Dynamic CPU Scheduling with Imprecise Knowledge of Computation-Time

Saud A. Aldarmi and *Alan Burns*

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

April 1999

Abstract

The majority of the studies conducted in scheduling real-time transactions mostly concentrate on concurrency control protocols, while overlooking the *CPU* as being the primary resource. Consequently, there are various techniques for scheduling the *CPU* in conventional time-critical systems; meanwhile, there does not seem to be any technique that is adequately designed for scheduling such a resource in *Real-Time Database (RTDB)* systems. Existing schemes for scheduling the *CPU* in *RTDB* systems considerably degrade under overload conditions, in addition to suffering a relatively high preemption rate under all operating loads. Coping with overload situations requires accurate knowledge of the computation time, and preemption must be augmented with synchronization protocols, which contribute to performance degradation as data contention rises – aside from context switching overhead. In this study, we construct an efficient *CPU* scheduling scheme that minimizes the preemption rate in order to reduce the frequency by which synchronization protocols must be invoked, along with their inherited performance degradation. In addition, we also introduce a new timing model upon which the newly introduced scheduler is incorporated in order to utilize the system's imprecise knowledge of computation time estimates.

1. Introduction

An *RTDB* system's scheduler is a mechanism for ordering the execution of the outstanding transactions on the processor *(CPU)* as well as the database according to some predefined criteria. Thus, an *RTDB* scheduler consists of two separate modules; i.e., *CPU* scheduler and Concurrency-Control (conflict resolution protocols). While there are several views on the roll of priority cognizance in concurrency control protocols; i.e., resolving contention over the database, priority ordering is necessary for resolving contention and conflicts over the *CPU*.

CPU scheduling algorithms used in current real-time *task* models assume a priori *accurate* knowledge of the tasks' resource requirement; i.e., worst-case computation-time. Such an assumption instigated various scheduling techniques for the task model. Meanwhile, it is nearly impossible to *accurately* determine the *actual* computation-time of a transaction, since the runtime behavior of a transaction is dynamic in the sense that it depends upon the values it finds in the database *at run-time*. Therefore, a worst-case computation-time may be very *pessimistic* in an environment such as an *RTDB*. Thus, *CPU* scheduling schemes that relay heavily on *accurate* computation-time may not perform as well when operating with *imprecise* computation-time estimates. Note that by computation-time we mean the amount of time that a given transaction requires *CPU* access in, a non-preemptive, memory resident database system. Thus, computation-time does not involve any synchronization and/or any of a database inherited delays; e.g., I/O. Many existing schemes for scheduling the *CPU* in *RTDB* systems are mainly based on *Earliest Deadline First (EDF)* [12], *Most Critical First (MCF)* [7, 8], or a combination of the two schemes. Both schemes are shown in this study not to be very suitable for an *RTDB* environment due in turn to their relatively high preemption rate, aside from their inferior performance when operating under overload situations.

Our goal in this study is to investigate, and consequently construct, an adequate *CPU* scheduling scheme that does not rely on accurate knowledge of computation time, while simultaneously minimizing preemption and its undesirable consequences, and thus, most suitable to operate under an *RTDB* environment.

This study focuses on scheduling "Soft-Deadline" transactions. That is, the transactions that complete their execution before their deadlines are considered successful and impart a full value, *V*, to the system[1]. Whereas transactions that complete after their deadlines are known to be *tardy* and only impart a portion of their net value that is proportional to their *tardiness*; i.e., the amount of time that a transaction executes beyond its prescribed deadline. Transaction scheduling, in the broad meaning of the term, consists of *CPU scheduling* and *Concurrency Control* in order to maintain data-consistency as defined by *serializability*. However, due to the vast dimensions of the problem, this study is dedicated to the first half of the problem; i.e., *CPU scheduling*. The rest of this study is based on the following assumptions:

- All transactions are *aperiodic*, and scheduling is *preemptive*,
- The system's scheduler learns of the following set of attributes *only* at (*A*), the transaction's arrival time.

[1] Refer to [1, 2, 5, 8, 13] for the notion of value and the manner in which values may be derived, assigned, and manipulated.

- I – an *importance* level, reflecting the transaction's *significance* to the overall functionality of the system. Every transaction is designed to accomplish a specific function within the system upon its execution. At system design time, each transaction may be assigned a specific level of importance to reflect the transaction's significance with respect to the functionality of the system as a whole[2].

- D^a – a transaction's *absolute* deadline. Every transaction submitted to the system carries a time-tag known as its *deadline*. Deadlines are mostly generated from the environment in which the system operates [15, 16, 19]. As a fundamental definition of *RTDB* systems, every transaction is supposed to have finished execution not later than its deadline for its execution to be of any validity [20]. However, as a fundamental definition of *soft-deadline* models, a transaction may be allowed to execute for some "*limited*" time period beyond its deadline.

- D^r – while D^a represents a transaction's *absolute* deadline, D^r represents a transaction's *relative* deadline, where $D^r = D^a - A$. Note that wherever we use D without a superscript, we are referring to D^r.

The remainder of this study is organized as follows: Section (2) presents previous work and the motivation of this study. Section (3) introduces a new timing model upon which our newly introduced scheduling scheme is incorporated. Section (4) introduces a new scheduling scheme called *Aging Timeliness Cumulative Computation (ATCC)*, which is capable of operating with *imprecise* knowledge of computation-time. Section (5) presents our simulation model along with its assumptions and the performance metrics used in this study. Section (6) presents a comparative study exhibiting the effectiveness of the newly introduced scheme. Section (7) concludes this research.

2. Motivation and Previous Work

RTDB systems can be viewed as an extension to conventional real-time systems. Therefore, there are subtle differences between the two systems as shown in figure (1). Conventional real-time systems can be perceived as the left side of the dashed line in figure (1), where a scheduled task might experience a preempt-resume behavior until termination. Whereas *RTDB* systems also include the right side of the graph where a transaction does not only experience the preempt-resume behavior, but rather is also subject to concurrency control, which causes transactions to block. After a transaction blocks, it might resume its previous execution, rollback and restart its previous execution, or get aborted and terminate its execution. Note that tasks in a conventional real-time system also block on synchronization protocols. However, the blocking period is for the duration of a critical region, whereas blocking in *RTDB* systems is for the duration of the lifetime of the blocking transaction(s). Thus, in order to

---

[2] The term *importance* is sometimes known as *Criticalness* in the literature; however, we use the term *importance* in the rest of this study to differentiate between *significance* of a transaction and *criticalness* in safety-critical systems.

enhance the performance of *RTDB* systems, we must reduce (minimize) the interaction between the *CPU* scheduler and the concurrency control module. The interaction between the two modules can be mainly the result of preemption, and partially, I/O operations. Both preemption and I/O operations cause transactions to wait while holding some resources; e.g., database granules.

