Multiclass Transaction Scheduling and Overload Management in Firm Real-Time Database Systems

Anindya Datta, Sarit Mukherjee, Prabhudev Konana, Igor R. Viguier, Akhilesh Bajaj

May 8, 1997

TR-11

A TimeCenter Technical Report

Title	Multiclass Transaction Scheduling and Overload Management in Firm Real-Time Database Systems
Author(s)	Anindya Datta, Sarit Mukherjee, Prabhudev Konana, Igor R. Viguier, Akhilesh Bajaj
Publication History	March 1996, Information Systems 21(1) A TimeCenter Technical Report

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector)
Michael H. Böhlen
Renato Busatto
Heidi Gregersen
Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector)
Anindya Datta
Sudha Ram

Individual participants
Curtis E. Dyreson, James Cook University, Australia
Kwang W. Nam, Chungbuk National University, Korea
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, ETH Zurich, Switzerland
Vassilis Tsotras, Polytechnic University, New York, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium

The TimeCenter icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors, The Rune alphabet (second phase) has 16 letters. They all have angular shapes and lack horizontal lines because the primary storage medium was wood. However, runes may also be found on jewelry, tools, and weapons. Runes were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Multiclass Transaction Scheduling and Overload Management in Firm Real-Time Database Systems

Anindya Datta, Sarit Mukherjee, Prabhudev Konana, Igor R. Viguier, Akhilesh Bajaj

May 8, 1997

TR-11

A TimeCenter Technical Report

Title	Multiclass Transaction Scheduling and Overload Management in Firm Real-Time Database Systems
Author(s)	Anindya Datta, Sarit Mukherjee, Prabhudev Konana, Igor R. Viguier, Akhilesh Bajaj
Publication History	March 1996, Information Systems 21(1) A TimeCenter Technical Report

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector)
Michael H. Böhlen
Renato Busatto
Heidi Gregersen
Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector)
Anindya Datta
Sudha Ram

Individual participants
Curtis E. Dyreson, James Cook University, Australia
Kwang W. Nam, Chungbuk National University, Korea
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, ETH Zurich, Switzerland
Vassilis Tsotras, Polytechnic University, New York, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium

The TimeCenter icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors, The Rune alphabet (second phase) has 16 letters. They all have angular shapes and lack horizontal lines because the primary storage medium was wood. However, runes may also be found on jewelry, tools, and weapons. Runes were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

Real-Time Database Systems (RTDBSs), have attracted considerable amount of research attention in the recent past and a number of important applications have been identified for such systems, such as telecommunications network management, automated air traffic control, automated financial trading, process control and military command and control systems. Due to the rapidity of change of the system state in such applications, as well as the inherent complexities in controlling such systems (which result in frequent violation of consistency requirements and consequent repeated firings of control actions), it is likely that the transaction load in these systems would be fairly high. Thus RTDBSs need to be equipped with overload management mechanisms. Unfortunately overload management has been a fairly neglected area in real-time systems research in general and real-time database research in particular. In this paper we introduce *Adaptive Access Parameter* (AAP), a scheduling mechanism for multiclass transactions in real-time database systems that employs an *explicit* admission control policy to manage overload as well as system bias towards particular transaction classes. We show the theoretical underpinnings behind AAP and then report a thorough performance study that demonstrates AAP's *substantial* superiority over current algorithms with regard to performance metrics as well as computational overhead.

1 Introduction

1.1 Research Context

The development of command and control systems, such as automated air traffic control, network management, automated stock trading, automated manufacturing etc., has emerged as an important technological challenge in recent times. The common theme of these applications is that they all involve monitoring and processing of environmental data and providing *timely* response. An example of this from a network management domain may be to monitor link utilizations in telecommunications networks and inform the network control center in a timely fashion, i.e., within some specified deadlines, if unacceptably high link utilizations were detected. An example from a process control domain may be to decrease pressure in a chemical reactor within a certain time in response to unacceptably high temperatures. Another important characteristic that defines these applications is their *data intensive* nature, which arises from the immense amount of monitoring data that needs to be processed – e.g., a network management application typically handles several gigabytes of data per day [4]. Aside from their data intensive nature, these applications also tend to be *data driven*, i.e., future decisions may depend upon data collected in the past. For instance, large telephone companies typically store traffic profiles, e.g., South Florida receives a large number of telephone calls on Mother's day. Depending upon this historical information, routing patterns are changed depending on system state (e.g., occurrence of Mother's day). The above mentioned data dependency of these applications mandate the use of database management systems (DBMSs) for these applications. Since, timely processing is of critical importance, such DBMSs are called *Real-Time Database Systems* (RTDBSs) [20].

