Dynamic Value-Density
For Scheduling Real-Time Systems

Saud A. Aldarmi and *Alan Burns*

Real-Time Systems Group
Department of Computer Science
The University of York
York, YO10 5DD, U.K.

November 1998

Abstract

Scheduling decisions in time-critical systems are very difficult, due to the vast number of systems' parameters and tasks' attributes involved in such decisions. Due to the intractability of the problem, time-critical systems often have to resort to heuristic techniques. Value-based scheduling heuristics have been found in the literature to experience a more graceful degradation under overload situations than various other heuristics. However, value-based scheduling heuristics found in the literature combine the tasks' significance with some of the tasks' *static* attributes, and therefore, they derive *fixed* scheduling priorities. In this study, we propose value-based scheduling heuristics that combine the tasks' significance with some of the tasks' *dynamic* attributes to derive dynamic scheduling priorities, in order to enhance the overall system's performance under normal operating loads and to reduce any performance degradation due to overload situations.

1. Introduction

In order to resolve contention and conflicts over the various resources of a time-critical system; i.e., the *CPU*, the scheduler needs to sequence the execution of the tasks within the system, which may be achieved by establishing a priority ordering among the collection of tasks within the system. Existing scheduling policies establish such ordering by relying on various heuristics, many of which are based on the tasks' deadlines, execution time, the significance of the tasks, and/or a combination of such attributes. Tasks within a time-critical system are designed to accomplish certain service(s) upon execution, and thus, each task has a particular significance (importance) to the overall functionality of the system. Therefore, each and every task within a time critical-system is augmented with an artificial entity known as the task's *value*[1] to the system, which reflects the task's significance to the system. Such an entity instigated what is known as *value-based* scheduling heuristics. Value-based scheduling heuristics found in the literature combine the tasks' values with some of the tasks' *static* attributes, and therefore, they derive *fixed* scheduling priorities. For example, the *Value-Density* (*VD*) [7, 9] is a value-based scheduling scheme that derives a *fixed* scheduling priority relying on the corresponding task's value and expected worst-case execution time.

Generally, many time-critical *CPU* scheduling schemes perform acceptably well under normal operating conditions. However, such an acceptable performance may not pertain under *overload* situations. An overload could occur in many practical real-time systems due to normal system activities in addition to unanticipated emergency conditions and exceptional situations [3, 4, 7]. The presence of an overload requires an amount of processing that can exceed the capacity of the system, thereby it is unable to fulfill its primary objectives; i.e., meeting timing constraints. However, if the underlying scheduling scheme utilizes the *CPU* more efficiently, in particular under overload situations, then the *CPU* will have surplus capacity, which can be redirected towards executing more tasks; hence, enhancing the overall system's performance.

Our goal in this study is not to detect and deal with overload situations, via load management schemes. Rather, it is to investigate, and consequently construct, value-based scheduling heuristics that combine the tasks' values with some of the tasks' *dynamic* attributes; i.e., the tasks' *remaining* execution time, in order to derive *dynamic* scheduling priorities. Consequently, the scheduler will be able to better utilize the *CPU*; hence, spare some of the wasted *CPU* capacity and redirect it towards executing tasks that otherwise would have been lost.

This research focuses on "Soft-Deadline"[2] task scheduling in a uniprocessor environment. Thus, the tasks that complete their execution before their deadlines are considered successful and impart a full value to the system. Whereas tasks that complete after their deadlines are still considered successful, but only impart a portion of their net value that is proportional to their tardiness[3]. The rest of this study is based on the following assumptions:

- All tasks are *aperiodic*.
- Tasks are *independent* of each other, excluding contention for *CPU* access.
- Scheduling is *preemptive*.
- The system's scheduler learns of the following set of attributes *only* at a task's arrival time.

[1] The interested reader may find valuable discussion(s) on values and the manner, in which they are derived, assigned, and manipulated in [1, 2, 5, 7, 9].

[2] Full discussion of real-time models may be found in [1, 7, 9].

[3] Tardiness is the amount of time that a task executes beyond its prescribed deadline.

- C – the total expected execution/computation time. While C represents a task's execution time, $\overline{C}$ represents a task's *remaining* execution time.

- D – the task's deadline.

- I – an *Importance* level, reflecting the tasks' significance to the overall functionality of the system. Note that *Importance* is sometimes known as *Criticalness* in the literature; however, we will use the term *importance* in the rest of this study in order to differentiate between importance (*significance*) of a task to the overall functionality of the system and criticalness in safety-critical systems.

Previous researchers [1, 7, 9] have defined what is called *value-functions* in order to account for the task's importance as well as the task's deadline into the scheduling decision(s). Thus, a value-function allows the *static* importance level of a given task to be made time-variant, thereby *deadline-cognizant*. As shown in figure (1), a task's value, *V*, may correspond directly to the task's level of importance prior to its deadline, and may decrease linearly after the task's deadline. A more detailed discussion of value-functions will follow in a subsequent section.

