Dynamic Value-Density For Scheduling Real-Time Systems

Saud A. Aldarmi and Alan Burns
Department of Computer Science
The University of York
York, YO10 5DD, United Kingdom
(saud,burns@cs.york.ac.uk)

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. Value-based scheduling heuristics have been found to experience a more graceful degradation under overload situations than various other heuristics. However, currently existing value-based heuristics utilize the tasks' static attributes, and therefore, they derive fixed scheduling priorities. In this paper, we propose value-based scheduling heuristics that utilize the tasks' dynamic attributes in order to enhance the overall system's performance under normal operating loads and to reduce performance degradation under 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 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* to the system (refer to [1, 2, 3, 5, 7] for the notion of value and the manner, in which values can be derived, assigned, and manipulated). Such an entity instigated *value-based* scheduling heuristics. Value-based scheduling heuristics currently found in the literature combine the tasks' values with some of the tasks' *static* attributes, and therefore, they derive *fixed* scheduling priorities. For example, *Value-Density* (*VD*) [5, 7] 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 [5, 7]. The presence of an overload requires an amount of processing that exceeds 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 paper 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" 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 termed *tardy* and only impart a portion of their net value that is proportional to their *tardiness*; i.e., the amount of time that a task executes beyond its deadline. The rest of this paper 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*); i.e., the task's arrival time.
- *C* – the total expected execution/computation time. While *C* represents a task's execution time, $\bar{C}$ 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.

Previous researchers [1, 5, 7] 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 monotonically after the task's deadline. A more detailed discussion of value-functions will follow in a subsequent section.

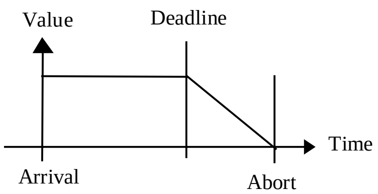

Figure (1)

The remainder of this paper is organized as follows: Section (2) introduces a *dynamic* approach to *Value Density*. Section (3) presents our simulation model along with the performance metrics being used in this paper. Section (4) presents a comparative study exhibiting the effectiveness of the newly introduced *CPU* scheduling policies. 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; and thus, their individual values. *Value-Density* (*VD*) is a value-based scheduling scheme, that is known to outperform many other scheduling algorithms under overload situations [4, 5, 7], in addition to having very low overhead. *VD* is described in function (2.1).

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

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. That is, regardless of whether a 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. To lower such wastage, the scheduling priority should rise in correspondence with the amount of time that a task executes. Such behavior would counteract any diminishment in the task's value after the deadline and allows an executing 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 a *Dynamic Value Density* (*DVD*), where $\bar{C}$ is the *remaining* execution time of the corresponding task.

$$P_{(t)} = \frac{Value\ at\ time\ (t)}{Remaining\ Computation\ time} \equiv \frac{V_{(t)}}{\bar{C}_{(t)}} \qquad (2.2)$$

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, which causes all of the execution units of a given task to have the same scheduling priority. On the other hand, the dynamic approach attempts to derive the scheduling priority on the level of the individual execution units. Thus, 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. Since each subsequent execution unit has a higher weight than the previous one, the 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*.

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.

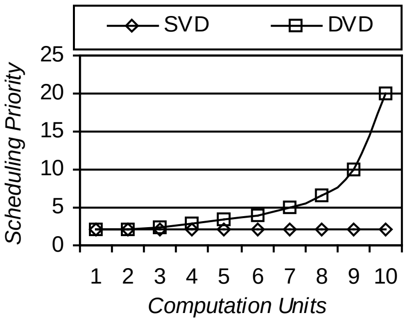

Figure (2)

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. 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, 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. 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 has 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 as depicted by the behavior of *DVD*, as shown in figure (2).