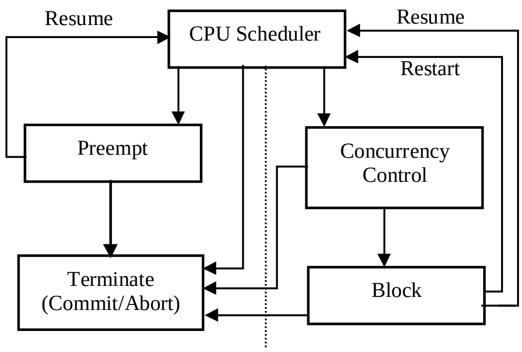

Figure (1)

In the task model, *Value-Density* (*VD*) [8, 13], i.e., V/C, is a value-based scheduling scheme that is known to outperform many other scheduling algorithms under overload situations. 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. On the other hand, *Dynamic Value Density (DVD)* [3], i.e., $V/\overline{C}^{\,2}$ (where $\overline{C}$ represents the remaining computation time), enhances the performance of *VD* by assigning an extremely low priority to newly arriving tasks, and such priority increases in correspondence with the amount of time that a task executes. Consequently, the scheduling priority is not derived *statically* on the task level; rather, it is derived *dynamically* for the individual execution unit(s). The consequences of such behavior are that preemption is significantly lowered, and execution resumption after preemption is also increased. *DVD* was shown in [3] to lower the amount of *CPU* time that is wasted on preemption and aborted, partially executed, tasks and is able to redirect the *CPU* towards executing the tasks that otherwise would have been lost. Hence, better utilization of the *CPU*, thereby achieving higher levels of performance.

In order to resolve contention and conflicts over resources, an *RTDB's* scheduler has to establish priority ordering among the transactions within the system. Priorities can be assigned to real-time transactions in a similar manner as assigning priorities to real-time *tasks*. However, priority assignment schemes using information about runtime behavior; e.g., computation time, have been claimed not to be feasible for a transaction model [6]. Huang et al. [7] introduced a technique that was intended to operate in an *RTDB* environment. The technique was called "*Scheduling by Criticalness and Deadline First*" *(CDF)*, as stated in the following function:

$$Priority\ (P) = \frac{Relative\ Deadline}{Criticalness} \equiv \frac{D}{I} \qquad (2.1)$$

Function (2.1) uses the transactions' *significance* to the system and *relative deadline*, both of which are not concerned with the computation time of the corresponding transaction(s). However, function (2.1) relies on two *static* attributes; and thus, the derived priorities are *static* and *fixed* from the instant that they are derived; i.e., the instant of submission, until the transactions' termination, where termination means commit or abort. Therefore, *CDF* is subject to similar weaknesses and limitations as *VD*. Since transactions dynamically undergo several changes as they migrate through the system, monitoring and utilizing the transactions' dynamic attributes can provide the means of a better *CPU* scheduling scheme; hence, better overall performance. In this study we wish to alter *CDF* and construct a scheduling technique that is capable of *mimicking* the effective behavior of *DVD*, in addition to minimizing the preemption rate in order to suite *RTDB* environments.

3. Imprecise-Timeliness-Functions (*ITF*)

Before we introduce our new scheduling scheme, we wish first in this section to construct a new timing-model; i.e., *Imprecise-Timeliness-Function(s) (ITF)*, upon which our scheduler is incorporated. *ITF* utilize the system's imprecise knowledge of computation-time estimates, and thus, causes the transactions' attributes to be time-cognitive, which provides the *CPU* scheduler with a better sense of timeliness.

If a transaction does not start executing before $t + \bar{C} = D^a$ (for t being the current time), then the transaction is guaranteed to become tardy. Thus, if a *waiting* transaction is subject to becoming tardy due to having an *infeasible* deadline, then it should not receive a relatively high scheduling priority in order not to delay the scheduling of other transactions that have a better chance of meeting their deadlines. *Timeliness-functions* [2] address such an issue, and therefore, they start reducing the system's internal representation of value at $(D^a - \bar{C})$ in order to project, at the scheduling instant, the final value to be gained at the finishing moment. Such scheme attempts to schedule the on-time transactions before the tardy ones.

Define Δ to be the diminishing-rate by which a transaction's value is computed when the transaction becomes tardy, and $\psi_{(t)}$ being a transaction's tardiness at time t. For V and $\bar{T}$ respectively being a transaction's *value* and *timeliness*, timeliness-functions introduced in [2] propose shifting V to the left by $\bar{C}$ to produce $\bar{T}$, both of which are illustrated in figure (1), and respectively computed in functions (3.1) and (3.2).

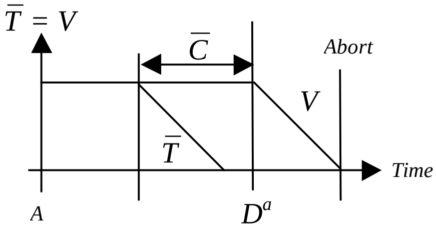

Figure (2)

$$\Psi_{(t)} = max\ (0, t - D^a)$$

$$V_{(t)} = I - [\Delta \times \Psi_{(t)}] \tag{3.1}$$

$$\overline{T}_{(t)} = I - [\Delta \times \Psi_{(t+\overline{C})}] \tag{3.2}$$

The proposed technique works well if the transaction's computation-time estimate is accurate. However, if the actual computation-time is much less than the worst-case estimate, then a transaction might be erroneously aborted; i.e., *aborted prematurely*, meanwhile, the transaction could have finished with a significant value. Hence, since transactions in *RTDB* systems do not generally have accurate computation-time estimates, function (3.2) may be more applicable for conventional real-time systems (the task model) than *RTDB* systems (the transaction model). In the next subsection, we address the different types of transactions that might exist in an *RTDB* system in order to avoid erroneously aborting transactions prematurely.

3.1. Transactions' Characteristics

In general, a transaction's computation-time either depends on the nature and functionality of the transaction, and/or on the environment and operating conditions. Several classes of transactions have been identified in previous studies; e.g., [10, 11], each of which has different characteristics and run-time requirements. If a transaction's body were closely inspected, it would either contain a single path with a fixed functionality; e.g., *canned transactions*, which require a specific runtime, and thus, a *precise* computation-time estimate, or it might contain multiple paths. A compiler can certainly approximate the maximum/minimum-required computation-time, which can be used to generate an expected error bound; i.e., a degree of imprecision, within the transaction's worst-case estimate. Consequently, we classify transactions into two categories, those with *precise* estimates, and those with *imprecise* estimates.

