An Approach for Monitoring Intrusion Removal in Real Time Systems [*]

Vishal Jain
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
vjain@cs.pitt.edu

Madalene Spezialetti
Computer Science Department
Trinity College
Hartford, CT 06106
mspezial@starbase.cs.trincoll.edu

Rajiv Gupta
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
gupta@cs.pitt.edu

Abstract

To assist in the development of a real-time application, monitoring is used to collect execution timing information for the application. In this paper we propose a strategy that accurately reports timing information by accounting for intrusion introduced by monitoring. In addition, by allowing processes that miss deadlines to run to completion, our approach provides the user with times by which the execution of these processes exceeds their deadlines. This information can be used to guide the user in restructuring the application to meet timing requirements.

1 Introduction

Developing and testing real time programs is a complex task due to the need to insure that the computation meets both its logical and timing requirements. The various modules which comprise the computation are frequently developed and tested in isolation for logical correctness and to acquire Worse Case Estimate Times (WCET). Modules are subsequently executed in an integrated fashion to test the interaction and timing among them. This integrated testing is complicated by the fact that errors in the logic or timing of one module can effect other modules, causing them to display erroneous behavior.

The runtime monitoring of the modules' behavior as they execute can provide a valuable tool in debugging logical errors as well as assessing the accuracy of the WCETs. The runtime monitoring can collect information on such aspects as message arrival and contents, external input reception, the execution time of various statements and statement groups and changes in variable status [3, 4, 5]. However, a significant drawback of runtime monitoring is that it introduces *intrusion*, that is, since the monitoring activities utilize the same resources as the application modules, the monitoring actions delay the progress of the monitored application. This intrusion is critical when studying real time tasks, as it will increase the execution time of the modules being monitored as well as potentially alter synchronization activities among the modules.

In static scheduling in which each task is allotted a fixed period in which to execute, one approach is to perform monitoring of a task as possible within its allocated period [5]. However, a problem arises if insufficient time is available for monitoring, leading to an inability to carry out the required monitoring in the time period allotted. To increase the effectiveness of a runtime monitoring tool, the tool must provide the user with a view of the module's behavior which does not include timing perturbations caused by monitoring intrusion, while allowing the monitoring system the time needed to capture the information required.

In this paper, we describe such a tool which captures a view of the task execution with the intrusive effects of the monitoring activity removed. We then extend the intrusion removal techniques to facilitate the analysis of WCETs during integrated module testing. The technique allows the modules to continue to execute even if they exceed their deadlines in order to provide a more comprehensive view of the potential error conditions, while preventing the timing perturbations such overruns may cause from effecting the timing analysis of other tasks. While this paper focuses on the use of these techniques in a uniprocessor system using static scheduling, they are also applicable to multiprocessor environments as well as dynamic scheduling.

2 Monitoring Intrusion Removal

A monitoring system for real time tasks may capture data at the system level by default, such as information related to CPU usage or blocked time, or may capture

[*]Supported in part by the National Science Foundation through Grant CCR-9212020 and a Presidential Young Investigator Award CCR-9157371, NASA/Connecticut Space Grant Consortium and Trinity College

data according to directives inserted by a user into the task code. In the latter case, for example, the user may specify the beginning and ending points of a particular sequence of actions which should be timed. In either case, points during the execution of a task will trigger the capture, storage or communication of information by the monitoring system.

Under static scheduling, which is assumed here, a time period is assigned to each task in which the task is expected to complete execution. However, in the presence of runtime monitoring, the activity of the task will trigger the execution of monitoring actions. Since time passes irrespective of whether application or monitoring activity is occurring, time which was allocated for the execution of the application task will be utilized by the monitoring activities, potentially causing the monitored task to exceed its WCET or miss its deadline. To accurately capture the true time of a task's execution, a means must be provided by which the time utilized by monitoring actions can be differentiated from the execution time required by the task. To achieve this end, we maintain a view of the current time which is composed of two components: the actual *clock time* and the amount of that time which was used in the execution of monitoring related activities, referred to as the *intrusion time*.