2.2. Intensifying the Role of Execution in DVD

The benefits that *DVD* offers over *SVD* are mainly due to the fact that a task starts with a small priority, which rises 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; i.e., *V*. To further intensify the enhancement of *DVD*, 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 most 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 value. 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 in function (2.2). Such behavior gives even higher preference to the tasks that have executed over the newly arriving tasks. Consequently, *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*.

$$P_{(t)} = \frac{V_{(t)}}{\overline{C}^{2}_{(t)}} \tag{2.3}$$

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. Note that the plotted functions after the deadline correspond to waiting before the deadline and starting to execute at the deadline. The figure shows the following major points:

- Whether a task waits or executes under *SVD*, 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 high preemption rate, relatively low resumption, and subjects many partially executed tasks to being aborted.

- If a task waits under *DVD-1*, 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, 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; i.e., contrast the performance of *DVD-1-e* and *DVD-1-w*. Thus, *DVD-1* is more likely to have a relatively low preemption rate, relatively high resumption, and partially executed tasks are more likely not to be aborted.

- *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*.

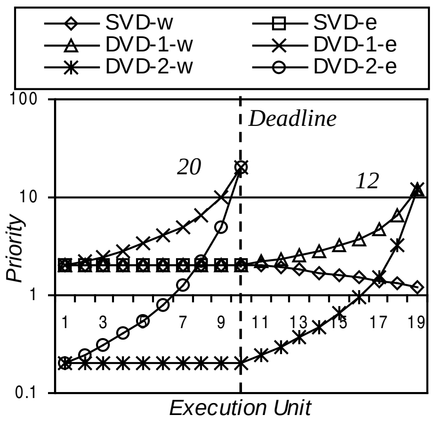

Figure (3)

Note that if $\overline{C}$ increases at a faster rate than that depicted in function (2.3); i.e., $\overline{C}^x$ for all $x > 2$, the performance should further enhance 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.

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 rate = 1, which corresponds to the speed at which $\overline{C}$ decreases. 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 (2.3) 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.

Value-functions allow value to remain constant until the corresponding 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 value at $(D - \overline{C})$; both behaviors (notions) are depicted in figure (4).

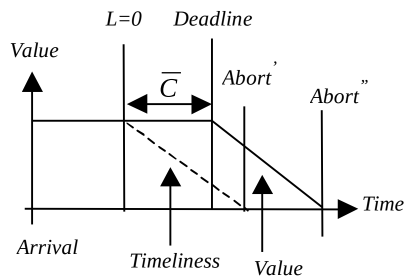

Figure (4)

Timeliness of a given task starts equivalent to *value*. However, timeliness starts decreasing when the task's *laxity (L) = 0*, in order to reflect the amount of reduction that the task's *value* 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*. Combining *timeliness*, $\overline{T}$, as proposed in [2] with *DVD* results in function (2.4), which may be called *Dynamic Timeliness Density* (*DTD*).

$$P_t = \frac{\overline{T}_{(t)}}{\overline{C}^{\,2}_{(t)}} \tag{2.4}$$

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 scheduling schemes described above; i.e., *SVD*, *DVD*, and *DTD*.

3. Simulation Model

The simulator we use in this paper is based on *CSIM*; a C-based process oriented language [8], 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 its *deadline (D)*, such that $D = A + C + uniform\ (3.0, 5.0) \times C$.
- All tasks have a *soft-deadline*, and when a task's (*value*) $V \le I/100$, the task is assumed to have lost its validity, and therefore, it is aborted; similarly for (*timeliness*), $\overline{T}$.
- Define $\hat{I}$, $\hat{C}$, Δ and $\Psi_{(t)}$ to respectively be the maximum *importance* level within the entire system, the maximum *execution* within the entire system, the *diminishing speed* of a task's value, and *tardiness* at time *t*. In our simulator $\Delta = \hat{I} / \hat{C}$; and thus, Δ is a 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:

$$\Psi_{(t)} = max\ (0, t - D).$$
$$V_{(t)} = I - \Psi_{(t)} \times \Delta,$$
$$\overline{T}_{(t)} = I - \Psi_{(t+\overline{C})} \times \Delta,$$