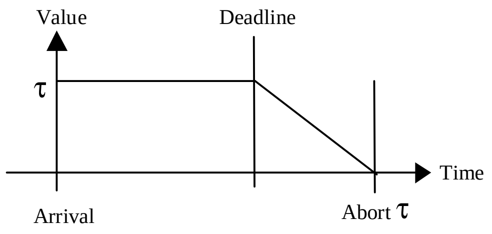

Figure (1)

The remainder of this study is organized as follows: Section 2 introduces a *dynamic* approach to *Value Density*; i.e., *Dynamic Value Density* (*DVD*) and *Dynamic Timeliness Density* (*DTD*). Section 3 presents our simulation model along with the performance metrics being used in this study. Section 4 presents a comparative study contrasting the performance of the newly introduced *CPU* scheduling policies with the traditional *Value Density* (*VD*). Section 5 presents our conclusions.

2. Dynamic Value-Density (*DVD*)

When two tasks are competing for the *CPU*, the scheduler must be sensitive to the tasks' significance; thus, sensitive to their individual values to the system. *Value-Density* (*VD*) [7, 9] is a value-based scheduling scheme, that is known to perform better than many other scheduling algorithms under overload situations [6, 7, 9], in addition to having very low overhead. *VD* is described in function (2.1).

$$Priority\ (P_t) = \frac{Value\ at\ time\ (t)}{Computation\ \ time} \equiv \frac{V_t}{C} \tag{2.1}$$

When a task is submitted to the system, its value is scaled by its expected worst-case computation/execution time in order to derive the task's scheduling priority. Once the scheduling priority is derived, it remains *fixed* until the task's deadline, after which it decreases for tardy tasks[4]. That is, regardless of whether the task remains waiting in the system, or executes for some time period, its scheduling priority remains fixed at its initial merit until its deadline.

The manner, in which *VD* scales the task's value by its execution time, causes all units of execution of a given task to have an equivalent *static* weight, regardless of whether the individual unit(s) have already been processed or remain to be processed. Therefore, *VD* is insensitive to the dynamic status of the task's execution units; hence, function (2.1) represents a *Static Value Density* (*SVD*).

Recall that the system does not collect the value of an executing task until the task is completely finished. That is, if a task executes but never finishes its last execution unit, it does not offer the system any benefit, although it has consumed the system's resources. In order to lower the amount of *CPU* time spent on partially executed tasks, the scheduling priority of an executing task should not only rise before the deadline, but rather, it should also rise even if the task is tardy. Such behavior would counteract any diminishment in the task's value after the deadline and allows an executing tardy task to remain executing. Thus, the priority of a waiting tardy-task decreases after its deadline, but the priority of an executing tardy-task should not decrease; rather, it should increase in order to avoid aborting partially executed tasks; hence, better utilization of *CPU* time. Therefore, we propose altering *VD* as given in function (2.1) such that the task's remaining execution unit(s) *inherit* the weight of the execution units that have already been processed. Consequently, the scheduling priority is not derived *statically* on the task level; rather, it is derived *dynamically* for the individual execution unit(s). Thus, we propose replacing function (2.1) by function (2.2), which represents *Dynamic Value Density* (*DVD*).

$$P_t = \frac{Value\ at\ time\ (t)}{Remaining\ Computation\ \ time} \equiv \frac{V_t}{\overline{C}_t} \tag{2.2}$$

Where $\overline{C}$ is the *remaining* execution time of the corresponding task.

2.1. Static vs. Dynamic Value-Density

The main difference between the *static* approach and the *dynamic* approach of *VD* is in the schedulable unit. The static approach as mentioned above attempts to derive the scheduling priority on the task level, whereas the dynamic approach attempts to derive the scheduling priority on the level of the individual execution units. Thus, all of the execution units of a given task have the same scheduling priority under the static approach; that is, all of the execution units of a given task have the same weight. On the other hand, the execution units of a given task in the dynamic approach have different scheduling priorities. When an execution unit is finished in the dynamic approach, its weight is distributed over the remaining execution units. The priority of a given task corresponds exactly to the priority (weight) of the first *remaining* execution unit. Since each subsequent execution unit has a higher weight than the previous one, then as a task executes, its scheduling priority increases proportionally with the amount of time that the task has executed. The difference in the task's scheduling priority between the two approaches is best described by figure (2), for a task with *V=10* and *C=10*.

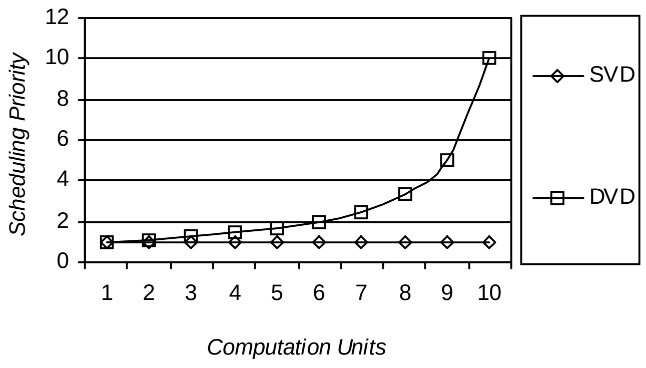

Figure (2)

The consequences of the two approaches affect the overall system's performance in various ways, a matter that will be explained in details in the rest of this section. In general, when a task is preempted by a new arrival, the preempted task has to wait in the system until the completion of the preempting task. However, as the load increases, new tasks arrive into the system at a faster rate, a rate that corresponds to the operating load. Thus, as the load increases, more tasks with higher scheduling priority than the preempted task are more likely to arrive into the system. Consequently, the preempted task is more likely to wait longer in the system before it resumes execution. As the load increases and the preempted task waits longer within the system, it is more likely that the task will become tardy and starts losing its value to the system. Therefore, it is more likely that the preempted task will be aborted under such conditions, after it had already consumed some amount of the system's resources.

