 finished	935	433	503	503	3499	0 finished	1254	447	447	2606

Deadline = 4500										
RTM	CPU_util = 1.00 wasted 0.00 %					**NO_RTM**	CPU_util = 0.86 wasted 67.08 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
1 finished	739	449	291	291	2590	1 finished	1410	291	291	2590
3 finished	738	422	317	317	3107	3 finished	1384	317	317	3107
4 finished	832	516	317	317	3124	4 XXX	1401	317	300	3108
7 finished	758	434	325	325	3249	5 XXX	1401	503	300	3108
9 finished	812	470	343	343	3342	6 XXX	1401	351	300	3108
2 finished	817	462	356	356	3398	7 XXX	1401	325	300	3108
6 finished	782	432	351	351	3449	8 XXX	1401	447	300	3108
0 finished	872	426	447	447	3546	9 XXX	1401	343	300	3108
8 finished	913	467	447	447	3643	2 XXX	1351	356	350	3158
5 finished	935	433	503	503	3696	0 finished	1254	447	447	3204

Deadline = 4750										
RTM	CPU_util = 1.00 wasted 0.00 %					**NO_RTM**	CPU_util = 0.94 wasted 31.46 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
1 finished	739	449	291	291	2590	1 finished	1410	291	291	2590
3 finished	738	422	317	317	3107	3 finished	1384	317	317	3107
4 finished	832	516	317	317	3124	4 finished	1384	317	317	3124
7 finished	758	434	325	325	3249	7 finished	1376	325	325	3249
9 finished	812	470	343	343	3342	9 finished	1358	343	343	3342
2 finished	817	462	356	356	3398	2 finished	1345	356	356	3398
6 finished	782	432	351	351	3449	6 XXX	1351	351	350	3449
0 finished	872	426	447	447	3546	8 XXX	1351	447	350	3449
8 finished	913	467	447	447	3643	0 finished	1254	447	447	3495
5 finished	935	433	503	503	3696	5 XXX	1301	503	400	3496

Deadline = 5000										
RTM	CPU_util = 1.00 wasted 0.00 %					**NO_RTM**	CPU_util = 1.00 wasted 0.00 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
1 finished	739	449	291	291	2590	1 finished	1410	291	291	2590
3 finished	738	422	317	317	3107	3 finished	1384	317	317	3107
4 finished	832	516	317	317	3124	4 finished	1384	317	317	3124
7 finished	758	434	325	325	3249	7 finished	1376	325	325	3249
9 finished	812	470	343	343	3342	9 finished	1358	343	343	3342
2 finished	817	462	356	356	3398	2 finished	1345	356	356	3398
6 finished	782	432	351	351	3449	6 finished	1350	351	351	3449
0 finished	872	426	447	447	3546	0 finished	1254	447	447	3546
8 finished	913	467	447	447	3643	8 finished	1254	447	447	3643
5 finished	935	433	503	503	3696	5 finished	1198	503	503	3696

Although the tasks consume more CPU cycles to finish and APET is adjusted in smaller steps, the RTM case shows again a much better ratio of finished tasks to cancelled tasks. The CPU utilization in the RTM case is always better than that in the NO-RTM case while wasting significantly less CPU time. For example, when D = 4000, the RTM case misses only one deadline, whereas the NO-RTM case cancels 9 of 10 tasks (since their deadlines cannot be met according to WPET).

C. Third Sequence of Experiments

In the third sequence of measurements, WPET is set to be twice as large as PET, and **APETV**'s for the ten tasks are computed as follows.

Task	**APETV** (1)	**APETV** (2)	**APETV** (3)	**APETV** (4)	**APETV** (5)
0	152	224	178	167	196
1	179	164	169	144	240
2	125	113	189	157	258
3	142	70	135	112	121
4	14	235	142	121	237
5	148	217	124	224	198
6	9	101	155	181	205
7	132	125	255	164	233
8	131	20	143	152	153
9	76	142	161	178	193

During this sequence, 6 experiments are conducted while varying deadlines to be 1800, 5500, 6500, 7500, 8000, and 8250, and keeping **APETV** and WPET unchanged.

Deadline = 1800										
RTM	CPU_util = 0.12 wasted 5.17 %					**NO_RTM**	CPU_util = 0.12 wasted 5.17 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
2 XXX	1701	1701	842	0	100	2 XXX	1701	842	0	100
3 XXX	1701	1701	580	0	100	3 XXX	1701	580	0	100
4 XXX	1701	1701	749	0	100	4 XXX	1701	749	0	100
5 XXX	1701	1701	911	0	100	5 XXX	1701	911	0	100
6 XXX	1701	1701	651	0	100	6 XXX	1701	651	0	100
7 XXX	1701	1701	909	0	100	7 XXX	1701	909	0	100
8 XXX	1701	1701	599	0	100	8 XXX	1701	599	0	100
9 XXX	1701	1701	750	0	100	9 XXX	1701	750	0	100
1 XXX	1701	1651	896	50	150	1 XXX	1651	896	50	150
0 finished	1243	327	917	917	966	0 finished	784	917	917	966