Because of the time cognizant nature of RTDBSs, transaction processing assumes special importance. In particular, transaction scheduling strategies are a major determinant of the degree of time cognizance of RTDBSs. A factor that significantly complicates scheduling decisions is the presence of *overload*, i.e., more transactions than the system can handle. It is also quite intuitive that in most real-time, data intensive command and control applications, the transaction processing load would, on occasions, be quite high, i.e., overload conditions would prevail. For example, in telephone networks, there are times when system load increases dramatically. To schedule transactions in RTDBSs therefore, not only does one need a time-cognizant scheduling policy, but one also needs

a mechanism that can detect and react to overload situation and keep the system performance from dramatic degradation.

1.2 Current State of Knowledge

In addition to enforcing conventional database management system (DBMS) transaction correctness criteria, RTDBSs attempt to fulfill temporal constraints on transaction execution duration. Most previous RTDBS studies have attempted to address the issue of scheduling transactions by assigning priorities. Moreover, in the context of priority based transaction scheduling in RTDBSs, most performance studies have adopted the Earliest Deadline (ED) principle for priority assignment [15, 1, 8, 11]. The ED policy has been shown to minimize number of missed transactions in lightly to moderately loaded systems (i.e., underloaded systems[1]) [2]. However, under overload conditions, transactions gain high priority under ED policy only when they are close to their deadlines and thus may not have enough time to complete. In fact, as commented upon in [22], executing transactions very close to their deadlines can create a vicious cycle in such a way that transactions relatively farther from their deadlines get delayed, and consequently, may also fail to satisfy their time constraints. This issue has been researched extensively in [18] where the authors show that ED acts in a discriminatory fashion (by discriminating against longer transactions) while trying to maximize transactions that complete on time. The authors in [18] go on to propose an *Adaptive Earliest Virtual Deadline* (AEVD) policy that performs significantly better than ED. AEVD is also shown in [18] to outperform the *Adaptive Earliest Deadline* (AED) policy postulated in [8]. In [8], AED was shown to perform better than the ED policy as well.

While scheduling underloaded systems has been studied extensively (see [22] for a comprehensive review) and several algorithms been suggested (e.g., ED, least-slack), it has been shown that these algorithms perform quite poorly under overload conditions [16, 18]. There has been considerably less work regarding overload management in the context of real-time systems in general and RTDBSs in particular. [16, 21] propose some on line heuristics to handle system overloads in real-time systems. However as far as RTDBSs are concerned, we are not aware of any published work on *explicit* overload management. However, there are some papers that perform *implicit* load control, i.e., they employ strategies that perform actions which have a secondary effect of limiting system load. The most notable of these mechanisms are the AED and the AEVD algorithms mentioned in the previous paragraph. Both of these algorithms assign incoming transactions to either a "hit" or a "miss" group, such that transactions in the miss group only receive processing if the hit group is empty. This has the effect of controlling system load. However, these assignments are somewhat arbitrary, i.e., transactions are assigned randomly to either group without regard to system profiles (e.g., current load) or transaction profiles (e.g., tightness of time constraint).

The results of a comparative performance study of AED and AEVD reported in [18] establish AEVD as the better performer. Thus AEVD may be considered the predominant RTDBS scheduling algorithm in the literature. We feel however, that AEVD suffers from several drawbacks: (a) the hit and miss group assignments is not a true overload management policy for reasons mentioned in the previous paragraph; (b) the authors make the assumption that transaction size is correlated to its time constraint which may not be completely valid in certain circumstances (i.e., long transactions may have short deadlines and short transactions may have long deadlines). Also, as we show in section 5, under overload, the performance of AEVD deteriorates rather sharply. Though this deterioration is slower than the dramatic performance degradation of ED, it is still

[1] We ascribe similar meaning to system loading as in [13]: a system is *underloaded* if there exists a schedule that will meet the deadline of every task and *overloaded* otherwise.

very significant. Consequently, the area of overload management remains very much an open area of research.

1.3 Contributions of This Paper

Given that overload management is important for RTDBSs and not much success has been achieved on controlling overload situations, we felt it important to explore the area. This paper introduces a dynamic admission control and priority based scheduling policy for disk resident RTDBSs, called *Adaptive Access Parameter* (AAP), that considers the arrival times and time constraints of transactions, makes no apriori assumptions about correlations between transaction sizes and time constraints and does *explicit* admission control. The admission control policy of AAP serves dual purposes – *overload management* as well as *bias control*. Bias control refers to reducing discriminatory behavior towards particular transaction classes (like ED is biased against longer transactions). AAP takes advantage of the "canned transaction" assumption in RTDBSs and is shown to have *substantially higher* performance than AEVD under overload conditions, while having *lower bias* and a *significantly lower* time overhead than AEVD. To the best of our knowledge this paper is one of the first to study admission control in RTDBSs in detail.

In summary, AAP has two major contributions to the real-time transaction literature:

1. First of all it introduces a whole new way of priority assignment. Until now, scheduling algorithms have assigned priorities on the basis of deadlines, real (e.g., ED) or virtual (AEVD). In this paper we introduce the notion of not only considering the deadline, but also factoring in the *work remaining*.