Furthermore, if the priority of an executing task does not increase, more tasks with higher scheduling priority than the currently executing task are more likely to arrive into the system as the load increases. Thus, many new arrivals are more likely to preempt the currently executing task. Conse-

[4] A tardy task is a task whose execution extends to beyond its prescribed deadline.

quently, an executing task is easily preempted by new arrivals. Therefore, *SVD* is more likely to experience a relatively high preemption rate, along with the inherited degradation due to context switching overhead and increasing the percentage of partially executed tasks in the system, many of which might be aborted. Thus, it would be a wiser decision to delay the execution of the newly arriving task(s) in order to complete the currently executing task if it had consumed a substantial amount of resources. Such delay should, in theory, reduce the amount of preemption along with its negative consequences. The system's scheduler can impose such delay by increasing the scheduling priority of the currently executing task, which is depicted by the behavior of *DVD*, as shown in figure (2).

2.2. Intensifying the Role of Execution in DVD

The benefits that *DVD* offer over *SVD* are mainly due to the fact that a task starts with a small priority, which rises to the task's level of significance in correspondence with the amount of system's resources; i.e., *CPU* time, that the task has consumed. Thus, there is a gap between the starting priority and its upper bound. To further intensify the enhancement of *DVD* as mentioned in the previous section, we have to widen the gap between the initial scheduling priority of a given task and its final priority. That is, when a task arrives into the system, its initial scheduling priority needs to be extremely small, in order to prohibit the relatively low significance tasks from competing with the currently executing task. When a task starts executing, its priority starts rising at a rate that allows its final scheduling priority to correspond to the task's level of importance. Thus, when a task starts executing, its scheduling priority not only rises, but it needs to rise at a rate faster than the one depicted by $V / \overline{C}$. Such behavior gives even higher preference, than *DVD* as described in function (2.2), to the tasks that have executed over the newly arriving tasks. Thus, *preemption* should be further lowered and *resumption* becomes more likely to happen before a task is aborted. In addition, an executing tardy-task is more likely to continue executing in order to minimize wasting the system's resources. Based on this observation, we map function (2.2); i.e., *DVD-1*, into function (2.3); i.e., *DVD-2*.

$$P_t = \frac{V_t}{\overline{C}_t^{\,2}} \tag{2.3}$$

For a task with *V=20*, *C=10*, and *D=10*, figure (3) shows the scheduling priority of six schemes; i.e., *SVD-w, SVD-e, DVD-1-w, DVD-1-e, DVD-2-w,* and *DVD-2-e,* where '*w*' stands for a *waiting* state and '*e*' stands for an *executing* state. The figure shows four major points:

- Whether a task waits or executes under *SVD*; i.e., *SVD-w* and *SVD-e*, its scheduling priority does not change prior to its deadline. In addition, even if the task starts executing after its deadline, its scheduling priority decreases due to the diminishment of its value. Thus, *SVD* is more likely to suffer a high preemption rate, have a relatively low resumption, and subjecting many partially executed tasks to being aborted.

- If a task waits under *DVD-1*; i.e., *DVD-1-w*, its scheduling priority does not change prior to its deadline. If it continues waiting after its deadline, its priority starts decreasing. If the task starts executing; i.e., *DVD-1-e*, its priority starts rising, whether it starts executing before or after its deadline. However, the task's priority rises at a faster rate before the deadline. Thus, *DVD-1* is more likely to have a relatively low preemption rate, in addition to having high resumption, and avoiding aborting partially executed tasks.

- *DVD-2* mimics the behavior of *DVD-1*, but it starts with a lower initial priority and causes the priority of an executing task to rise at a faster rate than that of *DVD-1*. Thus, *DVD-2* intensifies the behavior of *DVD-1*, which should enhance the behavior of *DVD-1*.

- If $\overline{C}$ increases at a faster rate than that depicted in function (2.3), the performance is even further enhanced due to increasing the gap between the initial priority and its upper bound; i.e., *V*. However, since the starting priority of function (2.3) is very small, then any enhancements achieved by such an increase would be insignificant and might not justify the extra multiplication overhead.