- 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 $\overline{C} \le 1$, but preemption may *not* occur under such condition. Such restriction on preemption is *necessary* due to the fact that the limit of *DVD* and *DTD* goes to infinity without this restriction. However, placing and insuring this restriction on preemption, limits the two functions to the tasks' values. Furthermore, when a task is preempted an artificial delay = $\hat{C} / 100$ 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, $\lambda = C_a/\sigma$. Note that the exponential distribution allows for *bursty* arrivals, which may result in some tasks missing their deadlines even under normal operating load.
- The results represent the average behavior of 10,000 tasks for each simulated load.

3.1. Performance Metrics

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

- *Value-Sum % = total value collected × 100 ÷ total value of all tasks submitted to the system.*
- *Success % = total tasks completed × 100 ÷ total tasks submitted to the system.*
- *Tardy % = total number of tardy tasks × 100 ÷ total number of tasks completed.*
- *Tardiness = total tardiness of all tardy tasks ÷ total number of tardy tasks.*
- *Preemption = total number of preemption × 100 ÷ total number of tasks submitted to the system.*
- *CPU Wastage % = (total time spent on aborted tasks + total time spent on preemption) × 100 ÷ 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* 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* [6], but incorporating timeliness in order to control the instants at which tasks may be aborted. Note that *EDF-T* was shown in [2] to significantly outperform the traditional *EDF*. In the rest of this paper, we use the term *EDF* while implicitly referring to *EDF-T*.

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 as the load increases. Consequently, preempted tasks are more likely to wait longer in the system before they resume execution. Under high operating loads, it is more likely that the preempted 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, 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, the other techniques do not waste more than 4%.

Figures 6 to 10 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 not only shows that an enhancement of about 5% may be achieved by employing *DTD* instead of *SVD*, under overload, 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.

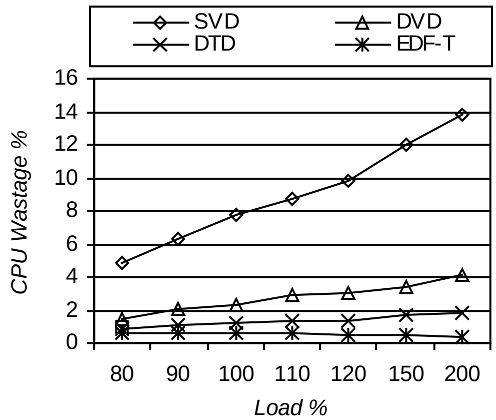

Figure (5)

On the other hand, figure (7) shows that *DVD* and *DTD* are able to reduce the gap between *EDF's* high completion rate and the relatively low rate of value-based schemes. In fact, the two schemes are able to compete and outperform *EDF* for load > 120; meanwhile, *SVD* is not able compete with *EDF* for all loads ≤ 200%.

Figure 8 supports the intuition behind function (2.2) and all subsequent functions given in this paper; i.e., using $\overline{C}$ instead of *C*. Earlier in the paper 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 9 and 10 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 task 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* 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*.

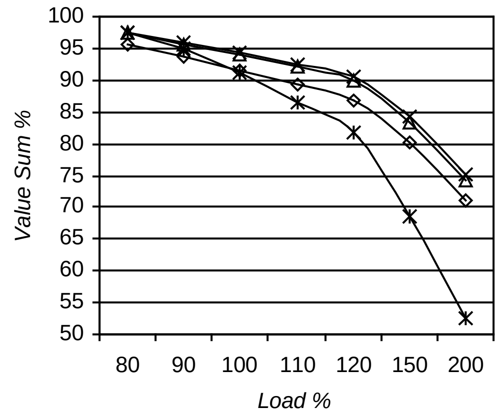

Figure (6)