2. Secondly, and more importantly, we introduce the notion of explicit *admission control* with the intent of performing both overload management (i.e., shielding the system from the effects of excessive load) as well as bias control (i.e., providing fair service to all transaction classes). Our admission control policy leads to dramatic performance improvements both in terms of reducing transaction misses as well as fairness.

The remainder of the paper is organized as follows: Section 2 introduces the AAP algorithm. Section 3 provides a brief overview of the AEVD model followed by a description of a simulator to study AAP performance in section 4. Finally in section 5 we describe the results of our simulation that compares AAP to AEVD and ED, and discuss the strengths and weaknesses of AAP.

2 Adaptive Access Parameter

The *Adaptive Access Parameter* (AAP) algorithm is designed for multiclass workloads, where classes are differentiated by their mean sizes. In AAP we adopt the basic philosophy that a transaction should be closely monitored as it progresses towards its deadline to prevent the situation where a transaction is close to its deadline and a bulk of its processing remains to be done. In AAP we also make the "canned transaction" assumption of real-time workloads, described below.

The canned transaction principle states that in typical real-time workloads, transactions tend to recur, i.e., there is repetitiveness in transaction patterns [22, 19, 6]. This happens particularly in command and control scenarios [6], which happens to be a classic application of RTDBSs. In such scenarios, where real-time transactions are triggered in order to correct unacceptable system behavior, similar transactions repetitively arrive due to the recurrence of similar problems. In other words, users do not run arbitrary programs, but rather request the system to execute specific functions out of a predefined set, where each function is an instance of a transaction type. For example,

in network management overutilization of links is a common problem. Thus the transactions that respond to this problem arrive frequently to the system. Also, given any link in the network (say link i), these transactions access the same data items (say $i.X$, $i.Y$). Thus, it makes sense to assume, that when the transaction arrives in the system, its data requirements are known. In AAP we assume that a transaction arrives with a *read set* and a *write set*, which denote the data items that the transaction wants to read and write respectively. Validation for this assumption is also offered in [5, 3]. Note however, that a bulk of the ideas presented in this paper *do not* depend on this assumption. As will be seen soon, the canned transaction assumption is used to measure a system parameter called access parameter (AP). There are other ways to measure this parameter, without using the canned transaction assumption (e.g., through system feedback). If such means were used then this paper would be independent of this assumption.

Throughout the remainder of this paper we use D_T, A_T, $C_T = D_T - A_T$, P_T, RS_T and WS_T to denote the deadline, arrival time, time constraint, priority, read-set and write-set of transaction T, respectively. Priority assignments are such that smaller P_T values reflect higher priority. Table 1 summarizes these notations, together with certain additional terminology to be used later.

Notation	**Description**
A_T	Arrival Time of transaction T
D_T	Deadline for Transaction T
C_T	Time Contraint of transaction $T = D_T - A_T$
P_T	Priority of transaction T
RS_T	Read Set of transaction T
WS_T	Write Set of transaction T
$Size_T$	Size of transaction T (number of page accesses from disk)
SR_T	Slack Ratio of transaction T
AP_T	Access parameter of transaction T
$DAPR_T$	Deadline Access Parameter Ratio for transaction T

Table 1: Notations used in this paper

The AAP algorithm introduced in this paper is a two-stage sequential algorithm, i.e., there are two successive stages: (a) admission control; and (b) priority based scheduling. When a transaction arrives, it first goes through the admission control stage, where a decision is made regarding the admissibility of the transaction, based on current system state, as well as the transaction's profile. If the transaction is admitted, it then goes through the priority based scheduling stage. Both these stages use two critical notions called *access parameter* and *deadline access parameter ratio*. To facilitate our description of AAP we adopt the following approach: we first describe the two notions referred to above in detail and outline our basic priority based scheduling strategy (i.e., stage (b)). Subsequently we describe our admission control policy (i.e., stage (a)) in detail.

2.1 Important Parameters and Scheduling Policy in AAP

The basic principle of AAP is as follows: we estimate the size of a transaction using its read and write sets. This is known as the *access parameter* (AP) of the transaction. Based on AP and deadline of the transaction a value called *Deadline Access Parameter Ratio* (DAPR) is computed which indicates how large the time constraint of a transaction is, in relation to its size. Thus, a transaction with a lower DAPR has a tighter time constraint (in relation to its size) than a transaction with a higher DAPR.

Based on this logic, transactions with lower DAPRs are awarded higher priority than transactions with higher DAPRs. During execution time, each time a transaction fetches a page from disk (i.e., has a page fault), its AP is reduced and DAPR recomputed. This adaptive nature of DAPR achieves the goal of monitoring transaction progress and ensuring a time-cognizant progression towards completion. The key notions in AAP are AP and DAPR. The remainder of this section is devoted to a detailed exploration of these notions.