Figure (3.a) shows a transaction whose body consists of a single execution path, and thus, its actual computation-time equals its worst-case computation-time. On the other hand, figure (3.b) shows a transaction whose body consists of multiple paths; note that the figure is simplified for illustration purposes. From figure (3.b) we note the following important issues:

- The period extending from the beginning of a transaction until the current time (t) corresponds to $\tilde{C}$, which is the total amount of time that a transaction has computed so far.
- When the transaction execution reaches point ‘t_1’, the transaction might finish or continue executing to point ‘t_2’, after which it might finish or continue to point ‘t_3’. Thus, the period elapsing from starting the transaction to point ‘t_1’ corresponds to the minimum amount of time that the transaction might require, whereas point ‘t_3’ corresponds to the maximum. Hence, the difference between the two points in time, t_1 and t_3, corresponds to the amount of *imprecision* (error bound) in the transaction’s computation-time estimate; i.e., α. In the rest of this section, we address the two cases of precision.

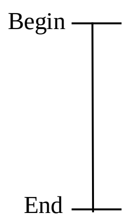

Figure (3.a)

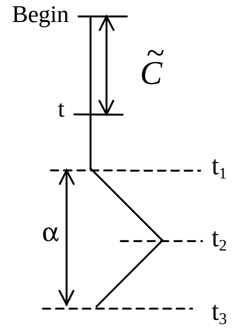

Figure (3.b)

Case (1) – Precise Computation-Time

Based on the model depicted in figure (2) and the assumption of *precise* computation-time; i.e., figure (3.a), it is reasonable to start diminishing a transaction’s timeliness exactly at $(D^a - \bar{C})$, without being subject to erroneously aborting the corresponding transaction(s) prematurely. Such claim is valid since $\bar{C}$ accurately reflects the corresponding transaction’s remaining computation-time.

Case (2) – Imprecise Computation-Time

For a transaction with an imprecise computation-time estimate, timeliness must not use $\bar{C}$ as stated above, but rather use an estimate of $\bar{C}$ that corresponds to the degree of imprecision. In order to do so, define α be the degree of imprecision in the transaction’s computation-time estimate; i.e., error bound, which implies that the minimum computation-time for the corresponding transaction is “$C \times (1-\alpha)$”, which we refer to C_{min}. That is, for a transaction with a computation-time estimate that is imprecise by *25%*, then $\alpha = 0.25$. For example, for a transaction with $C = 100$, and $\alpha = 0.25$, then the *minimum* computation-time $\equiv C_{min} = 75$; and hence, for a transaction with an accurate estimate $\alpha = 0$. Consequently, a transaction’s timeliness can be computed at “$D^a - C_{min}$”. However, such an instant is a fixed point in time, which does not account for $\tilde{C}$ (the amount of time that a transaction has computed so far). Therefore, we must further shift this instant by $\tilde{C}$, and thus, timeliness can be computed at "$D^a - C_{min} - \tilde{C}$". Hence, no transaction will be aborted prematurely. Thus, for $\Psi_{(t)}$ as defined above, timeliness, $\overline{T}$, may be computed as given in function (3.3).

$$\overline{T}_{(t)} = I - [\Delta \times \Psi_{(t + C_{min} - \tilde{C})}] \tag{3.3}$$

3.2. Analysis

In one extreme, with timeliness defined above as such, function (3.3) reduces to function (3.2), whose behavior is depicted in figure (2) for all transactions with *precise* computation-time estimates. On the other extreme, for those transactions with maximal *imprecise* estimates; e.g., 99% imprecision, function (3.3) shifts timeliness practically to D^a; that is, timeliness does not start diminishing until $t = D^a$, which is practically equivalent to function (3.1), whose behavior is also depicted in figure (2). These are the two behaviors of waiting transactions. However, as a transaction starts executing, its $\tilde{C}$ increases in correspondence with t, and $\tilde{C}$ may actually exceed C_{min} resulting in a negative quantity being added to t, which counteracts the corresponding increase in t. Hence, once a transaction starts executing, before or after its deadline, its timeliness remains steady (does not diminish).

The three behaviors; i.e., the two extremes for waiting transactions, and timeliness for an executing transaction, are respectively illustrated in figure (4) at three different instants as illustrated by the dashed lines – where $\hat{\Psi}$ corresponds to the maximum allowable amount of tardiness for the corresponding transaction. Note that if timeliness was reduced prior to the deadline, then it remains steady at the level it was reduced to once the transaction starts executing.

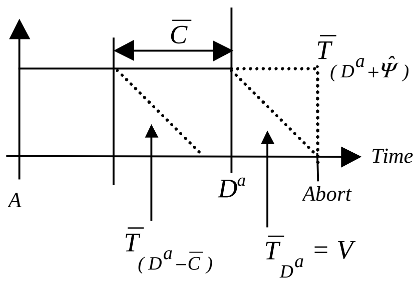

Figure (4)

For all other transactions, timeliness dynamically floats in between the above extremes in correspondence with the transactions' own requirements and level of imprecision. Thus, while previously defined timeliness-functions; i.e., function (3.2), might abort a transaction with a feasible minimum computation-time, *ITF* presented in figure (4) would not, because if the transaction's minimum computation-time were feasible, then its timeliness would not have been reduced to zero (the abort point).

ITF as defined in this study are not as rigid as value-functions or previously defined timeliness-functions. Consequently, they can adapt to a wider range of environments and application domains; e.g., *RTDB* systems. Aside from such advantages, *ITF* do not require any extra overhead over what is required by previously defined timeliness-functions; i.e., keeping track of $\overline{C}$, and thus, $\tilde{C}$. Finally, just as C_{min} is used in function (3.3) to account for the minimum computation-time, we could replace it with the average computation-time, a matter that we do not peruse any further in this study.

4. Aging-Timeliness-Cumulative-Computation (*ATCC*)

In this section, we construct a new *CPU* scheduler that utilizes the timing-model proposed in the previous section. Due to the static nature of *CDF*; i.e., function (2.1), the scheduler becomes unaware of time as a major parameter in *RTDB* systems. Thus, the manner in which *CDF* considers the transactions' significance into its decision renders the system the ability to distinguish between various levels of importance, but causes the system to be incognizant to timing constraints.