Deadline = 5500										
RTM	CPU_util = 0.65 wasted 100.00 %					**NO_RTM**	CPU_util = 0.59 wasted 80.35 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
5 XXX	1246	796	911	450	4705	4 XXX	1301	749	400	4200
0 XXX	1257	757	917	500	4744	5 XXX	1301	911	400	4200
1 XXX	1224	724	896	500	4777	6 XXX	1301	651	400	4200
7 XXX	1138	638	909	500	4863	7 XXX	1301	909	400	4200
6 XXX	968	468	651	500	5033	8 XXX	1301	599	400	4200
8 XXX	968	468	599	500	5033	9 XXX	1301	750	400	4200
3 XXX	981	431	580	550	5070	1 XXX	1251	896	450	4250
9 XXX	901	401	750	500	5100	2 XXX	1251	842	450	4250
4 XXX	1034	484	749	550	5100	3 XXX	1251	580	450	4250
2 XXX	1106	506	842	600	5150	0 finished	784	917	917	4666

Deadline = 6500										
RTM	CPU_util = 0.79 wasted 70.38 %					**NO_RTM**	CPU_util = 0.72 wasted 83.81 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
3 finished	981	402	580	580	5679	6 XXX	1201	651	500	5300
8 finished	968	370	599	599	5928	7 XXX	1201	909	500	5300
2 XXX	1106	506	842	600	5995	8 XXX	1201	599	500	5300
9 XXX	1079	479	750	600	6022	9 XXX	1201	750	500	5300
4 XXX	1034	434	749	600	6067	1 XXX	1151	896	550	5350
7 XXX	1034	434	909	600	6067	2 XXX	1151	842	550	5350
0 XXX	1076	426	917	650	6075	3 XXX	1151	580	550	5350
1 XXX	1034	384	896	650	6117	4 XXX	1151	749	550	5350
5 XXX	1011	361	911	650	6140	5 XXX	1151	911	550	5350
6 finished	968	318	651	651	6179	0 finished	784	917	917	5666

Deadline = 7500										
RTM	CPU_util = 0.91 wasted 41.83 %					**NO_RTM**	CPU_util = 0.85 wasted 68.69 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
3 finished	981	402	580	580	5679	3 finished	1121	580	580	5679
8 finished	968	370	599	599	5928	8 finished	1102	599	599	5928
6 finished	968	318	651	651	6629	2 XXX	1051	842	650	6479
4 finished	1034	286	749	749	6928	4 XXX	1051	749	650	6479
0 XXX	1243	493	917	750	7008	5 XXX	1051	911	650	6479
5 XXX	1235	485	911	750	7016	6 XXX	1051	651	650	6479
7 XXX	1198	448	909	750	7053	7 XXX	1051	909	650	6479
1 XXX	1178	428	896	750	7073	9 XXX	1051	750	650	6479
9 finished	1079	330	750	750	7078	1 XXX	1001	896	700	6529
2 finished	1106	265	842	842	7170	0 finished	784	917	917	6695

Deadline = 8000										
RTM	CPU_util = 0.98 wasted 33.91 %					**NO_RTM**	CPU_util = 0.92 wasted 51.41 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
3 finished	981	402	580	580	5679	3 finished	1121	580	580	5679
8 finished	968	370	599	599	5928	8 finished	1102	599	599	5928
6 finished	968	318	651	651	6629	6 finished	1050	651	651	6629
4 finished	1034	286	749	749	6928	4 finished	952	749	749	6928
9 finished	1079	330	750	750	7078	9 XXX	1001	750	700	7029
2 finished	1106	265	842	842	7470	1 XXX	951	896	750	7079
5 XXX	1235	385	911	850	7616	2 XXX	951	842	750	7079
7 XXX	1198	348	909	850	7653	5 XXX	951	911	750	7079
0 XXX	1243	343	917	900	7658	7 XXX	951	909	750	7079
1 finished	1178	283	896	896	7666	0 finished	784	917	917	7195

Deadline = 8250										
RTM	CPU_util = 1.00 wasted 0.00 %					**NO_RTM**	CPU_util = 0.95 wasted 42.97 %			
task	WPET	RPET	PET	EPET	time	task	RPET	PET	EPET	time
3 finished	981	402	580	580	5679	3 finished	1121	580	580	5679
8 finished	968	370	599	599	5928	8 finished	1102	599	599	5928
6 finished	968	318	651	651	6629	6 finished	1050	651	651	6629
4 finished	1034	286	749	749	6928	4 finished	952	749	749	6928
9 finished	1079	330	750	750	7078	9 finished	951	750	750	7078
2 finished	1106	265	842	842	7470	1 XXX	901	896	800	7379
1 finished	1178	283	896	896	7666	2 XXX	901	842	800	7379
0 finished	1243	327	917	917	7783	5 XXX	901	911	800	7379
5 finished	1235	325	911	911	7794	7 XXX	901	909	800	7379
7 finished	1198	290	909	909	7803	0 finished	784	917	917	7445

$D = 1800$ is selected to demonstrate an extreme case in which both the system with and without the RTM process only one task while throwing out all the others. An anomaly occurs when $D = 5500$; the system with the RTM attempts to process all 10 tasks but misses all deadlines, while the system without the RTM throws 9 out of 10 tasks very early and, therefore, saves resources to finish one task in time. However, this occurs very rarely, and all our subsequent measurements show that even when the difference between WPET and APET is small, the RTM case finishes more tasks and wastes less CPU cycles than the NO-RTM case.