2.1.1 Access Parameter

The access parameter (AP) of a transaction denotes, at a specific instant, an estimate of its unprocessed content. Thus its initial AP is a measure of its size. On the arrival of a transaction into the system, its AP is computed based on its read and write sets and is revised each time the transaction accesses a page from disk. Thus, the key activity is the initial computation of AP. For a transaction T the initial AP computation is done in the following fashion:

$$AP_T = \alpha \times BCE_T + (1 - \alpha) \times WCE_T \tag{1}$$

where

- α is a dynamic control variable, such that $0.0 \leq \alpha \leq 1.0$
- BCE_T is the best-case-estimate of the transaction size in terms of pages it needs to access from disk
- WCE_T is the worst-case-estimate of the transaction size in terms of pages it needs to access from disk

The notions of best and worst case estimates arise because we do not know how the data items are located on pages on disk. Thus based on RS_T and WS_T, it is not possible to predict accurately the number of pages the transaction needs to access from disk. In the worst case, each data item request translates into a disk access, i.e., $WCE_T = |RS_T| + |WS_T|$. The best case estimate is a little more complicated. We compute BCE_T as follows: we consider all the active transactions in the system and the data items they need. Let this set of data items be denoted as *CurrAccessed*. In other words, *CurrAccessed* is the union or the read and write sets of all active transactions in the system. Let $CurrAccessed \cap (RS_T \cup WS_T) = \beta$. Then, *in the best case*, β denotes the set of data items required by T that are already in memory. Thus the most number of page accesses that T would need to make is $BCE_T = |RS_T| + |WS_T| - |\beta|$. Thus truly, BCE_T denotes the "worst given best" case. Also, it is clear from equation 1 that AP_T is a weighted average of the best and worst cases.

The AAP algorithm uses a feedback mechanism to monitor the quality of our estimate and, based on the results of this feedback, adjusts the value of α accordingly (we initialized α to 0.5 at system startup). The feedback is in the form of a linear correlation between our initial size estimate and the true size of a transaction, which we get by tracking how many pages the transaction actually accessed during its execution. This is achieved by performing a linear regression between the AP_T and the true size values. A slope less than 1 indicates underestimation on our part and a slope greater than 1 indicates overestimation of the true size. Thus, if our feedback mechanism indicates underestimation, we need to give more weight to WCE_T. This is achieved by decreasing α by 5%. Conversely, an overestimation means the true size was closer to BCE_T, and α is increased by 5%.

The AAP algorithm recomputes AP each time a transaction does a page access from disk. More specifically, in our implementation, each time T fetches a page from disk, AP_T is reduced by 1.

2.2 Deadline Access Parameter Ratio

The *Deadline Access Parameter Ratio* (DAPR) is how we assign priorities in AAP. DAPR is defined as *the time constraint per unit of unprocessed transaction size* and is a measure of how much work a transaction still needs to perform with respect to its deadline. We compute DAPR of a transaction T as:

$$DAPR_T = \frac{D_T - Clock}{AP_T}$$

The numerator in the expression denotes time left till the deadline expires and the denominator is AP, i.e., the amount of unprocessed work. Therefore, transactions with lower DAPRs need to do more work in less time than transactions with higher DAPRs. Thus priorities are assigned in inverse order of DAPRs, i.e., transactions with lower DAPRs get higher priorities. Each time a resource request is granted, AP is recomputed, resulting in re-evaluation of DAPRs. Correspondingly, the active set of transactions are reprioritized.

As the transaction is granted resources, its AP is decreased to reflect work done. For each page access, we decrease AP_T by 1. One problem that we encountered with this is that in cases of gross underestimation, AP_T becomes 0, which causes a problem in our DAPR calculation, as the AP value appears in the denominator. We handle this case, by setting $AP_{T_{init}}$ (the initial AP estimate) to its worst case estimate i.e., WCE_T and recomputing $AP_{T_{curr}}$ (the current AP estimate), if $AP_{T_{curr}}$ ever becomes 0 for transaction T. Note that with this recomputation, AP can never reach 0 before completion of the transaction.

Along with recomputation of AP, we also check if it is likely that the transaction will successfully complete. This is done as follows: for each page access by a transaction, we check if the condition, given by equation 2 below, is true.

$$(D_T - Clock) \geq \begin{cases} (BCE_T - (AP_{T_{init}} - AP_{T_{curr}})) \times Proc_T & \text{if } BCE_T > AP_{T_{init}} - AP_{T_{curr}} \\ AP_{T_{curr}} \times Proc_T & \text{otherwise} \end{cases} \quad (2)$$

where

- D_T and *Clock* are the deadline of T and the current time respectively;
- $AP_{T_{init}}$ and $AP_{T_{curr}}$ are the initial and current AP values of T respectively;
- BCE_T is the best case size estimate of T, described in section 2.1.1; and
- $Proc_T = ProcCPU + ProcDisk$, where *ProcCPU* and *ProcDisk* are the CPU and disk processing time per data page respectively. These are simulation parameters described below in section 4.1. Thus $Proc_T$ denotes the total processing time per data page.

