Preemptive uniprocessor scheduling for mixed-criticality systems

Bipasa Chattopadhyay

Abstract—In this paper we implement the EDF-VD (Earliest Deadline First with Virtual Deadline) algorithm described in [1] for scheduling mixed-criticality, real-time task systems on preemptive uniprocessor platforms. We compare the scheduling overheads of this algorithm with the scheduling overheads of the well known EDF (Earliest Deadline First) algorithm for scheduling real-time task systems on preemptive uniprocessor platforms.

I. INTRODUCTION

Safety-critical real-time systems are subject to stringent certification requirements. These systems have various functionalities, but not all of the functionalities are safety-critical. For example, on an aircraft, which is an example of a safety-critical real-time system, the navigation system is safety-critical, whereas the entertainment system is not. The certification authority (CA) is only concerned with the correctness of the safety-critical functionalities of the system. The CAs tend to be very conservative and require that the safety critical functionalities be shown to be correct at a very high level of assurance; the remaining (non safety-critical) functionalities are usually validated correct by a system designer at lower levels of assurance.

In order to certify that a system is correct, the certification authority (CA) mandates certain assumptions about the worst-case behavior of the system during run-time. The worst-case behavior of a system is quantified by estimating the worst-case execution time (WCET) of the real-time tasks running in the system. Traditionally the more confidence one needs in a task’s WCET, the larger and more conservative the WCET tends to be in practice. Since, the CA has to certify the system at a high level of assurance, the CA’s WCET estimates tend to be larger than the WCET estimates during the design process. A consequence of this is that the same real-time task is characterized by different WCET parameters. Different values of WCET may be obtained by using different execution-time analysis tools.

Context and Related work. An approach for certification-aware scheduling of real-time tasks is discussed in [1]. If a task needs to be certified by the CA then the task is called a high criticality task and if a task does not need to be certified by the CA then it is called a low criticality task. All tasks have a high criticality and low criticality WCET. The high WCET is used by the CA to certify the correctness of the high criticality tasks. The low criticality WCET is used by the system designer to validate all tasks (high and low criticality). The algorithm presented in [1] is a uniprocessor scheduling algorithm and is called EDF-VD (Earliest Deadline First with Virtual Deadlines). Further work has been done to use the EDF-VD algorithm on multiprocessor platforms. In [5] and [2] scheduling algorithms for global and partitioned EDF-VD on multiprocessors have been described. In this paper we implement the uniprocessor EDF-VD algorithm for dual criticality systems that only have high and low criticality tasks. Note, that the notion of criticality can be extended to several levels. In fact, the DO-178B standard that is widely used in the avionics domain specifies five different levels (A:-catastrophic, B:-hazardous, C:-major, D:-minor, E:-no effect). Prior work on multiple criticality systems for multiprocessor platforms has been done. In [4] the authors implement and describe a mixed criticality system for multiprocessors. In this system the scheduling algorithm used for scheduling tasks at different criticality levels is different. For example, level A tasks are scheduled as per cyclic executive (offline) scheduling and level D tasks are scheduled as per global EDF for multiprocessors. In the
global and partitioned EDF-VD algorithms proposed in [5] and [2] respectively, all tasks are scheduled as per EDF-VD. The authors in [5] and [2] hope to extend the global and partitioned EDF-VD algorithms to incorporate multiple criticality levels. We hope that the implementation presented in this paper can be extended to implement the scheduling algorithms for global and partitioned EDF-VD on multiprocessors.

**Organization.** The rest of the paper is organized as follows. In Section II we describe the mixed criticality task system model. In Section III we provide an overview of the uniprocessors EDF-VD algorithm. A detailed outline of our implementation of algorithm EDF-VD is presented in Section IV. In Section V we describe the experiments and the present the results. Section VI gives a summary of the work.

II. TASK-SYSTEM MODEL

A mixed criticality (MC) implicit-deadline sporadic task is an extension of the (non-MC) implicit-deadline sporadic task described in [6]. Each MC implicit-deadline sporadic task \( \tau_k \) is characterized by a four-tuple \((x_k, C_k(LO), C_k(HI), T_k)\), with the following interpretation:

- \( x_k \) - the criticality of the task. In our task model the criticality of a task either high(HI) or low(LO).
- \( C_k(LO) \) - low criticality WCET.
- \( C_k(HI) \) - high criticality WCET, \( C_k(LO) \leq C_k(HI) \).
- \( T_k \) - period of the task. Task \( \tau_k \) generates an unbounded number of jobs. The period \( T_k \) of a task is the minimum time interval between the arrival of two successive jobs. Each job has a deadline that is \( T_k \) time units after its release.