VII. 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 exe-

cution 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 on-line whether or not task deadlines can be met.

The TMP measures precisely the EPET of each task by separating RSD from the actual task execution time. The measured EPETs are then used to 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. Various experiments with a set of tasks show that the ratio of tasks finished in time to tasks missing their deadlines is significantly higher when the system is equipped with the proposed real-time monitor. Moreover, the CPU utilization is improved significantly while the percentage of wasted time due to cancellation of tasks or missing deadlines is decreased. 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 1) the scheduler not use any CPU cycles of the host system and 2) 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.

Appendix

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.

References

[1] D. Haban and W. Weigel, "Global events and global breakpoints in distributed systems," in *Proc. 21st Hawaii Int. Conf. System Sciences*, vol. II, Jan. 1988, pp. 166–175.

[2] D. Haban and D. Wybranietz, "Monitoring and measuring parallel systems," in *Proc. 3rd Annu. Parallel Processing Symp.*, vol. 2, Mar. 1989, pp. 499–513.

[3] ——, "A hybrid monitor for behavior and performance analysis of distributed systems," *IEEE Trans. Software Eng. (Special Issue on Experimental Computer Science)*, vol. 16, no. 2, pp. 197–211, Feb. 1990.

[4] J. Nehmer *et al.*, "Key concepts of the INCAS multicomputer project," *IEEE Trans. Software Eng.*, vol. SE-13, no. 8, pp. 913–923, 1987.

[5] D. Peng and K. G. Shin, "Static allocation of periodic tasks with precedence constraints in distributed real-time systems," in *Proc. 9th Int. Conf. Distributed Computing Systems*, June 1989, pp. 190–198.

[6] K. G. Shin and H. Lee, "Port manipulator for the distributed realization of an integrated manufacturing system," *Int. J. Comput. Syst. Sci. Eng.*, vol. 3, no. 1, pp. 21–31, Jan. 1988.

[7] K. G. Shin and Y. K. Muthuswamy, "Message communications in a distributed real-time system with a polled bus," in *Proc. 22nd Annu. Hawaii Int. Conf. System Sciences*, vol. II, Jan. 1989, pp. 703–711.

[8] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "A unified method for evaluating real-time computer controllers and its applications," *IEEE Trans. Automat. Contr.*, vol. AC-30, no. 4, pp. 357–366, Apr. 1985.

[9] M. H. Woodbury, "Analysis of execution time of real-time tasks," in *Proc. 1986 Real-Time Systems Symp.*, Dec. 1986, pp. 89–96.

Dieter Haban received the Diploma and Ph.D. degrees in computer science from the University of Kaiserslautern, West Germany, in 1984 and 1988, respectively.

From 1984 to 1988 he was Faculty Research Assistant in Computer Science at the University of Kaiserslautern. In 1988 and 1989, he was a visiting researcher at the International Computer Science Institute at the University of California, Berkeley, where he was engaged in the areas of very large distributed systems and real-time systems. Currently, he is with the research center of Daimler-Benz AG, Stuttgart, West Germany. He also gives lectures on distributed systems at the University of Stuttgart. His research interests are centered around distributed and parallel systems, operating systems, programming environments, computer integrated manufacturing, debugging, and monitoring.

Dr. Haban is a member of the German Computer Society (GI).

Kang G. Shin (S'75–M'78–SM'83) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

He is a Professor in the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, which he joined in 1982. He has been very active and authored/coauthored over 180 technical papers in the areas of fault-tolerant computing, distributed real-time computing, computer architecture, and robotics and automation. In 1987, he received the Outstanding Paper Award from the IEEE Transactions on Automatic Control for a paper on robot trajectory planning. He also received the Research Excellence Award from the University of Michigan in 1989. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are currently building a 19-node hexagonal mesh multicomputer, called **HARTS**, to validate various architectures and analytic results in the area of distributed real-time computing. From 1970 to 1972 he served in the Korean Army as an ROTC officer and from 1972 to 1974 he was on the research staff of the Korea Institute of Science and Technology, Seoul, Korea, working on the design of VHF/UHF communication systems. From 1978 to 1982 he was an Assistant Professor at Rensselaer Polytechnic Institute, Troy, NY. He was also a visiting scientist at the U.S. Airforce Flight Dynamics Laboratory in Summer 1979 and at Bell Laboratories, Holmdel, NJ, in Summer 1980. During the 1988–1989 academic year, he was a Visiting Professor in the CS Division, Department of Electrical Engineering and Computer Science, UC Berkeley.

Dr. Shin was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chairman of the 1987 RTSS, and the Guest Editor of the 1987 August special issue of IEEE Transactions on Computers on Real-Time Systems. He is a Distinguished Visitor of the IEEE Computer Society. He is a member of ACM, Sigma Xi, and Phi Kappa Phi.