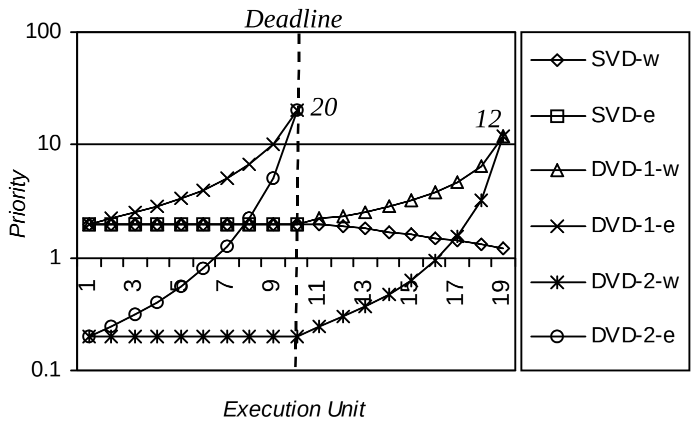

Figure (3)

Note that the rate at which priority rises after the deadline for *DVD-1* and *DVD-2* depends on the rate at which value diminishes after the deadline. Figure (3) is plotted with diminishing speed = 1, which corresponds to the speed at which $\overline{C}$ decreases. Furthermore, the plotted functions after the deadline correspond to waiting before the deadline and starting to execute at the deadline. Finally, for clarity purposes, figure (3) is plotted under logarithmic scale.

2.3. Dynamic Timeliness-Density

The final issue that we need to address in this section is that, if a *waiting* task is subject to becoming tardy due to having an *infeasible* deadline, then it should not receive a relatively high scheduling priority. A task may not be aware of any future interruptions from higher-priority tasks, but (at least) it should be aware, at the scheduling instant, of the relationship between its remaining execution time and its own deadline. Function (2.2) above cannot determine whether a task is subject to becoming tardy due to having an infeasible deadline. Recent studies [2] proposed replacing *value-functions* with *timeliness-functions* for time-critical systems.

The notion of *value* allows a task to hold its level of importance until the task's deadline, after which it starts diminishing according to some monotonically decreasing function. On the other hand, the notion of *timeliness* calls for decreasing the task's significance at $(D - \overline{C})$; both behaviors (notions) are depicted in figure (4) below. *Timeliness* of a given task starts equivalent to the level of importance. However, timeliness starts decreasing when the task's *laxity = 0*, in order to reflect the amount of reduction that the task's significance may experience at the finishing moment. That is, timeliness tells us at the moment that we schedule a task, whether the task is to become tardy and the expected amount of value-reduction at the finishing moment. The reader may refer to [2] for a detailed discussion of *timeliness-functions*.

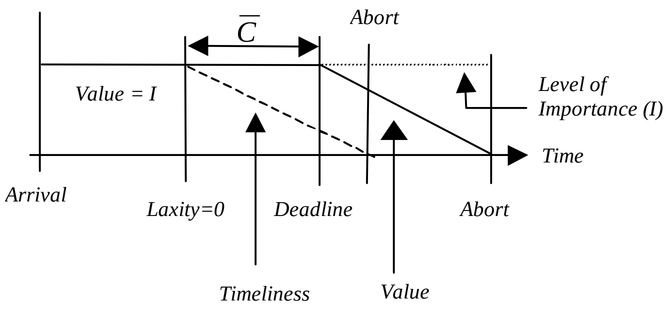

Figure (4)

Combining *timeliness*, $\overline{T}$, as proposed in [2] with *DVD* results in function (2.4), which may be called *Dynamic Timeliness Density* (*DTD*).

$$P_t = \frac{\overline{T}_t}{\overline{C}_t^{\,2}} \tag{2.4}$$

DTD is not only sensitive to the remaining execution time, $\overline{C}$, of a given task such as *DVD*, but also sensitive to the relationship between such an attribute and the task's own deadline. Consequently, *waiting* tasks that are more likely to become tardy are given lower scheduling priorities; therefore, they are not allowed to delay other tasks that have a better chance of meeting their deadlines.

Note that since the scheduling priority of *DVD* as given in functions (2.2) and (2.3), depends on *V* as well as $\overline{C}$ of a given task, then a small $\overline{C}$ for a tardy task could counteract the reduction in *V*. Therefore, although *V* of a tardy task has diminished to a lower value, the task might still receive a relatively high scheduling priority due to its small $\overline{C}$. Consequently, a tardy task could delay the scheduling of a non-tardy task under *DVD*. However, employing timeliness as described above causes the value of a given task to start diminishing at an earlier point in time. Therefore, timeliness does not allow a task to become too tardy in the first place, and hence, the system will not allow a task that is subject to becoming too tardy to delay the scheduling of another task that has a better chance of meeting its deadline. Since tasks get aborted at an earlier point in time when employing timeliness, then the system should be able to save even more *CPU* time. Furthermore, timeliness enables the scheduler to finish executing tasks at an earlier point in time, which enables the scheduler to collect higher values from the set of completed tardy tasks.

In the next section, we describe our simulation model along with its parameters and assumptions, and in the next subsequent section we present a comparative study contrasting the behavior of the three scheduling schemes described above; i.e., *SVD*, *DVD-1, DVD-2,* and *DTD*.

3. Simulation Model

The simulator we use in this paper is based on *CSIM*; a C-based process oriented language [10], and has the following parameters and assumptions.

- The levels of importance are randomly assigned to tasks from a *uniform* distribution from (1.0, 5.0), which may be viewed as {low, mid-low, mid, mid-high, high}.
- Execution times are randomly assigned to tasks from a *uniform* distribution from (1.0, 100.0).
- When a task is submitted to the scheduler, it is assigned a feasible *deadline (D),* such that:

  $D = A + C + uniform\ (3.0, 5.0) \times C$, where *(A)* is a task's arrival time.