An MC implicit-deadline sporadic task system is a finite collection of MC implicit-deadline sporadic tasks. Each MC task produces a sequence of jobs. We refer to a job of task \( \tau_k \) as \( J_k \). If a job \( J_k \), that is currently running, executes for less than \( C_k(LO) \) time units then we say that the system is in low criticality. If job \( J_k \) executes for more than \( C_k(LO) \) time units then the system switches to high criticality. If job \( J_k \) executed for more than \( C_k(HI) \) then the system is exhibiting erroneous behavior. We assume that \( C_k(HI) \) is an upper bound on the WCET of all tasks and no task executes for more than \( C_k(HI) \) time units i.e the system does not exhibit erroneous behavior.

The EDF-VD algorithm for scheduling MC task systems is correct if:

- all jobs receive enough execution time to complete execution between their arrival and deadline when the system is in low criticality
- all high criticality jobs receive enough execution time to complete execution between their arrival and deadline when the system is in high criticality.

Note, that in the EDF-VD algorithm low criticality jobs are not executed when the system is in high criticality.

**Utilization parameters** During the pre-processing phase of algorithm EDF-VD, described in Section III, the sum of the utilizations of all the MC tasks in a given MC task system is used to determine whether the task system is schedulable. The utilization of a regular (non-MC) task is defined as the ratio of its WCET to its period. It is essentially the amount of computing capacity a job of a task may utilize when executing on the processor. In the case of MC tasks the notion of utilization is slightly modified. When the system is in low criticality the utilization of a MC task \( \tau_k \) is the ratio of \( C_k(LO) \) and \( T_k \). When the system is in high criticality the utilization of a low criticality MC task is 0 because low criticality jobs are not executed in high criticality and the utilization of a high criticality MC task \( \tau_k \) is the ratio of \( C_k(HI) \) and \( T_k \). Let, \( \tau \) denote a MC task system. For each of \( x \) and \( y \) in \{LO, HI\}, we define a utilization parameter:

\[
U^y_x(\tau) = \sum_{i \in \tau \land x_i=x} \frac{C_i(y)}{T_i}
\]

Thus, \( U^y_{HI}(\tau) \) denotes the sum of the utilizations of the high criticality tasks in \( \tau \), under the assumption that the system is in low criticality.
III. AN OVERVIEW OF ALGORITHM EDF-VD

The EDF-VD algorithm is presented and analyzed in [1]. In this section we briefly describe the algorithm. The EDF-VD algorithm has two phases the pre-runtime processing phase and the runtime processing phase.

Pre-runtime processing

The pre-runtime processing of Algorithm EDF-VD is shown in Figure 1. This is a schedulability test to check whether a given task system $\tau$ is schedulable prior to runtime. It has been proved in [1] that if the pre-processing phase is a success then the task system $\tau$ is schedulable. If a task system $\tau$ is schedulable, then an additional parameter which is called a modified period denoted $\hat{T}_i$ is computed for each high criticality task $\tau_i \in \tau$. For all high criticality tasks $\hat{T}_i \leq T_i$.

Given MC task system, $\tau = \tau_1, \tau_2, \ldots, \tau_n$

1) Compute $x$ as follows:

$$x = \frac{U_{LO}(\tau)}{1 - U_{LO}(\tau)}$$

2)

if $(xU_{LO}(\tau) + U_{HI}(\tau) \leq 1)$

$\hat{T}_i = x.T_i$ for each high criticality task $\tau_i$

declare success and return

else declare failure and return

Figure 1. EDF-VD: The preprocessing phase

Runtime processing

During runtime the system executes as follows:

- Initially the system is in low criticality
- While the system is in low criticality,
  - Suppose a job of some task $\tau_i \in \tau$ arrives at time $t$. If the task is a low criticality task, the job is assigned a scheduling deadline equal to $t + T_i$, which we refer to as the unmodified deadline. If the task is a high criticality task then it is assigned a scheduling deadline equal to $t + \hat{T}_i$, which is called the virtual deadline of the high criticality tasks.