Transactions in *RTDB* systems are associated with particular levels of importance, reflecting their significance to the overall functionality of the system. In a *soft-deadline* model, a transaction becomes less important if it does not complete its execution by its prescribed deadline. Fundamentally, if a transaction experiences a substantial amount of delay such that it becomes tardy, its function becomes less important to the system and consequently, its significance should diminish accordingly. Imprecise-Timeliness-Functions *ITF* described in the previous section, can be utilized to account for many of the transaction's attributes; i.e., importance, deadline, amount of resources consumed, and amount of resources yet to be consumed in relation to the transaction's deadline. Thus, *ITF* allow the scheduler's decisions to be made *deadline-cognizant* as illustrated in figure (4). Based on such advantages, *CDF* would be a better scheme if it utilized the transaction's time-variant $\overline{T}$ instead of *I*. In addition, since we wish preference to be given in an *increasing* order of the priorities; i.e., opposite to the order of Huang et al. [7], we alter function (2.1) above as shown in function (4.1) below for all *t*.

$$P = \frac{\overline{T}}{D} \tag{4.1}$$

However, the scheduling priority is still derived on the transaction level. Recall that *DVD* achieves its performance significance (superiority) over the traditional *VD* by increasing the scheduling priority by an amount that corresponds to *V* for every execution unit. Therefore, function (4.1) must also increase by (at least) a corresponding amount if it is to mimic the effectiveness of *DVD*. Adding $(\overline{T} \times \tilde{C})$ to the numerator of function (4.1) as shown in function (4.2) allows the scheduler to weight each execution unit according to the corresponding transaction's timeliness. Therefore, the scheduler not only becomes aware of various valued execution units, but also is able to credit each transaction for all the units it executed with respect to the transaction's own timeliness.

$$P = \frac{\bar{T} + \bar{T} \times \tilde{C}}{D} = \frac{\bar{T} \times [1 + \tilde{C}]}{D} \tag{4.2}$$

Furthermore, transactions in an *RTDB* environment have to block for synchronization, I/O, commit protocols, buffer management, etc. Therefore, we should account for such delays in the scheduling decision. Thus, define ω to be the total amount of time that a transaction has been waiting in the system since its arrival, excluding any $\tilde{C}$. That is, $\omega_{(t)} = t - A - \tilde{C}_{(t)}$, which is a *dynamic* attribute corresponding to *aging*. In order to account for a database inherited delays, we need to add such an attribute to function (4.2). In addition, we must also treat it in the same manner that we treated $\tilde{C}$; i.e., multiply it by $\bar{T}$, in order to distinguish between a waiting highly significant transaction vs. a waiting less-significant transaction. However, if the priority of a waiting transaction increases in a manner similar to that of an executing transaction, then the waiting transaction might interfere with an executing transaction, and *thrashing* would be expected due to the corresponding high preemption rate. Therefore, we need to scale down ω first, then multiply it by $\bar{T}$.

Note that if a transaction is short then it is not going to require a substantial amount of systems' resources and it should be given scheduling preference. We argue that in order to increase the system's throughput, then we must speedup the execution of all short transactions. Based on such an observation we propose scaling ω down by *C*. However, since *C* might involve a certain degree of imprecision, then such imprecision must be utilized when considering *C*. Thus, define $\Omega_{(t)}$ to be a weighted aging factor at time *t* such that:

$$\Omega = \frac{\omega}{C \times (1 - \alpha)} \equiv \frac{\omega}{C_{min}}$$

With Ω defined as such, function (4.2) is mapped into function (4.3) below, which may be called *Aging Timeliness Cumulative Computation* (*ATCC*), owing its name to all the parameters involved in it. Note that function (4.3) utilizes the underlying timing-model by using $\bar{T}$ in weighting $\tilde{C}$ and Ω.

$$ATCC = \frac{\bar{T} \times [1 + \tilde{C} + \Omega]}{D} \tag{4.3}$$

Function (4.3) clearly shows the feasibility of utilizing information about runtime behavior; e.g., *computation-time*, in priority assignment schemes for the transaction model; contrary to previous claim(s); e.g., [6]. The function involves only simple arithmetic, and utilizes the transactions' timeliness (thus value with respect to deadline), in addition to the amount of computation-time that has already been consumed and remains to be acquired. Furthermore, the function also accounts for the amount of time that transactions wait in the system (whether for I/O or concurrency control).

Figure (5) below shows the manner in which *ATCC* derives the scheduling priorities in contrast to *DVD*. In figure (5), '*w*' corresponds to the scheduling priority in a waiting state, and '*e*' corresponds to the scheduling priority in an executing state. The figure shows the scheduling priority of an *executing* and *waiting* transaction(s) before the deadline, and the scheduling priority of an *executing* transaction after the deadline. From figure (5) one can clearly see that the scheduling priority of an executing transaction under *ATCC* increases at a faster rate than the priority of an executing transaction under *DVD*.

Based on the behavior depicted in figure (5), *ATCC* should, in theory, reduce *preemption* and increase a transaction's chance of *resumption* over *DVD*, which in addition to reducing the preemption overhead, it should also reduce the amount of *CPU* time that is wasted on partially executed transactions. Furthermore, the scheduling priority of a waiting transaction does not change under *DVD*, whereas the scheduling priority of a waiting transaction under *ATCC* increases at a rate that avoids *thrashing*.

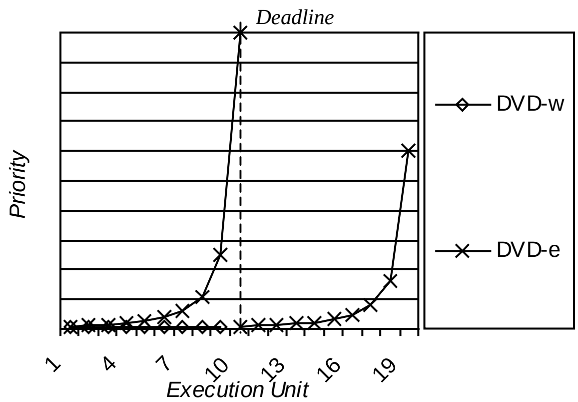

Figure (5.a)

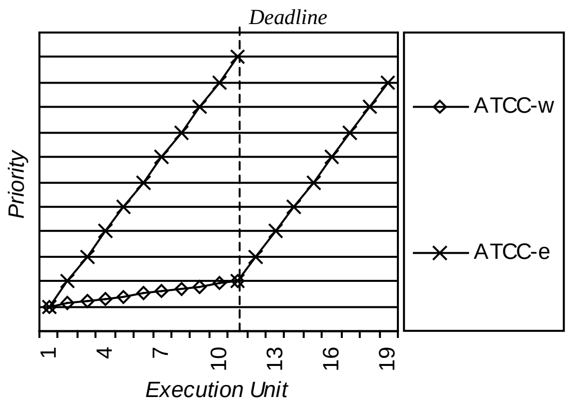

Figure (5.b)

When a transaction is submitted to the system, its priority is computed by function (4.1). Whether the transaction executes or waits, it is reflected in its $\tilde{C}$ or Ω, respectively. When a transaction is restarted due to conflict resolution, its $\tilde{C}$ is set back to zero, because the transaction has to redo all of its work. Thus, once $\tilde{C}$ is set back to zero, its value automatically transfers to ω, because $\omega = t - A - \tilde{C}$. Hence, the time period that a transaction invested into executing is not lost once a transaction restarts. Rather, such investment is only reduced by automatically transferring to ω. Such behavior allows the corresponding transaction to receive a higher scheduling priority than newly arriving transactions, which gives the restarted transaction a better chance of meeting its deadline without jeopardizing the newly arriving transactions since (more likely) they can afford to wait more than the restarted transaction.