Basically, this condition ascertains whether a transaction can complete within its deadline, *even if it were the only transaction in the system.* The left hand side of equation 2 is the amount of time left before the deadline of T expires, while the right hand side is the *least* amount of time T needs to finish execution, assuming it executes without interference. If this condition is satisfied, we allow T to continue, otherwise, it is aborted.

2.3 Admission Control in AAP

In this section we introduce the admission control policy of AAP, which serves two primary purposes, *overload management* and *bias control.* The *overload management* component attempts to regulate

transaction entry into the system such that active transactions have a chance of completing by their deadlines. Basically our overload management policy attempts to avoid the following scenario: due to the unrestricted entry of transactions, the system ends up servicing a large number of transactions partially, but is able to finish very few. This scenario, which is symptomatic of the ED policy [18], results in a lot of wasted work. In other words, the overload management philosophy in AAP is to reduce wasted work in the system. The *bias control* component attempts to ensure that AAP treats all transaction classes fairly, i.e., it is not discriminatory towards transactions of any class.

2.3.1 Overload Management

The overload management policy in AAP attempts to optimize resource usage under highly loaded conditions to protect the system from performing "wasted work". The basic mechanism that implements this policy is quite simple: before admitting a transaction, we check if the existing load in the system is such, that the system resources can successfully service this transaction. This is achieved by using transaction DAPRs to compute resource requirement of current workload and comparing that to existing resource availability. The logic and the exact mechanics behind this policy is explained below through an example.

Example: Assume the following RTDBS resource configuration: there are n different types of resources $R_1, R_2, \ldots, R_n$, with m_i identical instances of resource type R_i. In other words, there are m_1 resources of type R_1, m_2 of type R_2 and so on. Let the processing power of an instance of resource type R_i be $Proc_i$ seconds/page. Then, the maximum number of pages that may be processed by all instances of R_i is $\frac{m_i}{Proc_i}$ pages/second. Since the resource types have different service capacities, the maximum number of pages that may be processed by the system, assuming each page processed is routed through an instance of each resource, is:

$$min\left(\frac{m_1}{Proc_1}, \frac{m_2}{Proc_2}, \ldots, \frac{m_n}{Proc_n}\right) \tag{3}$$

The assumption that each page must be routed through an instance of every resource in the system is actually a very restrictive assumption – a transaction may find several data pages in memory and may not have to go to disk. Later on in the section we show how we relax this assumption.

Now let us turn our attention to computing the service requirements of the transactions in the system. Let us assume that at a particular point in time, there exist active transactions $T_1, T_2, \ldots, T_k$ in the system with corresponding DAPRs, $\mathrm{DAPR}_1, \mathrm{DAPR}_2, \ldots, \mathrm{DAPR}_k$ respectively. Now the DAPR of a transaction indicates, as shown earlier, the average rate of progress (seconds/page) a transaction should maintain, in order to finish by its deadline. For example, if T had a DAPR of d, it means on an average, the system needs to process each data page accessed by T in d seconds. This means that the inverse of DAPR, i.e., $\frac{1}{\mathrm{DAPR}}$, indicates number of pages of that transactions that need to be processed per second. Referring back to example in the previous line, it means that the system needs to process $\frac{1}{d}$ pages in one second, on an average, for T to have a chance of completing by its deadline. Therefore, in order to finish all the k transactions existing in the system, the processing speed of the system, in pages/second, is given by:

$$\frac{1}{\mathrm{DAPR}_1} + \frac{1}{\mathrm{DAPR}_2} + \ldots + \frac{1}{\mathrm{DAPR}_k} = \sum_{i=1}^{k} \frac{1}{\mathrm{DAPR}_i} \tag{4}$$

Now, for the system to successfully complete all k transactions, the service requirement of the transactions, given by equation 4, must be less than or equal to the system capacity, given by equa-

tion 3. In other words, a necessary[2] condition for the system to successfully serve all k transactions is given by:

$$\sum_{i=1}^{k} \frac{1}{\text{DAPR}_i} \leq min\left(\frac{m_1}{Proc_1}, \frac{m_2}{Proc_2}, \ldots, \frac{m_n}{Proc_n}\right)$$

Based on the above example, we can now easily identify a necessary condition to test the eligibility of a new transaction for entry into the system. Clearly, if a transaction, say T_{new}, arrives in the system, it can only be allowed in if there is excess capacity in the system to process this transactions, i.e.,

$$\frac{1}{\text{DAPR}_{new}} \leq min\left(\frac{m_1}{Proc_1}, \frac{m_2}{Proc_2}, \ldots, \frac{m_n}{Proc_n}\right) - \sum_{i=1}^{k} \frac{1}{\text{DAPR}_i} \tag{5}$$

As mentioned before, equation 5 makes the assumption that each data page is routed through an instance of each resource. The problem with employing this policy is that it is too restrictive, as a transaction may find several of its data pages already in memory and not have to go to disk. If such a restrictive policy were employed all the time, we run the risk of denying entry to jobs that would actually have been completed. Now we explore how to ameliorate this restrictive effect of the above condition.