- All tasks have a *soft-deadline*, and when a task's (*value*) $V \leq I/100$, the task is assumed to have lost its validity and therefore is aborted; similarly for (*timeliness*), $\overline{T}$.

  Let: $\hat{I}$: The maximum *importance* level within the entire system.

  $\hat{C}$: The maximum *execution* within the entire system.

  S: The *diminishing speed* of a task's value.

  ψ_t: *Tardiness* at time *t*.

In our simulator $S = \hat{I} / \hat{C}$. Therefore, S is a *linear* decay function that diminishes at the same speed for all tasks regardless of the tasks' remaining attributes. Both V and $\overline{T}$ of a given task are computed at time t as follows:

$$\psi_t = max\,(0, t - D).$$
$$V_t = I - \psi_t \times S,$$
$$\overline{T}_t = I - \psi_{(t + \overline{C})} \times S,$$

The reader may refer to [2] for further discussion of the above functions.

- All tasks are *aperiodic*, and scheduling is *preemptive*. Preemption may only occur at the boundaries of single time units. This does not mean that the computation time of any task is a multiple of a single time-unit. Rather, it is possible to have $\overline{C} \leq 1$, but preemption may *not* occur under such condition. Such restriction on preemption is *necessary* due to the fact that the limit of functions (2.2) and (2.3) goes to infinity without this restriction. However, placing and insuring this restriction on preemption, limits the two functions to 'I' (the level of importance) associated with each task within the system. Furthermore, when a task is preempted an artificial delay equivalent to $\hat{C} / 100$ is introduced in order to simulate the overhead of context switching.

- The load simulated is 80% to 200% controlled by an *exponential* distribution for the tasks' arrival, which in turn is controlled by the average execution time (C_a) divided by the desired load (σ). Thus, the submission rate, λ, is computed as follows:

  $$\lambda = \frac{C_a}{\sigma}$$

  Note that an exponential distribution allows for *bursty* arrivals. That is, an exponential distribution may cause a number of tasks to arrive into the system within a very small period of time; i.e., very close to one another. Consequently, even under normal operating load, the system may experience a burst of tasks arriving very close to one another, which results in some tasks missing their deadlines.

- The results of the simulation are collected from one hundred runs, where each run consists of one hundred tasks.

The performance of the simulated techniques is measured according to the following metrics; the interested reader may refer to [2] for a detailed discussion of these metrics.

- Value-sum %, is the percentage of value that the system was able to collect, relative to the total value of all tasks submitted to the system, those that were completed and those that were aborted. Thus,

$$\textit{Value-Sum \%} = \frac{\textit{total value collected} \times 100}{\textit{total value of all tasks submitted to the system}}$$

- Success %, which is the percentage of tasks that are able to complete, whether tardy or on time, relative to the total number of tasks submitted to the system. Thus,

$$\textit{Success \%} = \frac{\textit{total tasks completed} \times 100}{\textit{total tasks submitted to the system}}$$

- *Tardy* %, which is the percentage of tasks that are tardy, relative to the total number of tasks that are completed. Thus,

$$\textit{Tardy \%} = \frac{\textit{total number of tardy tasks} \times 100}{\textit{total number of tasks completed}}$$

- *Tardiness* which is the average lateness of tardy tasks within the system. Thus,

$$\textit{Tardiness} = \frac{\textit{total tardiness of all tardy tasks}}{\textit{total number of tardy tasks}}$$

- *Preemption*, which reflects the total number of preemption(s), normalized with respect to the total number of tasks submitted to the system. Note that some of the preempted tasks may be pre-empted more than once.

$$\textit{Preemption} = \frac{\textit{total number of preemption} \times 100}{\textit{total number of tasks submitted to the system}}$$

- *CPU Wastage %*, which is the percentage of time that was spent (wasted) on tasks that end up being aborted relative to the total amount of time that was spent on all tasks submitted to the system, plus the time spent on all preemption. Thus,

$$\textit{CPU Waste \%} = \frac{(\textit{total time spent on aborted tasks} + \textit{total time spent on preemption}) \times 100}{\textit{total time spent on all tasks}}$$

4. Comparative Study

In this section, we simulate the performance of four scheduling schemes; namely:

- *SVD* as given in function (2.1),

- *DVD-1* as given in function (2.2),
- *DVD-2* as given in function (2.3),
- *DTD* as given in function (2.4), and
- *Earliest Deadline-First* by using *Timeliness (EDF-T)* [2], which is the traditional *EDF* [8], but incorporating timeliness in order to control the instants at which tasks may be aborted. Note that *EDF-T* was shown in [2] to outperform the traditional *EDF*.