In the next section, we present our simulation model along with its assumptions, and in the next subsequent section, we simulate the performance of function (4.3).

5. Simulation Model

The simulator we use in this study is based on *CSIM*; C-based process oriented language [17], and has the following parameters and assumptions.

- The levels of importance are randomly assigned to transactions from a *uniform* distribution from (1.0, 5.0), which may be viewed as {low, mid-low, mid, mid-high, high}.

- The worst-case computation-time, C, is randomly assigned to transactions from a *uniform* distribution from (100.0, 200.0). However, the simulated load has the following characteristics:

- 1/3 of the transactions have *precise* computation-time estimates; and thus, such transactions use their worst-case computation time estimates; i.e., $\alpha = 0$.
- 1/3 of the transactions have *imprecise* computation-time such that $0.5C \leq \hat{C} \leq C$, for $\hat{C}$ being the actual computation time. Thus, $\hat{C}$ of these transactions can be as low as 50% of their worst-case estimates; i.e., $\alpha = 0.5$.
- 1/3 of the transactions have *imprecise* computation-time such that $0.1C \leq \hat{C} \leq C$. Thus, $\hat{C}$ of these transactions can be as low as 10% of their worst-case estimates; i.e., $\alpha = 0.9$.

- The time it takes to process each data granule is 10 time units, and the number of data granules requested by any transaction corresponds to the transaction's actual length; e.g., a transaction that is 100 units long requires 10 locks. However, since we are assuming memory resident database, there is no I/O blocking delays; rather, there is only concurrency control blocking delay.

- When a transaction starts executing, it is forced to request its first data granule. For every unit of execution after the first, the corresponding transaction requests a data granule with a probability of 0.1, and thus, a transaction requests a new data granule every 10 time-units on the average. In order to insure that a transaction requests all of its data granules before its deadline, it is forced to request, at least, one lock every 10 time units. Thus, the first lock is requested at the first execution unit, and the second one can be requested any time before the 10^{th} unit. However, if the transaction does not request the second lock by the 9th unit, it is forced to request it on the next execution unit. The same behavior applies to all subsequent locks.

- All required data granules are requested from the database manager, which employs *Priority-Abort (PA)* [1] for conflict resolution. *PA* simply restarts the lower priority transaction involved in a conflict if it is the lock holder, and it is made to wait if it is the requester. Furthermore, a new reader of a given item may join a group of readers of such an item only if its priority is higher than all transactions (if any) waiting on the requested item.

- When a transaction is submitted to the system, it is assigned its *deadline* (D^a), such that:

  $D^a = A + C + L,$

  $L\ (laxity) = uniform\ (2.0,5.0) \times C$

  Note that the C used in computing a transaction's *laxity* corresponds to the worst-case computation time estimate and not the actual computation time.

- All transactions have *soft-deadline,* and when a transaction's (*value*) $V \leq I/100$, the transaction is assumed to have lost its validity and therefore is aborted. For $\hat{I}$ and $\hat{C}$, respectively, being the maximum importance level and the maximum computation-time required by any transaction within the entire system, the *diminishing speed* (Δ) of a transaction's value = $\hat{I}/\hat{C}$, and thus, the values of all transactions diminish at the same speed.

- All transactions are *aperiodic*, and scheduling is *preemptive*. Furthermore, when a transaction is preempted an artificial delay = $\hat{C}\,/\,100$ is introduced in order to simulate the overhead of context switching.

- The simulated load is 80% to 200% controlled by an *exponential*[3] distribution for the transactions' arrival, which in turn is controlled by the *average actual* computation-time ($\hat{C}_a$) of all transactions divided by the desired load (σ). Thus, the submission rate $\equiv \lambda = \hat{C}_a/\sigma$.

- The results of the simulation correspond to the average behavior of 5,000 transactions at each simulated load.

5.1. Performance Metrics

The performance of the simulation is measured according to the following metrics, and the interested reader may refer to [2] for a detailed discussion such metrics.

[3] Note that an exponential distribution allows for *bursty* arrivals. Consequently, even under normal operating load, the system may experience a burst of transactions arriving very close to one another, which results in some of them missing their deadlines.

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

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

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

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

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

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

- *Tardiness*, the average lateness of completed tardy transactions within the system. Thus,

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

- *ILA%*, the average percentage of the aborted importance levels with respect to the maximum importance level within the system. Thus,

$$\textit{ILA\%} = \frac{\textit{sum of importance levels aborted}}{\textit{total number of transactions aborted}} \times \frac{100}{\textit{maximum importance level}}$$

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

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

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

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

- *Restarts* %, the percentage of transactions that are restarted due to conflict resolution, with respect to the total number of transactions submitted to the system. Thus,

$$Restarts\ \% = \frac{total\ number\ of\ restarted\ transactions \times 100}{total\ number\ of\ transactions\ submitted\ to\ the\ system}$$

- *Wasted Locks* %, the percentage of locks that are wasted due to aborts and restarts with respect to the total number of locks acquired by all transactions within the system. Thus,

$$Wasted\ Locks\ \% = \frac{total\ number\ of\ lost\ locks \times 100}{total\ number\ of\ locks\ acquired\ by\ all\ transactions}$$

6. Comparative Study

In the section, we contrast the performance of function (4.3) against several *CPU* scheduling schemes; namely:

- *Most Critical First (MCF)* [7, 8] – the transaction with the highest significance receives the highest priority,
- *Earliest Deadline First (EDF)* [12] – the transaction with the closest deadline receives the highest priority,
- *Scheduling by Criticalness and Deadline First (CDF)* [7]; i.e., function (2.1),
- *Value-Density (VD)* [8, 13]. Note that *VD* uses worst-case computation time estimates in its priority assignment.
- *Dynamic-Value-Density (DVD)* [3]. Note that we use $(C - \tilde{C})$ in computing the remaining computation time, $\bar{C}$. However, since the actual computation time may differ from the worst-case estimate in *RTDB* environments as well as our simulator, *VD* and *DVD* might make the wrong decision, and this scenario exhibits their limitation in such environments.