- The job with the earliest scheduling deadline is scheduled.
- If the currently executing job executes for more than its low criticality WCET then the system switches to high criticality.

- Once the system is in high criticality,
  - The scheduling deadline of each high criticality job that is currently ready to execute or executing (we refer to these jobs as ready jobs) is changed to its unmodified deadline. For example, if a high criticality job of task $\tau_i$ is ready then the scheduling deadline of the job is changed to $t + T_i$, where $t$ is the time at which the job arrived and $T_i$ is the period of task $\tau_i$. Note, that this step is necessary because when the system is in low criticality the scheduling deadline of the high criticality jobs is equal to the virtual deadline $(t + \hat{T}_i)$, and a job with the earliest virtual deadline need not have the earliest unmodified deadline. Figure 2 illustrates this.
  - All future high criticality jobs of task $\tau_i$ are assigned a scheduling deadline equal to $t + \hat{T}_i$.
  - Low criticality jobs are not executed when the system is in high criticality.

- If the system is in high criticality and becomes idle, i.e there are no high criticality jobs ready to execute, then the system

Figure 2. Virtual Deadline vs Unmodified Deadline: Let job $J_1$ and job $J_2$ be high criticality jobs. Job $J_1$ has an earlier virtual deadline where as job $J_2$ has an earlier unmodified deadline. When the system is in low criticality job $J_1$ has higher priority than job $J_2$. If there is a switch from low to high criticality at time $s$ then job $J_2$ has higher priority than job $J_1$. 
switches back to low criticality. The algorithm in [1] does not describe how to handle the low criticality jobs that arrived while the system was in high criticality. We assume that these jobs are dropped and not executed when the system reverts to low criticality. Thus, when the system goes back to low criticality the system is just as it was initially.

IV. Implementation

The Pre-runtime processing phase of the EDF-VD algorithm is implemented in user space along with the generation of the task systems that we used for our experiments. In the kernel, we implement the runtime phase. An efficient implementation of the runtime dispatching has been described in [1]. For a task system with \( n \) tasks the runtime complexity is \( O(\log n) \) per event, where an event is an arrival of a job, or the completion of the execution of a job. Our implementation is similar to what is suggested in [1].

The runtime phase in implemented within a real-time operating system, \( LITMUS^{RT} \). \( LITMUS^{RT} \) is an abbreviation of Linux Testbed for Multitiprocessor Real-Time systems. \( LITMUS^{RT} \) is a real-time extension of the Linux kernel with a focus on multiprocessor real-time scheduling and synchronization. Even though most of the work done using \( LITMUS^{RT} \) has focused on extending the Linux kernel for multiprocessor platforms, it can also be used for uniprocessor platforms. Within \( LITMUS^{RT} \) the schedulers are implemented as modular plugins. We add a new uniprocessor scheduler plugin to \( LITMUS^{RT} \) for algorithm EDF-VD.

One of the important functionalities of EDF-VD is to detect a when a job executes for longer than its low criticality worst-case execution time. When this happens the system switches from low to high criticality. We use the precise enforcement timers that are already implemented in \( LITMUS^{RT} \) to implement this functionality. \( LITMUS^{RT} \) keeps track of the amount of execution time received by every job in the system. The remaining execution budget \( e \) of every job is equal to the difference between the low criticality worst-case execution time of the job and the amount of execution time received by the job. When a job begins execution at time \( t \), a timer is set to go off at \( t + e \) time units. If a job continues execution beyond \( t + e \) time units, it implies that the job has executed for over its low criticality worst-case execution time and the system switches to high criticality.

As described in the overview of algorithm EDF-VD, when the system switches to high criticality the deadline of all high criticality jobs that are ready is changed from their virtual deadlines \( (t + \tilde{T}_i) \) to their unmodified deadlines \( (t + T_i) \). In \( LITMUS^{RT} \) the ready jobs are ordered as per decreasing order of deadlines in binomial heaps. The job with the shortest deadline is at the top of the heap. The scheduler picks the job at the top of the binary heap for execution. This is a \( O(1) \) operation. When the high criticality job deadlines are changed the binary heap needs to be re-ordered. Re-ordering a binomial heap involves re-inserting every job in the heap with its new deadline. Inserting into a binomial heap is an \( O(\log n) \) operation, where \( n \) is the number of elements in the binomial heap. In our implementation there can be at most \( n \) jobs in the binomial heap where \( n \) is the number of tasks in the task system. Thus, re-inserting every job in the heap will a \( O(n \log n) \) operation.