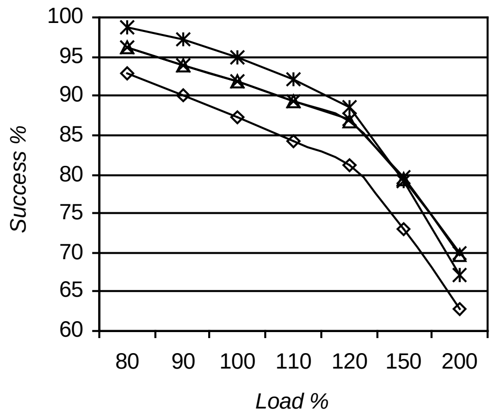

Figure (7)

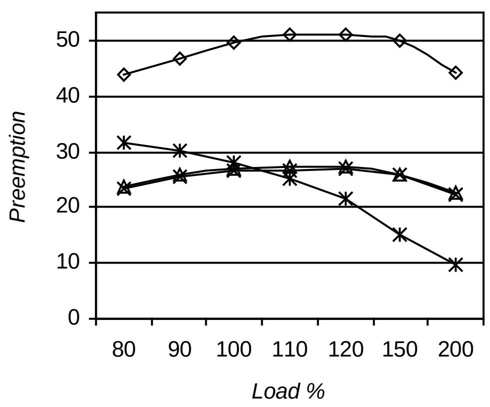

Figure (8)

Note that the performance of *DVD* and *DTD* as depicted in figures 5 to 10 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 becoming tardy, and significantly lowers the total amount of tardiness experienced by tardy tasks.

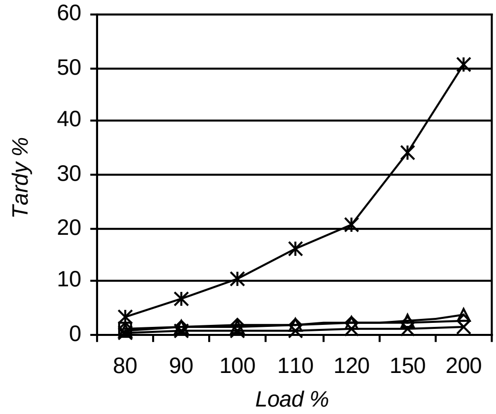

Figure (9)

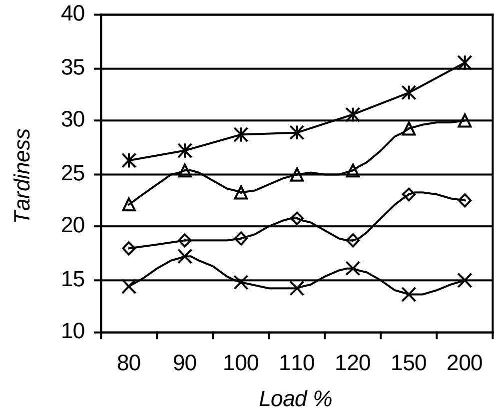

Figure (10)

4.1. Overhead

The implementation of *DTD* in time-critical systems would be more practical in systems where the underlying operating system supports *budget timers*; i.e., the new proposed extensions for *POSIX* [9].

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 an *unrealistic* overestimation.

5. Conclusion

In this paper, 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' value 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(s) 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 paper to make the scheduler's performance degradation to be more graceful when operating under overload conditions.

We conclude that *Dynamic Timeliness Density* (*DTD*) is an effective *CPU* scheduling scheme, and it is more suitable than the traditional *Static Value Density* and/or *Earliest Deadline First*, to operate under all operating loads.

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. 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.
4. G. Buttazzo, M. Spuri, and F. Sensini, "Value vs. Deadline Scheduling in Overload Conditions", *Proceedings of IEEE Real-time Systems Symposium*, 1995.
5. 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.
6. 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.
7. C. D. Locke, "Best-effort Decision Making for Real-time Scheduling", Ph.D. thesis, Computer Science Department, Carnegie Mellon University, 1986.
8. H. Schwetman, *CSIM* Reference Manual, 1994.
9. 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].