Figures (6 to 15) show the overall performance of the scheduler. In general, the figures seem to classify the six scheduling schemes being simulated into three groups: *EDF* being in its own group, while *MCF*, *CDF*, and *VD* seem to be comparable as in one group, and finally, *DVD* and *ATCC* being somewhat comparable as in a separate group.

Figure (6) shows the total amount of *CPU* time that is wasted on preemption and aborted, partially processed, transactions. The figure shows that *EDF*, which increases the scheduling priority of all transactions proportionally, wastes as much as 38% under a load of 200%. Meanwhile, *MCF*, *VD*, and *CDF* waste as much as 10-15% under a load of 80-200%, respectively. On the other hand, while *DVD* waste up to 3% of the *CPU* capacity, *ATCC* wastes as little as 0.19% (practically 0%). The efficiency of *DVD* and *ATCC* in utilizing the *CPU* over the other schemes is also reflected in the rest of the system's performance as depicted in figures (7 through 14).

Figure (7) shows the total amount of value that the scheduler is able to collect from the completed set of transactions. For clarity, we enlarged the outlined section in figure (7.a) into figure (7.b). The figure clearly shows that *ATCC* not only is capable of sustaining higher performance under overload situations; i.e., experiencing a graceful degradation, but also outperforms the remaining simulated schemes under normal operating conditions as well! On the other hand, figures (8.a, 8.b) show the percentage of transactions that the scheduler is able to complete relative to the total amount submitted to the system. The figures also show that while *ATCC* slightly outperforms *DVD*, it significantly outperforms the remaining schemes being simulated. Thus, an efficient *CPU* scheduling can be achieved without accurate knowledge of computation time estimates.

Figure (9) shows that *EDF* subjects as many as 60% of the transactions submitted to the system to being tardy under a load of 200%, while the percentage of tardy transactions does not exceed 7% under the remaining schemes. On the other hand, figure (10) shows that with the exception of *EDF* experiencing the highest amount of tardiness, and *ATCC* experiencing the lowest amount of tardiness, the remaining schemes are somewhat comparable with respect to the amount of tardiness.

Figure (11) shows *EDF* aborts transactions with 50-55% significance levels. Meanwhile, the other schemes seem to abort transactions with comparable significance levels, except that under heavy loads, *ATCC* seems to abort transaction with slightly higher significance level than the remaining schemes.

Figures (12 and 13), respectively, show the preemption rate and the percentage of transactions that are restarted due to conflict resolution. Notice the strong correlation between the two figures! Figure (12) shows the preemption reduction that *ATCC* has over the other schemes, which is *very* important in an environment such as an *RTDB* as reflected in figure (13). If a transaction is preempted while it holds some resources, then the problem of *priority-inversion* [1, 14, 18] becomes an issue, which causes aborting and possibly restarting several conflicting transactions. *Aborting* and *redoing* overhead could accumulate as data-contention rises, which could increase further as preemption rises. Regardless of the concurrency control mechanism being employed (pessimistic or optimistic), a higher preemption rate could significantly contribute to higher amounts of conflict within the system, which translates into performance degradation. Thus, lowering the preemption rate is very important in an *RTDB* system due to the fact that even if data-contention is high between the current set of transactions within the scheduler's queue, due to the nature of the transactions, it is naturally reduced if preemption is reduced. Tailoring the underlying *CPU* scheduling scheme in a manner that minimizes the preemption rate can significantly reduce concurrency control degradation. While figures (6 to 8) show the *direct* enhancement of *ATCC*, figure (12) clearly shows its significant *indirect* enhancement as reflected in figures (13 and 14).

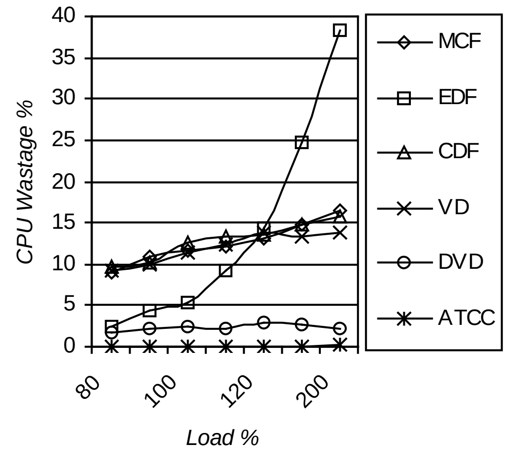

Figure (6)

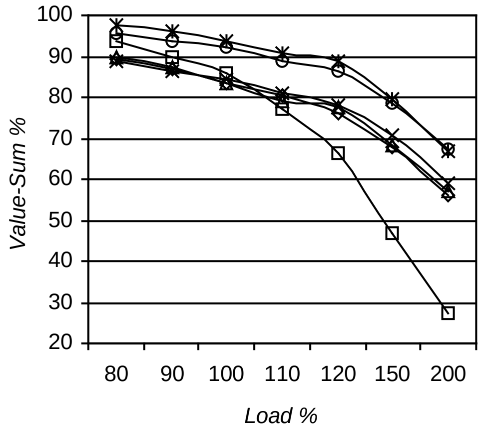

Figure (7.a)

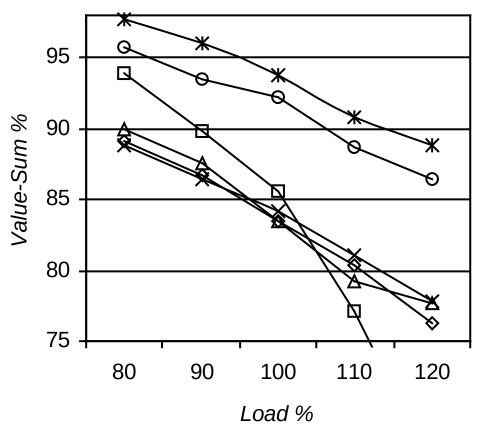

Figure (7.b)

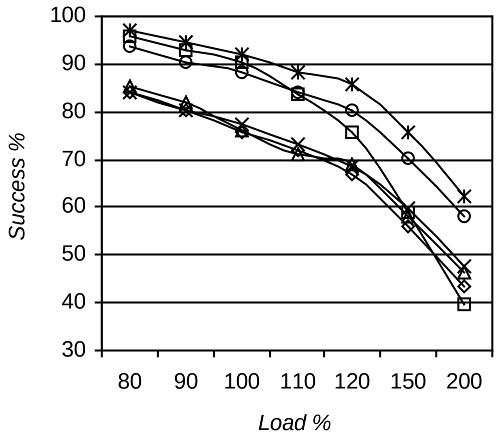

Figure (8.a)