In order to avoid incurring a \( O(n \log n) \) cost when switching to high criticality the description of the implementation suggested in [1] uses two ready heaps. Let the two heaps be \( H_l \) and \( H_h \). The \( H_l \) binomial heap has both the high and low criticality ready jobs ordered according to their virtual deadlines. The virtual deadline of a low criticality job is equal to its unmodified deadline. The \( H_h \) binomial heap only has the high criticality tasks ordered according to their unmodified deadline. When the system is in low criticality, a ready low criticality job that is not executing is inserted in the \( H_l \) heap and a ready high criticality job that is not executing is inserted in the \( H_l \) and \( H_h \) heap. The scheduler picks the job at the top of the \( H_l \) heap for execution. When a high criticality job completes execution it is removed from the \( H_h \) heap. Note, that this job is
not in the \( H_l \) heap because it was executing prior to completion. When there is a switch from low to high criticality the scheduler picks the job at the top of the \( H_h \) heap for execution. Therefore, the switch from low to high criticality becomes a \( O(1) \) operation. As long as the system is in high criticality a high criticality job that is not executing is inserted only in the \( H_h \) heap.

In addition to ready heaps, \( LITMUS^{RT} \) has release queues associated with each ready heap. Thus, each ready heap \( H_l \) and \( H_h \) has a release queue \( Q_l \) and \( Q_h \) associated with it. The release queue enqueues all jobs that are to be released. If two or more jobs are to be released at the same time then these jobs are inserted in a heap that is ordered as per the order in which the corresponding ready heap is ordered. For example, all the jobs that are to be released at the same time \( r \) are inserted in a heap ordered by virtual deadline that is enqueued at a queue slot entry associated with time \( r \) of the release queue \( Q_l \) because the ready queue \( H_l \) is ordered as per virtual deadline. When these jobs are released at time \( r \) the release heap is merged with the ready heap. If low and high criticality jobs are released at the same time then they are inserted in the same heap. When a system shifts from high criticality to low criticality we switch to the release queue \( Q_h \) associated with the high criticality ready heap \( H_h \). For this we need to enqueue the release queue \( Q_l \) into the release queue \( Q_h \). However, this enqueue may lead to low criticality jobs being released when the system is in high criticality. Releases incur release overheads. We would like to avoid these overheads. In our implementation we ensure that only high criticality tasks are released in high criticality.

In order to ensure that only high criticality jobs are released in high criticality we use three sets of ready heaps, \( H_{ll}, H_{lh}, \) and \( H_{hh} \) and their corresponding release queues, \( Q_{ll}, Q_{lh}, \) and \( Q_{hh} \). The heap \( H_{ll} \) has only low criticality jobs ordered as per virtual deadline, the heap \( H_{lh} \) has only high criticality jobs ordered as per virtual deadline, and the heap \( H_{hh} \) has only high criticality jobs ordered as per deadline. When the system is in low criticality, a ready low criticality job that is not executing is inserted in the \( H_{ll} \) heap and a ready high criticality job that is not executing is inserted in the \( H_{lh} \) and \( H_{hh} \) heap. The scheduler compares the deadline of the tasks at the top of the \( H_{ll} \) and \( H_{lh} \) heaps and schedules the task with the earlier deadline. When a high criticality job completes execution, it is removed from the \( H_{hh} \) heap. The low criticality jobs that are to be released are enqueued in the \( Q_{ll} \) release queue. And the high criticality jobs that are to be released are enqueued in the \( Q_{lh} \) release queue. When the system switches from low to high criticality the release queue \( Q_{lh} \) is enqueued to the \( Q_{hh} \) release queue. Thus only the high criticality jobs that are to be released are added to the high criticality release queue. As long as the system is in high criticality a high criticality job that is ready but not executing is inserted only in the \( H_{hh} \) ready heap and the to be released high criticality jobs are enqueued in the \( Q_{hh} \) release queue.

In the discussion above when a high criticality job completes execution and the system is in low criticality the job is removed from the \( H_{hh} \) heap. Since it is not necessary that a job with the earliest virtual deadline has the earliest unmodified deadline, we may have to remove this job from any position in the \( H_{hh} \) heap. The operation of removing an element from any position in a heap is already implemented in \( LITMUS^{RT} \).