Definition: The **local time** at processing site S during the execution of an instrumented version of a task, denoted by LT^{in}, is given by the value pair (CT, Δ), where CT is the **clock time** of S obtained from S's internal clock and Δ is the current **intrusion time** of S. Thus, the estimate of the clock time at S during the execution of the original uninstrumented program, denoted by LT^{o}, is given by $CT - \Delta$.

Directly prior to the execution of any monitoring activity, the time will be saved and directly after the monitoring activity completes the time used in the execution of monitoring actions will be added to the local intrusion time, as shown in *StartMonitoring* and *EndMonitoring* in Figure 1. Thus, as activity occurs which is related solely to the original task, only the clock time will increase and when activity related to monitoring occurs, including activity required to execute the intrusion removal algorithm, both the clock time and the intrusion time will advance. When each task begins execution it invokes *StartTask* which calculates and saves the time at which the task would have begun in the absence of intrusion. *EndTask*, which is invoked when a task terminates, derives the actual execution time of the task and also determines if the task exceeded its WCET or would have missed its deadline in the absence of monitoring.

As an example, consider Figure 2, in which task T_1

```
**StartMonitoring** { $(t_{bm}, \Delta_{bm}) = LocalTime()$ }

**EndMonitoring** {
    $t_{em} = ClockTime();$
    $\Delta = (t_{em} - t_{bm}) + \Delta_{bm}$
}

**StartTask** { $t_{start_i} = (ClockTime() - \Delta)$ }

**EndTask** {
    $t_{end} = (ClockTime() - \Delta);$
    $t_{exec_i} = t_{end} - t_{start_i};$
    **if** $(t_{exec_i} > WCET()or(t_{end} > deadline_i)$
        { raise exception }
}
```

Figure 1. Maintaining Monitoring Intrusion Times.

begins execution at time 0 and completes at time 11 after experiencing 3 units of intrusion. In this case, the local time would initially be (0,0) and at the completion of T_1 the local time is (11,3), reflecting T's true execution time of 8. It should be noted that although the monitoring actions caused the execution of T_1 to appear to exceed its WCET and to overrun its deadline, the adjusted calculations will reflect the true execution of the task with the monitoring intrusion removed. It should also be noted that the task whose execution would directly follow T_1 would likewise recognize its true start time as 8, and determine its subsequent true execution time and completion time accordingly.

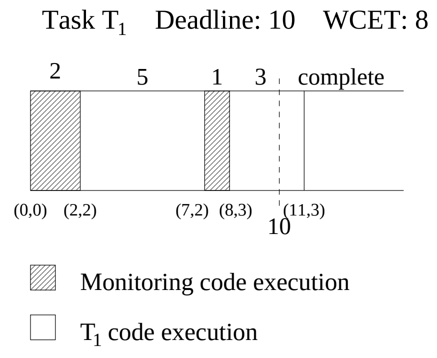


Figure 2. Task Execution with Monitoring intrusion.

The notion of intrusion is generalizable and can be extended to apply to any activity in a real time environment which should be screened from effecting other tasks. Thus, the same notions of intrusion and intrusion removal can be extended to provide a more flexible, comprehensive methodology for analyzing WCETs.

3 Worst Case Execution Analysis

Consider the execution of the series of tasks for WCET analysis, whose intended performance is shown in Figure 3a. Assume that during testing T_2 exceeded its WCET and missed its deadline of 15, as shown in Figure 3b. One option is to abort T_2 and then change T_2, its WCET or its deadline. The task sequence would then be re-executed to determine if the adjustments were sufficient, resulting in an iterative process of testing and adjustment. An alternative approach would be to allow T_2 to continue running even after exceeding its WCET to obtain a more thorough view of the behavior of all tasks before making timing adjustments or alterations, as in Figure 3c. However, delays in the start of subsequent tasks would be caused by T_2's failure to meet its deadline, potentially leading to erroneous conclusions regarding the timing performance of later tasks. In addition, a mechanism must be provided by which to distinguish those WCETs or deadlines which were exceeded due to overruns by preceding tasks from those which represent actual execution overruns. For example in Figure 3c, the delay to T_3 caused by the overrun of T_2 leads to the erroneous appearance that T_3 missed its deadline of 25. In contrast T_4's execution did exceed its WCET and it would have missed its deadline, irrespective of the behavior of T_2. Thus, if all tasks in a series are permitted to complete in spite of execution overruns, it is necessary to provide a means by which to identify task overruns and determine their length.