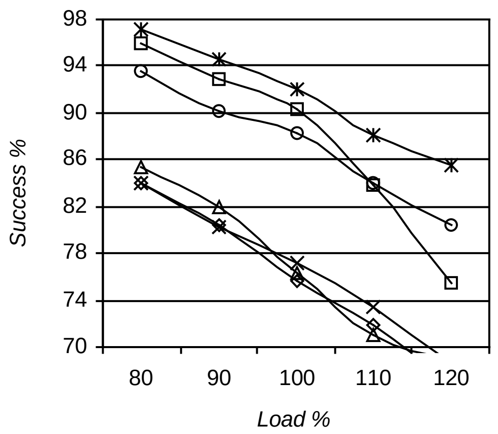

Figure (8.b)

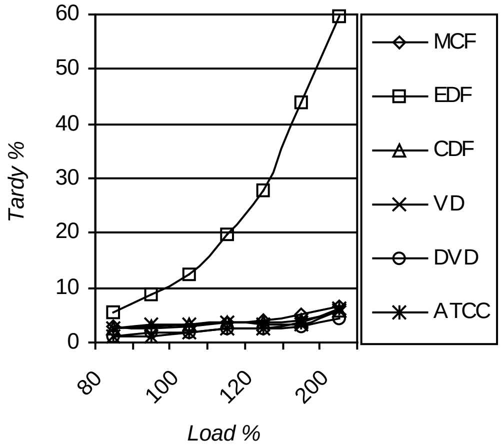

Figure (9)

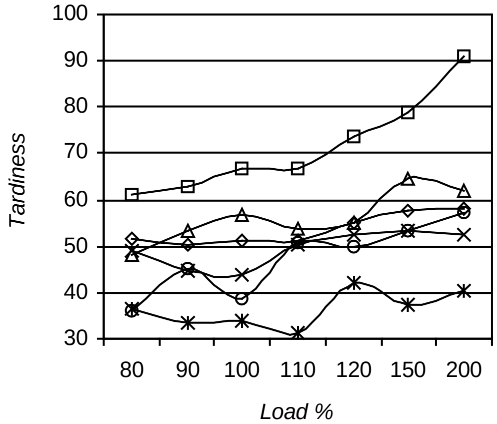

Figure (10)

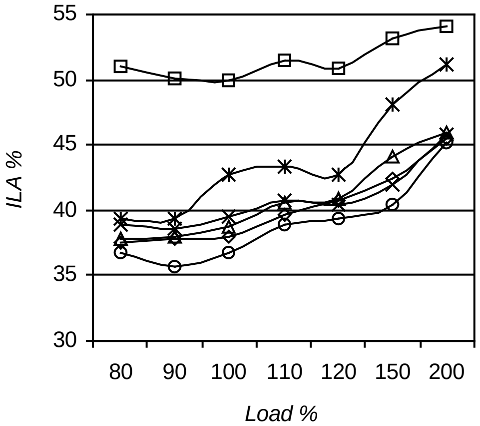

Figure (11)

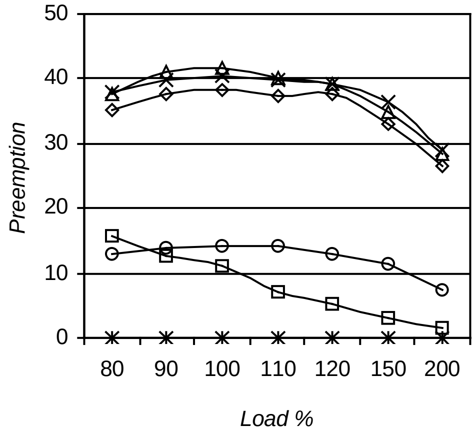

Figure (12)

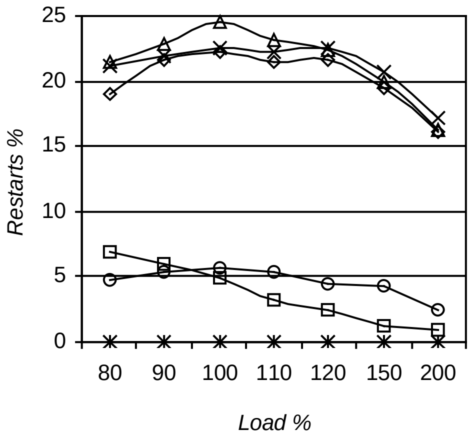

Figure (13)

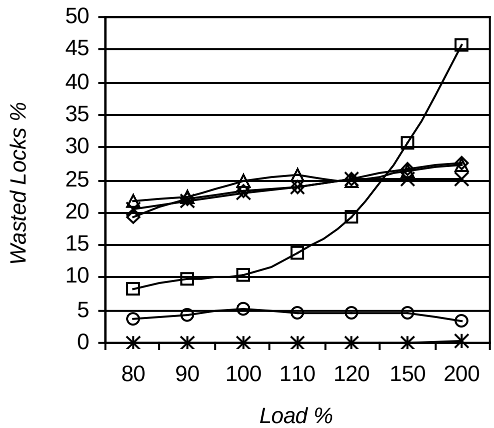

Figure (14)

Finally, figure (14) shows the percentage of locks that is lost (wasted) due to both, restarts as well as aborting partially processed transactions. Notice the correlation between the figures (6 and 14). The percentage of restarts implies the percentage of locks that are lost due to conflict resolution. The two together indicate the database-manger's wasted time. Hence, in contrast to the wasted *CPU* time, figure (14) can be viewed as the wasted database-manger's time. Thus, figures (13 and 14) show that *ATCC* assigns the scheduling priorities in a manner that not only minimizes restarts, but also attempts to prevent wasting any acquired locks and thereby not wasting any of the database-manager's time.

In a hypothetical non-preemptive *CPU* scheduler, once a transaction holds the *CPU*, it does not release it until terminations (abort or commit). It is true that such behavior reduces parallelism; however, what is the point of increasing parallelism while increasing the amount of conflict on database granules, and therefore, having to redo high amounts of accomplished work? However, the question that remains, is how would this hypothetical non-preemptive scheme select the best sequence of transactions to non-preemptively access the *CPU*, and yet, achieve high level(s) of performance? On the other hand, *ATCC* is a preemptive scheme that attempts to derive and assign the scheduling priorities in a manner that actually construct a sequence of transactions that minimizes preemption while achieving a relatively high level(s) of performance as shown in figures 6 through 14. It can be clearly seen in figure (13), the percentage of transactions that are restarted due to conflict resolution is not more than 0%. Thus, many existing studies conducted in scheduling real-time transactions attempt to minimize restarts and wasted locks by devising new concurrency control protocols. However, figures (13 and 14) show that the answer to reducing restarts and wasted-locks does not highly depend on concurrency control protocols, but mainly on the *CPU* scheduler and the manner in which it derives and assigns the scheduling priorities!