New tasks arrive into the system at a rate that corresponds to the operating load. Thus, if the scheduling priority remains fixed, then more tasks with higher scheduling priorities than the preempted tasks are more likely to arrive into the system. Consequently, the preempted tasks are more likely to wait longer in the system before they resume execution. As the load increases and the preempted tasks wait longer within the system, it is more likely that the tasks become tardy and start losing their values to the system. Therefore, it is more likely that the preempted tasks will be aborted under such conditions, after they had already consumed some amount of the system's resources, which will increase the amount of *wasted CPU* time. Such behavior can be clearly seen in figure (5), which shows that *SVD* wastes 5-14% of the *CPU* time to partially executed tasks and preemption as the load increases from 80-200%. Meanwhile, *DTD* wastes as little as 1-2% of the *CPU* capacity as the load increases from 80-200%. The extra 12% that *DTD* is able to re-salvage from the aborted tasks (and preemption) at a load of 200% is actually an extra *CPU* capacity that is redirected towards conducting useful work as will be seen in the rest of this section.

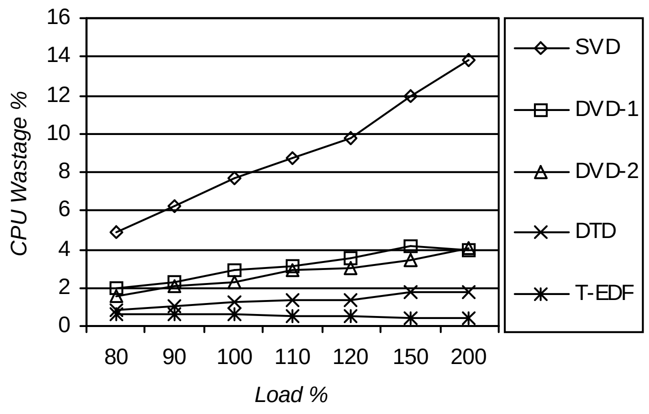

Figure (5)

Figures 6 to 11 show the overall system's performance. Figure (6) shows the total amount of value that the system is able to collect from the completed set of tasks. The figure shows that an enhancement of about 4.5% may be achieved by employing *DTD* instead of *SVD*, due to the following reasons:

- Completing more tasks; i.e., figure (8),

- Lowering preemption; i.e., figure (9).
- Reducing the percentage of tardy tasks; i.e., figure (10),
- Reducing the amount of tardiness; i.e., figure (11),

For clarity purposes, the outlines section in figure (6) is enlarged in figure (7). An important observation that should be noted from figure (7) is that *DTD* does not only outperform the standard *VD* scheme, but also *competes* with *EDF* for operating load ≤ 80%, and *significantly* outperforms *EDF* for all loads > 80%. Hence, *DTD* solves the dilemma of employing *EDF* for normal operating loads and switching to *VD* for overload situations. Rather, a single *CPU* scheduler; i.e., *DTD*, can be employed for all operating loads.

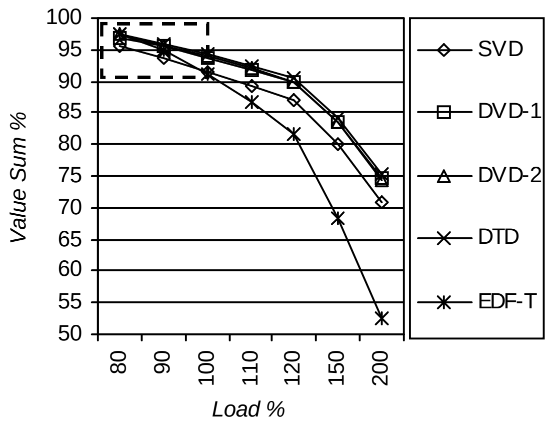

Figure (6)

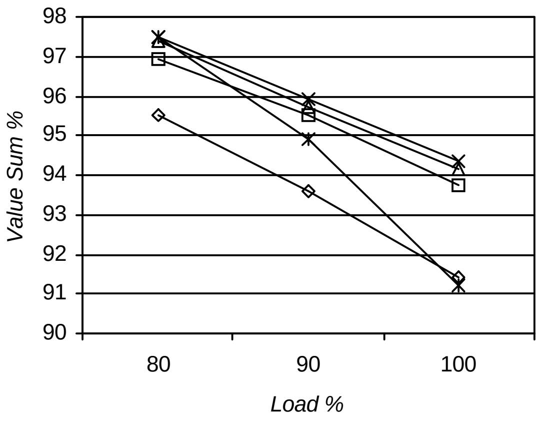

Figure (7)

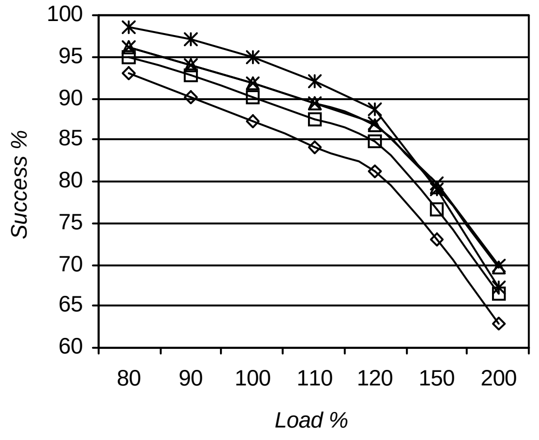

Figure (8)