Ideally, we would like a *load sensitive* overload management policy, i.e., under "low" overload conditions the policy would be less restrictive than under "high" overload conditions. In AAP we attempt to incorporate *load dependency* in our overload management mechanism by modifying equation 5 to make it sensitive to system load. Note that equation 5 gives the *load independent* version of admissibility criteria for T_{new}. However, we recognize that the expression given by equation 3 is too stringent, because of assumptions made while deriving that expression. We build in load dependence by incorporating a *load sensitivity factor* λ in the above expression as follows:

$$\frac{1}{\text{DAPR}_{new}} \leq \lambda \left[min\left(\frac{m_1}{Proc_1}, \frac{m_2}{Proc_2}, \ldots, \frac{m_n}{Proc_n}\right)\right] - \sum_{i=1}^{k} \frac{1}{\text{DAPR}_i} \tag{6}$$

λ can assume any real value such that $\lambda \geq 1.0$. It can be easily seen that increasing λ has a *less restrictive* effect on our overload management policy, as the right hand side increases, thereby enlarging the admissible set of left hand side values. We start with a λ value of 1.5 and monitor the miss ratio of transactions that are allowed to enter the system. After the entry of every *SampleBatch* transactions in the system, where *SampleBatch* is a system parameter, we re-evaluate miss ratio of transactions that were allowed to enter the system. The goal is to keep this system miss ratio as close to 5% as possible, i.e., of the transactions that are allowed into the system, the objective is to allow no more that 5% to miss their deadlines. Thus after every re-evaluation, if the miss ratio is greater than 5%, λ is decreased, making the admission control policy more restrictive. Conversely, if the miss ratio is lesser than 5%, λ is increased making the admission control policy less restrictive. Also, we would like to state at this point that we assumed that the system has two types of resources: CPUs and disks. Thus equation 6 may be restated for our case as:

$$\frac{1}{\text{DAPR}_{new}} \leq \lambda \left[min\left(\frac{n_{CPU}}{ProcCPU}, \frac{n_{disk}}{ProcDisk}\right)\right] - \sum_{i=1}^{k} \frac{1}{\text{DAPR}_i} \tag{7}$$

[2]But not sufficient

where n_{cpu} = number of CPUs in the system,
n_{disk} = number of disks in the system,
$ProcCPU$ = CPU processing time per data page,
$ProcDisk$ = disk processing time per data page

Equation 7 is the actual overload management condition employed in AAP.

2.3.2 Bias Control

The overload management policy by itself does not always suffice to ensure that an unbiased miss ratio distribution is achieved across all transaction classes, i.e., all classes have fairly similar miss ratios. Also, it has been shown that an RTDBS can produce biased behaviors that do not conform to the requirements of fairness in miss ratio distribution across different classes [18]. To prohibit such undesirable behavior, AAP is equipped with a bias control mechanism that helps transaction classes that would otherwise suffer from high miss ratios relative to other classes. This is achieved by regulating transaction entry into the system. Before we describe our mechanism we would like to introduce two notions to facilitate the discussion – *normalized miss ratio* (NMR) and *class miss ratio* (CMR). As we have mentioned before, transaction classification is based on sizes, i.e., transactions of the same size belong to the same class. The *class miss ratio* of transaction class i, denoted by CMR_i is the fraction of transactions missed with regard to that class, i.e.,

$$\text{CMR}_i\ (\%) = \frac{\#\text{ of missed transactions of class } i}{\text{total } \#\text{ of transactions of class } i \text{ that arrived into the ystem}} \times 100$$

Normalized Miss Ratio (NMR) is one of the two primary performance metrics used in this paper and is explained in detail in section 5. Suffice it to say here that NMR is the overall miss ratio in the system. The goal of a fair scheduling algorithm is to ensure that the miss ratios of all classes are around the NMR value for the system, i.e., $\forall i, \text{CMR}_i \approx \text{NMR}$. Thus, if $\text{CMR}_i >> \text{NMR}$ and $\text{CMR}_j << \text{NMR}$, this means that the system dynamics are such that transactions of class j are better able to complete than transactions of class i. Our bias control policy utilizes this information and admits more transactions of class j into the system and less of class i to take advantage of the current system dynamics. However, we *do not* unilaterally reject all transactions of class i until CMR_i comes down to an acceptable level as this goes against the basic philosophy of trying to complete as many transactions as possible. The bias control policy will only deny entry to a transaction of class i if it is not likely to complete. The bias control algorithm may be stated as follows:

```
on arrival of transaction T belonging to class i
   if CMR_i > NMR
      if T likely to complete
         admit T;
      else reject T;
   else admit T;
```

We now turn our attention to how we ascertain whether a transaction is likely to complete. This analysis, as our overload analysis explained in section 2.3.1, uses DAPR values and is partly based on a powerful statistical technique known as *discriminant analysis* [12]. Discriminant analysis is a *classification* mechanism, i.e., given past history, it can classify an observation as belonging to

a particular class. For instance, salmon come from both Alaska and Canada but have different characteristics. Given the characteristics of a particular salmon, discriminant analysis is used in classifying that fish as Alaskan or Canadian. Our situation (though definitely less fishy) is analogous. We maintain the characteristics of the transactions that have failed and succeeded in the past. Based on this history, we want to classify an incoming transaction as likely to fail, or likely to succeed.