Task	Deadline	WCET
T_1	10	5
T_2	15	10
T_3	25	10
T_4	35	10

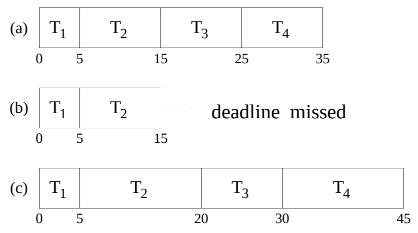


Figure 3. Execution for WCET.

By treating execution overruns as a form of intrusion, not only can they be permitted to occur, but a means is provided by which they can be identified and their effects can be isolated from effecting subsequent tasks. Thus, the tasks can be run in sequence until completion, giving a more comprehensive view of the system behavior. This approach also provides the opportunity to collect such data as the length of an overrun or the total CPU usage or waiting time, which would not be available if the tasks were aborted.

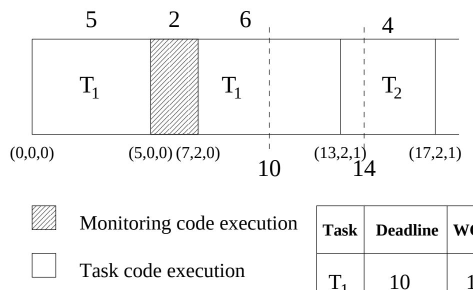
To that end, the notion of local time presented above can be extended to the form $(CT, \Delta_{mon}, \Delta_{over})$, where Δ_{over} represents the amount of intrusion which has occurred due to the execution overrun of tasks. In maintaining Δ_{over}, the perceived effects of execution overruns are bounded by the task's WCET, thus any portion of the execution, exclusive of monitoring induced intrusion, which exceeds the boundaries of a task's WCET is considered overrun intrusion, and Δ_{over} is incremented accordingly. Thus, in the enhanced *Start-Task*, which is shown in Figure 4, the adjusted start time for a task will reflect the sum of the adjusted termination points of all preceding tasks which met their WCETs and the maximum execution up to the WCET boundaries for all preceding tasks whose actual execution overran their estimates. Thus, when determining the adjusted finish time of a task and its length of execution, the calculation will accurately reflect the task's behavior with the intrusion from any monitoring and preceding task overruns removed.

```
Start-Task { $t_{start_i} = (CT - \Delta_{mon}) - \Delta_{over}$ }

End-Task {
    $t_{finish_i} = (CT - \Delta_{mon}) - \Delta_{over}$;
    $t_{exec_i} = t_{finish_i} - t_{start_i}$;
    if ($t_{exec_i} > WCET_i$) {
        $wcover_i = t_{exec_i} - WCET_i$;
        $\Delta_{over} = \Delta_{over} + wcover_i$ }
    if ($t_{finish} > deadline_i$) {
        $dlineover_i = (t_{finish_i} - deadline_i)$ }
}
```

Figure 4. Maintaining Monitoring and Overrun Intrusion Times.

As an example, consider Figure 5. T_1's execution is delayed by 2 units of monitoring and it completed at 13, 3 units over its WCET. However, after adjusting for the monitoring intrusion, the task execution still required 11 units, which exceeds its WCET, thus it missed its deadline of 10 by 1 unit. At the completion of the task, the local time of (13,2,1) indicates that the current time is 13, of which 2 units are monitoring intrusion and 1 unit is execution overrun. Thus, while T_1's contribution to the overall timing of the task sequence is bounded at its WCET, information regarding the amount by which T_1 exceeded its WCET and deadline can be reported to the user. When T_2 begins execution it will utilize the local time of (13,2,1) to determine that its adjusted start time is 10, and upon completion, will correctly determine from the local time (17,2,1) that it completed within its WCET of 4 and its deadline of 14.