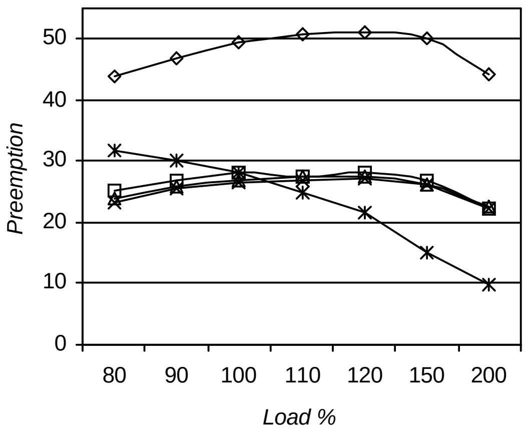

Figure (9)

Figure 9 supports the intuition behind function (2.2) and all subsequent functions given in this study; i.e., using $\overline{C}$ instead of *C*. Earlier in the study we stated that since the priority of an executing task does not increase under *SVD*, more tasks with higher scheduling priority than the currently executing task are more likely to arrive into the system. Therefore, many new arrivals are more likely to preempt the currently executing task. Thus, *SVD* is more likely to experience a relatively high preemption rate, along with the inherited degradation due to context switching overhead and subjecting partially executed tasks to being aborted.

Figures 10 and 11 support our earlier observation, which stated that since the scheduling priority of *DVD* depends on *V* as well as $\overline{C}$ of a given task, then a small $\overline{C}$ for a tardy task could counteract the reduction in *V*. Therefore, although *V* of a tardy is diminished, the task might still receive a relatively high scheduling priority due to the small $\overline{C}$. Consequently, a tardy task could delay the scheduling of a non-tardy task under *DVD*. However, employing *timeliness* to *DVD* does not allow a task to become too tardy, and hence, the system will not allow a task that is subject to becoming too tardy to delay the scheduling of another task that has a better chance of meeting its deadline. Consequently, the percentage of tardy tasks as well as the amount of tardiness should be reduced when employing timeliness as defined in [2] and stated in this study.

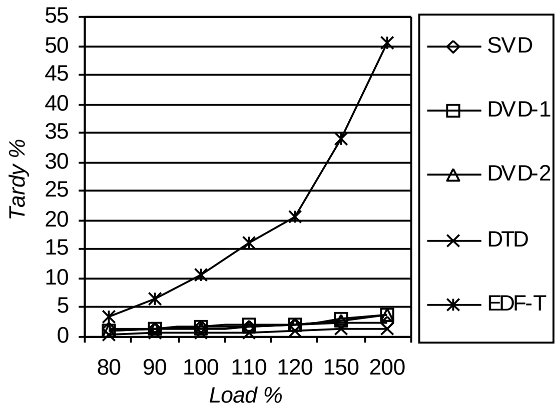

Figure (10)

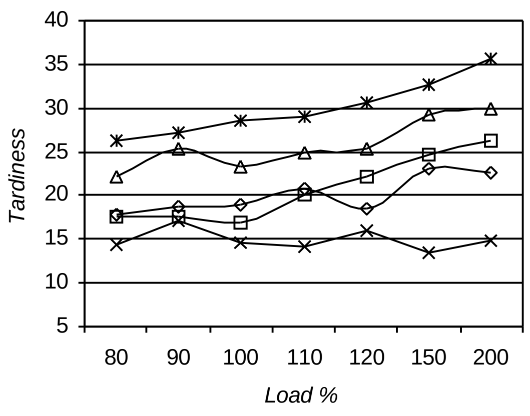

Figure (11)

Note that the performance of *DVD-2* and *DTD* as depicted in figures 5 to 11 is comparable with respect to total number of tasks completed and the total amount of value collected from such set. However, *DTD* utilizes the *CPU* better as reflected in figure (5), subjects less percentage of the tasks within the system to be tardy, and significantly lowers the total amount of tardiness experiences by tardy tasks.

Overhead

From a practical standpoint, *DTD* may be implemented as follows. Let Δ = the time period elapsing from the instant of accessing the *CPU* to the instant at which preemption occurs. When a task is scheduled for the first time, its $C = \overline{C}$. If the task is preempted, then its $\overline{C} = \overline{C} - \Delta$. We can compute Δ in two ways:

- Record the instant at which a task accesses the *CPU* and the instant of preempting the task. Thus, the system's timer needs to be read twice.

- When a task accesses the *CPU*, the scheduler sets a special timer, and when the task is preempted, the timer is stopped and its value reflects Δ. Thus, the timer acts as a stopwatch.

Each approach requires a certain amount of overhead that depends on the underlying operating system and architecture. The implementation of *DTD* in time-critical systems would be more practical in systems where the underlying operating system supports *special* and *budget timers*; i.e., the new proposed extensions for *POSIX* [11].

Recall that the cost of context switching in all of the above simulations = $\hat{C} / 100.$ In order to find the amount of overhead that might render *DTD* to be ineffective, we simulated the system's performance under various overheads. We found that in order for *DTD* to be ineffective, the cost of context switching must increase to $15\hat{C} / 100.$ It is true that *DTD* requires a certain amount of overhead; however, assuming that the overhead of *DTD* is as much as 1500% of the context switching of *SVD* is *unrealistically* overestimated.