Altogether, figures (6 to 14) clearly show that efficient *CPU* scheduling can be achieved with imprecise knowledge of computation time estimates as reflected in the performance of *ATCC*. The scheme; i.e., *ATCC*, is shown to outperform the remaining simulated schemes and is more suitable to operate within an *RTDB* environment due to its high efficiency.

To realize the effectiveness of the underlying timing-model on the *CPU* scheduler, consider figure (15). The figure shows the percentage of transactions that are aborted by the underlying timing-model; i.e., *ITF*, as the load increases. Note that such transactions are the ones whose minimum computation time; i.e., C_{min}, is infeasible within the corresponding transactions' deadlines. Thus, the corresponding transactions are aborted by the timing-model in order to allow the *CPU* scheduler on a higher layer in the system to be redirected towards executing the transactions that have a better chance of meeting their deadlines. Under a load of 200%, up to 17% of the transactions submitted to the system are actually aborted before they even start executing, because the timing-model discovered that these transactions have waited for *too* long in the system and executing such transactions would only waste the system's resources. On the other hand, a value-function would not even be sensitive to such a fact. Rather, a value-function would have probably executed such transactions and end up aborting them midst execution after they have already consumed some amount of the system's resources. Although we do not show it in this study, we found that executing this 17% transactions can waste as much as 5% of the *CPU* time.

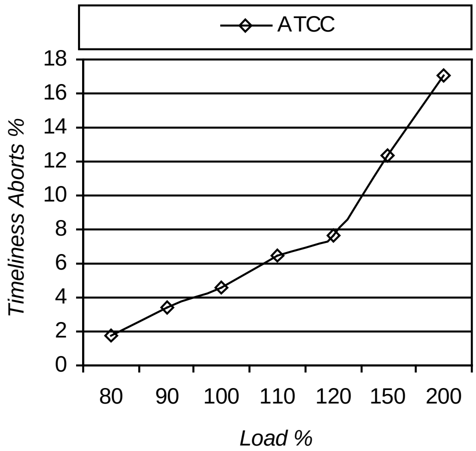

Figure (15)

System's Dependency

The newly introduced scheduling scheme; i.e., *ATCC*, requires an extra amount of overhead due to accounting for the two newly introduced attributes; $\tilde{C}$ and ω. The actual amount of overhead depends on the underlying architecture and operating system. If *special timers* are implemented; i.e., the new proposed extensions for *POSIX* [22], then a separate timer is required for each system-resident transaction, whether the transaction is executing or waiting (where waiting means *ready* or *blocking*). Note that in our simulation, we found that the maximum number of concurrent system-resident transactions = 9 for a load = 80%, and 23 for a load = 200%; thus, a corresponding number of *POSIX* timers are needed.

7. Conclusion

In this study, we investigated the manner in which the *CPU* scheduler in an *RTDB* environment may utilize imprecise knowledge of the transactions' computation-time. Consequently, we constructed a new *CPU* scheduler (*ATCC*) and the necessary underlying timing model (*ITF*), both of which render an *RTDB* system the ability to treat each and every transaction based on its own individual attributes. Consequently, the scheduler becomes capable of accommodating *precise* as well as *imprecise* computation-time estimates. The introduced schemes were shown to substantially reduce the amount of wasted system's resources; and thus, redirect such resources towards sustaining a higher level of performance. Hence, providing an *RTDB* system with a time cognitive behavior complying with the requirements and specifications of time-critical systems.

This study clearly shows that transaction *CPU* scheduling in an *RTDB* system is not hindered by the absence of accurate knowledge of the execution time. Rather, it demonstrates that priority assignment scheme(s) using information about runtime behavior; e.g., *computation-time*, is indeed feasible for a transaction model. In addition, it showed that it is not the concurrency control module, but rather, it is the *CPU* scheduler that is mainly responsible for wasting the system's resources. Hence, with the techniques introduced in this paper, concurrency control is no longer a bottleneck in the performance of *RTDB* systems.

References

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

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

3. S. Aldarmi and A. Burns, "Dynamic Value-Density for Scheduling Real-Time Systems", *Technical Report YCS-310*, Real-Time Research Group, Department of Computer Science, The University of York, U.K., 1998. (To appear in the 11th Euromicro Conference on Real-Time Systems – York, U.K., 1999).

4. A. P. Buchmann, D. R. McCarthy, M. Hsu, and U. Dayal, "Time-Critical Database Scheduling: A Framework For Integrating Real-time Scheduling And Concurrency Control", *Proceedings of IEEE Real-time Systems Symposium*, 1989.

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. M. H. Graham, "Issues in Real-Time Data Management", *The Journal of Real-Time Systems*, 4, pp. 185-202, 1992.

7. J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Experimental Evaluation of Real-time Transaction Processing", *Proceedings of the 10th Real-time Systems Symposium*, pp. 144-153 1989.

8. 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.

9. B. Kao and H. Garcia-Molina, "An Overview of Real-Time Database Systems", *Advances in Real-Time Systems*, S.H. Son (ed.), Prentice-Hall, Englewood Cliffs, NJ, 1995.

10. Y. Kim and S. H. Son, "An Approach Towards Predictable Real-Time Transaction Processing", *Proceedings of the 5th Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, 1993.

11. Y. Kim and S. H. Son, "Supporting Predictability in Real-Time Database Systems", *IEEE Real-Time Technology and Application Symposium (RTAS' 96)*, Boston, MA, June 1996.

12. 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.

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

14. R. Rajkumar, “Synchronization in Real-Time Systems – A Priority Inheritance Approach”, Kluwer Academic Publishers, 1991.

15. K. Ramamritham, “Real-Time Databases”, *Distributed and Parallel Databases*, Vol. 1, No. 2, 1993.

16. K. Ramamritham, “Where Do Time Constraints Come From and Where Do They Go?”, *International Journal of Database Managment*, Vol. 7, No. 2, pp. 4-10, 1996.

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

18. L. Sha, R. Rajkumar, and J. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1175-1185, September 1990.

19. S. H. Son, "Scheduling Real-time Transactions", *Proceedings Euromicro' 90 Workshop on Real-time*, IEEE Computer Society Press, 1990.

20. J. A. Stankovic, “On Real-Time Transactions”, *SIGMOD Record*, Vol. 17, No. 1, March 1988.

21. J. Stankovic, K. Ramamritham, and D. Towsley, “Scheduling In Real-Time Transaction Systems”, *Foundations of Real-Time Computing: Scheduling and Resource Management*, edited by Andre van Tilborg and Gary Koob, Kluwer Academic publishers, pp. 157-184, 1991.

22. 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].