We have described the important components of the implementation. In the following section we describe the experiments that we performed to analyze the scheduling overheads of this algorithm.

V. EXPERIMENTS

In our experiments we measured the scheduling overheads of EDF-VD and compared it with the scheduling overheads of regular EDF. In low criticality we use timers to detect when a job executes for more than it’s low criticality WCET. We expect that managing the timers adds to the scheduling overheads for algorithm EDF-VD. In high criticality, EDF-VD schedules tasks according to EDF and the overheads are expected to be comparable in that case. Therefore, we compare the overheads
when the system is in low criticality for EDF-VD and EDF.

Our experiments were run on a i7 quad-core system running at 2.67GHz. Since our algorithm is for scheduling on uniprocessors, three of the four cores were disabled. We generated synthetic MC task sets for the experiments. A uniform binary distribution was used to decide whether a task is a low or high criticality task. Therefore, the probability of a task being either a high or low criticality task was 50%. The low criticality utilizations of the MC tasks were drawn from a uniform distribution in the range $[0.01, 0.1]$. If the task was a low criticality task then the high criticality utilization was set equal to the low criticality utilization. If the task was a high criticality task then the high criticality utilization was uniformly drawn from the range $[0.01, 0.1]$, such that high criticality utilization of each task is greater than or equal to its low criticality utilization. The periods of the tasks were generated from three ranges: short $[3, 33]$ milliseconds, moderate $[10, 100]$ milliseconds and, long $[50, 250]$ milliseconds. These period ranges have been used in prior work [3]. The low and high WCETs of the tasks were computed from the utilizations and the periods. For each task system that was generated we checked to see if it was schedulable according to EDF-VD using the pre-runtime processing phase. If the task system was not schedulable then we discarded it. Each job was allowed to execute for half of its low criticality WCET. Therefore, in EDF-VD the system was always in low criticality. Note, that with EDF the utilization of each task is equal to its high criticality utilization. For the task systems we generated it is possible that for some task system the sum of the high criticality utilizations of all tasks is greater than one. Therefore, the task system may be considered un-schedulable as per regular EDF. However, each job is allowed to execute for only half of its low criticality WCET, just as in the case of EDF-VD. Therefore, if the task system is schedulable with EDF-VD then the task system is schedulable with EDF.

To compare the scheduling overheads for EDF-VD and EDF, we generated 100 task systems with 2 tasks each with periods in the short range and measured the scheduling overheads with both algorithms. For each task system the average scheduling overhead incurred over the scheduling decisions was measured with the help of existing tools in LITMUSRT called feather-trace and sched-trace. Using this data the average scheduling overheads for 100 task systems were computed. The scheduling overheads were measured
in microseconds. We repeated the experiment with 4, 6, 8, and 10 tasks per task system and with moderate and long periods. The graphs in Figure 3 show the results of these experiments. To compare the scheduling overheads incurred by tasks of different periods under EDF-VD we plotted the graph shown in Figure 4.

**Observations** From the graphs in Figure 3 we observe that the scheduling overheads incurred by EDF-VD is greater than the scheduling overheads incurred by EDF. This is as expected because EDF-VD uses precise enforcement timers in low criticality. However, the difference in scheduling overheads that we observe is at most 4 microsecond. Therefore, the cost of using timers is not very high. We also observe in Figure 4 that the scheduling overheads with EDF-VD for tasks with short periods is larger than the scheduling overheads for tasks with moderate and long periods. This is because when tasks have short periods they are released more often and cause more checks for preemptions. When a job is released a preemption is needed if the released job has an earlier deadline than the currently executing job. The scheduler checks if a preemption is needed and this adds to the scheduling overheads.

**VI. Summary**

We implemented the EDF-VD preemptive uniprocessor scheduling algorithm for scheduling mixed criticality tasks. The algorithm was described and proved correct in [1]. In this paper we measured the scheduling overheads of the algorithm and compared the scheduling overheads with the preemptive EDF scheduling algorithm for unprocessors. We observed that the scheduling overheads for EDF-VD were greater than the scheduling overheads for EDF. However, the difference in the scheduling overheads was not significant.

**REFERENCES**