Task	Deadline	WCET
T_1	10	10
T_2	14	4

Figure 5. Execution Sequence with Monitoring and Overrun Intrusion.

4 Dynamic Scheduling

Although the approach for removing execution overrun intrusion presented assumes static scheduling, it can also be extended for dynamic scheduling with appropriate alterations made in the updating of overrun intrusion. In particular, since a blocked task may be taken off the CPU to allow another task to run, if a process blocks after exceeding its deadline the time during which it is blocked will be added to its execution time but will not contribute to Δ_{over}. The use of the local time must also be incorporated into the scheduling algorithms, so that appropriate decisions can be made with regard to parameters used in determining priorities, such as the remaining time until a task's deadline.

5 Distributed Environments

The methods presented in this paper are also being extended to a distributed models. Three types of intrusion has been identified: execution intrusion [2], scheduling intrusion [6, 7, 8], and communication intrusion [9, 10]. Although techniques have been developed to track and avoid these intrusive effects in non real-time environments, our solution based upon intrusion times can also be used for real-time applications. In a distributed environment separate intrusion times must be maintained for each processor. Execution overruns and monitoring actions at one site can result in intrusion at remote sites with which the site communicates (directly or indirectly). Thus, the intrusion times must be used across processors to accommodate the effects of varying intrusion times on inter-processor communication. Using the above techniques we intend to provide the user with accurate busy-idle execution profiles [1] of a real-time application.

References

[1] R. Gupta and M. Spezialetti, "A Compact Task Graph Representation for Real-Time Scheduling", *Real Time Systems* journal, vol. 11, no. 1, pages 71-102, 1996.

[2] R. Gupta and M. Spezialetti, "Dynamic Techniques for Minimizing the Intrusive Affects of Monitoring Actions," *IEEE-CS 15th International Conference on Distributed Computing Systems*, pages 368-376, Vancouver, Canada, June 1995.

[3] F. Jahanian and A. Goyal, "A formalism for monitoring RT constraints at run time," *Proc. Fault-Tolerant Computing Symposium (FTCS-20)*, pages 148-155, June 1990.

[4] S. Raju, R. Rajkumar, and F. Jahanian, "Monitoring Timing Constraints in Distributed Real Time Systems," *Proc. of RTSS*, pages 57-67, 1992.

[5] H. Tokuda, M. Koreta, and C. Mercer, "A Real-time Monitor for a Distributed Real Time Operating System," *ACM Sigplan Notices*, vol. 24, no.1, pages 68-77, January 1989.

[6] W. Wu, M. Spezialetti, and R. Gupta, "Designing a Non-intrusive Monitoring Tool for Developing Complex Distributed Applications," *Second IEEE International Conference on Engineering of Complex Computer Systems*, pages 450-457, Montreal, Canada, October 1996.

[7] W. Wu, M. Spezialetti, and R. Gupta, "Guaranteed Intrusion Removal from Monitored Distributed Applications," *Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 422-425, New Orleans, Louisiana, October 1996.

[8] W. Wu, M. Spezialetti, and R. Gupta, "On-line Avoidance of the Intrusive Affects of Monitoring on Runtime Scheduling Decisions," *IEEE-CS 16th International Conference on Distributed Computing Systems*, pages 216-223, Hong Kong, May 1996.

[9] W. Wu, M. Spezialetti, and R. Gupta, "On-line Avoidance of Communication Intrusion in Token Ring Networks," *Technical Report TR-96-07*, University of Pittsburgh, March 1996.

[10] W. Wu, M. Spezialetti, and R. Gupta, "On Intrusive Effects of Monitoring Distributed Systems," *Technical Report TR-96-09*, University of Pittsburgh, April 1996.