Stochastic Execution Time

In many environments, worst-case estimates of execution time may be far from accurate. Thus, in this section we look at the system's behavior if the execution time can be as much as 50% inaccurate. That is, the actual execution time is probabilistic and can be between 50-99% of the worst-case execution time; thus, $\overline{C}$ is always less than *C*.

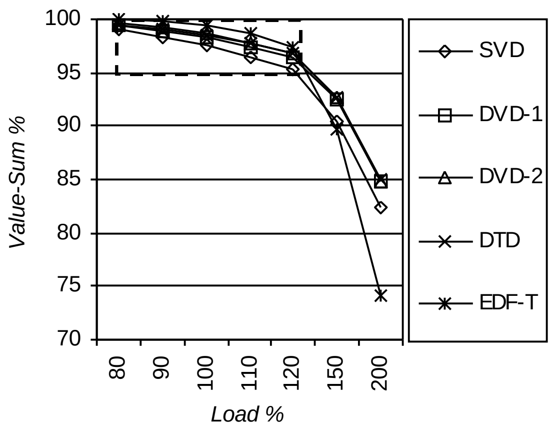

Figure (12-a)

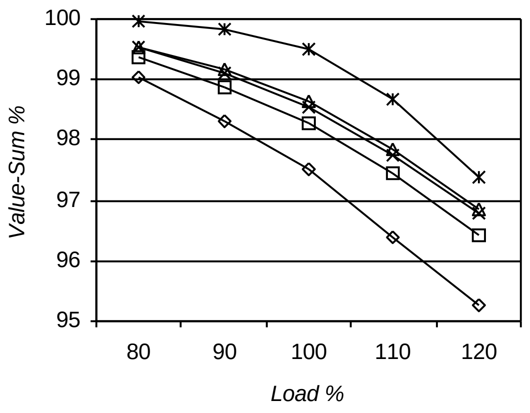

Figure (12-b)

Figure (12) shows the total amount of value earned by the system when $0.5C \leq \overline{C} \leq 0.99C.$ The figure shows that the performance superiority of the schemes being simulated in this study is not as significant as that shown in figure (6). On the other hand, we found that if the deadlines become tight; i.e., the initial laxity becomes small, then the performance becomes more significant than that shown in figure (6).

5. Conclusion

In this study, we constructed a value-based scheduling scheme that combines the tasks' values with some of the tasks' *dynamic* attributes; i.e., the tasks' *remaining* execution time. Therefore, instead of deriving the scheduling priority on the task level, we were able to derive it on the level of the individual computation units. The consequences of such a technique is that the scheduling priority becomes dynamic and not only accounts for the tasks' significance to the system, but also the amount of system's resources that the tasks have already consumed in addition to the amount of resources that they still require until completion.

The dynamic priority forces the newly arriving tasks to wait longer in the system in order to allow the currently partially executed task to finish execution. Consequently, preemption along with its associated degradation is reduced. Therefore, the system can spare an extra *CPU* capacity and redirect it towards executing tasks that otherwise would have been lost. Such an extra capacity was shown in this study to make the performance degradation of the system to be more graceful when operating under overload conditions.

Based on the performance presented in this study, we conclude that *Dynamic Timeliness Density* (*DTD*) it is an effective scheme and it is more suitable to operate under all operating loads than the traditional *Static Value Density* and/or *Earliest Deadline First*.

References

1. R. Abbott, and H. Garcia-Molina, "Scheduling Real-time Transactions: A Performance Evaluation", *Proceedings of the 14th International Conference on Very Large DataBases*, Los Angeles - California (August 1988).

2. S. A. Aldarmi and A. Burns, "Time-Cognizant Value Functions for Dynamic Real-Time Scheduling", *Technical Report YCS-306*, Real-Time Research Group, Department of Computer Science, The University of York, U.K., 1998.

3. S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, "On the Competitiveness of On-Line Real-time Task Scheduling", *Proceedings of the 12th Real-time Systems Symposium*, 1991.

4. S. Baruah, J. Haritsa, and N. Sharma, "On-Line Scheduling to Maximize Task Completion", Proceedings *of IEEE Real-time Systems Symposium*, 1994.

5. A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, L. Strigini, "The Meaning and Role of Value in Scheduling Flexible Real-Time Systems”, To appear in Journal of Systems Architecture, 1998.

6. G. Buttazzo, M. Spuri, and F. Sensini, "Value vs. Deadline Scheduling in Overload Conditions", *Proceedings of IEEE Real-time Systems Symposium*, 1995.

7. E. D. Jensen, C. D. Locke, H. Tokuda, "A Time-Driven Scheduling Model for Real-time Operating Systems", Proceedings *of IEEE Real-time Systems Symposium*, 1985.

8. C. L. Liu, and J. W. Layland, "Scheduling Algorithms for Multiprogramming in Hard-Real Time Environments", *Journal of the ACM*, Vol. 20, No. 1, January 1973.

9. C. D. Locke, “Best-effort Decision Making for Real-time Scheduling”, Ph.D. thesis, Computer Science Department, Carnegie Mellon University, 1986.

10. H. Schwetman, *CSIM* Reference Manual, *info@mesquite.com*, 1994.

11. IEEE draft standard P1003.1d - Draft Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment d: Additional Realtime - Extensions [C Language].