The classification parameter used is the initial DAPR computed for the incoming transactions. We maintain a history of the initial DAPR values of previously failed and succeeded transactions. Although we could have considered the entire past history, our view is that this is not desirable, as workload characteristics change in real-time databases. More specifically, we feel that immediate past history is a truer indicator of current workload characteristics than older history. Based on the DAPR values of the last *SampleBatch* transactions, we compute a *threshold* DAPR, $\text{DAPR}_{threshold}$ as follows:

$$\text{DAPR}_{threshold} = \frac{f \times \mu_{DAPR_f} + s \times \mu_{DAPR_s}}{s + f}$$

where f = number of failed transactions in last *SampleBatch* transactions; and
s = number of successful transactions in last *SampleBatch* transactions; and
μ_{DAPR_f} = mean of the initial DAPRs of the f failed transactions; and
μ_{DAPR_s} = mean of the initial DAPRs of the s successful transactions;

If the DAPR of the new transaction, $\text{DAPR}_{new} > \text{DAPR}_{threshold}$, we classify the transaction as likely to complete, otherwise it is classified as likely to fail. It is clear from the $\text{DAPR}_{threshold}$ expression that it is nothing but a weighted average of the average DAPRs of failed and successful transactions.

3 Adaptive Earliest Virtual Deadline (AEVD)

3.1 Basics

As mentioned in section 1.2, AEVD is the predominant RTDBS scheduling algorithm. Thus it forms the primary basis of performance comparison with AAP, reported in subsequent sections. For completeness, this section summarizes the AEVD mechanism presented in [18]. In AEVD, all transactions that are currently active are divided into a "hit" group and a "miss" group. The capacity of the "hit" group is *HITCapacity*. Upon arrival, each transaction is randomly assigned a unique key I_T and is then inserted into a *key-ordered* list of transactions where its position, POS_T, is noted. If $POS_T < HITCapacity$, the new transaction is assigned to the "hit" group; otherwise it is assigned to the "miss" group. There is an algorithm parameter *HITbatch*. After *HITbatch* transactions leave the "hit" group, the miss ratio is noted and *HITCapacity* is dynamically adjusted such that the miss ratio of the "hit" group is less than 5%. Inside the "hit" group an *earliest virtual deadline* (EVD) policy (described below) is used to assign priorities while inside the "miss" group a random priority policy is used. All transactions in the "hit" group have higher priority than any transaction in the "miss" group.

The AEVD policy uses a sequence of virtual deadlines to control the pace at which a transaction progresses towards its deadline. Each time a transaction requests one of the system resources, its current virtual deadline is tested, and, if it has expired, a new virtual deadline is assigned in the following fashion:

$$V_T = (D_T - Clock) \times PF_T + Clock \tag{8}$$

where

- $Clock$ is the current clock value;
- V_T is the virtual deadline for transaction T;
- D_T is the actual deadline for transaction T; and
- PF_T is the pace factor such that $0.0 < PF_T < 1.0$

The pace factor notion is key to AEVD and controls the way a transaction proceeds towards its deadline. AEVD assigns a pace factor to each transaction as soon as it arrives at the system and it remains fixed throughout its lifetime. It is dependent upon the initial time constraint (i.e., $D_T - A_T$, where A_T is the transaction arrival time) of the transaction and is computed as

$$PF_T = \alpha + (1 - \alpha) \times \left(\frac{C_{max} - C_T}{C_{max} - C_{min}} \right)^2 \tag{9}$$

where

- α is a control variable;
- C_{max} and C_{min} are the maximum and minimum time constraints respectively, of the last $2 \times HITbatch$ transactions.

3.2 Discussion

The key idea in AEVD is the notion of pace factor (PF_T). The pace factor underscores the adaptive nature of the algorithm by controlling the pace at which transactions progress towards completion. This is achieved by monitoring system performance and dynamically adjusting the value of α. However, if one looks carefully at the expression for PF_T given in equation 9, it is clear that the authors assume that a transaction's time constraint is an indicator of its size[3]. This is clear from the subexpression $\frac{C_{max}-C_T}{C_{max}-C_{min}}$, which is nothing but an interpolation of the current transaction's time constraint based on the largest and smallest transactions so far. Below we show with an example how this might lead to problems:

Example: Consider two transactions T_1 and T_2. Let S_{T_1} and S_{T_2} denote the sizes of transactions T_1 and T_2 respectively. Let C_{T_1} and C_{T_2} denote the initial time constraints of transactions T_1 and T_2 respectively. Further assume that $S_{T_1} > S_{T_2}$ and $C_{T_2} > C_{T_1}$. Basically this says that T_1 is a longer transaction than T_2, but with an earlier deadline. Intuitively, we can see that T_1 should advance faster than T_2 as it is longer, and moreover, needs to finish earlier. However, substituting the values in equation 9, it turns out that AEVD would award T_1 a pace factor larger than that of T_2, i.e., $PF_{T_1} > PF_{T_2}$. Substituting these pace factors in equation 8, it is seen that T_1 would be assigned a larger virtual deadline than T_2. This would actually accelerate T_2's progress in relation to T_1's, which should not be the case.

It can be easily seen that the example scenario is not esoteric, and may very well occur in practice. For example, in a RTDBS used for network management, certain problems may need quicker restorative actions (e.g., in case of a link failure reroute traffic) than others (e.g., if utilization on a particular link exceeds 60%, then reduce retransmission rates on transmitting nodes). Clearly, setting traffic parameters is a longer job than setting few node attributes.

[3]Transaction *size* is the number of pages accessed by the transaction from disk

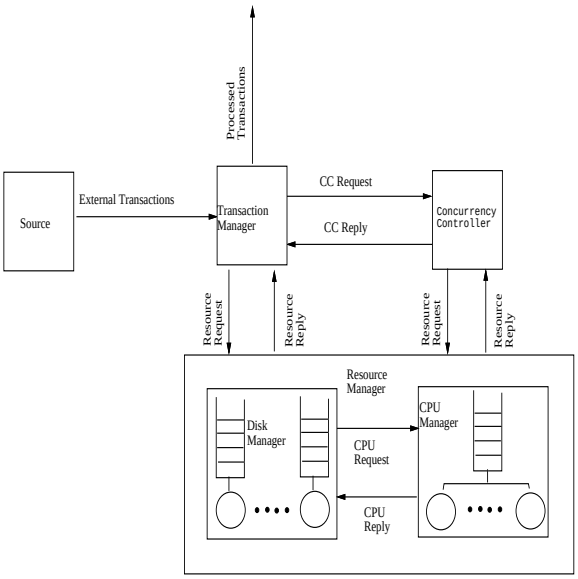

Figure 1: The RTDBS Model

4 Real-Time Database System Model

In this section we present a synopsis of our simulation model to aid the reader in better understanding of the performance analysis results.

The RTDBS consists of a shared-memory multiprocessor that operates on disk-resident data. We model the database as a collection of pages. A transaction is nothing but a sequence of read and write page accesses. A read request submits a data access request to the concurrency control (CC) manager, on the approval of which, a disk I/O is performed to fetch the page into memory followed by CPU usage to process the page. Similar treatment is accorded to write requests with the exception that write I/Os are deferred until commit time.

As far as the CC scheme is concerned, we adopt the *broadcast commit* variant of the *optimistic concurrency control* approach (OPT-BC) [17] for our simulation experiments. This choice was based on previous performance studies that have shown that OPT-BC produces low miss ratios under overload conditions [9]. In OPT-BC, a transaction is allowed to commit when it wants to, but all conflicting transactions are aborted and restarted. All restarted transactions follow the same access patterns as the original transactions.

Our RTDBS model is shown in figure 1. There are four major components:

1. An *arrival generator*, that generates the real time workload with deadlines
2. A *transaction manager* that models transaction execution and implements the AAP algorithm
3. A *concurrency controller*, that implements the CC algorithm, in our case OPT-BC
4. A *resource manager* that models system resources, i.e., CPUs and disks and the associated queues

4.1 Resource Model

Our resource model considers multiple CPUs and disks as the physical resources. For our simulations we assumed the data items are uniformly distributed across all disks. The *NumCPU* and *NumDisk* parameters specify the system composition, i.e., the number of each type of system resource. There is a single queue for the CPUs and the service discipline is assumed to be preemptive-resume based on the transaction priorities. Each individual disk has its own queue with a non-preemptive, transaction priority based service discipline. The parameters *ProcCPU* and *ProcDisk* denote the CPU and disk processing time per page respectively. The total processing time per page is denoted as $Proc_T = ProcCPU + ProcDisk$. These parameters are summarized in table 2.

4.2 Workload Model

Our workload model consists of modeling the characteristics of transactions that arrive and are processed in the system as well as their arrival rate. We consider two broad classes of transaction characteristics: (a.) $Size_T$, which denotes the number of pages accessed by T; and (b.) D_T, the deadline of T. The service demand of T is denoted as $SD_T = Size_T \times Proc_T$ (recall that $Proc_T$ is the time required to process a page). The arrival generator module assigns a deadline to each transaction using the following expression: $D_T = SD_T \times SR_T + A_T$, where D_T, SR_T and A_T denote the deadline, slack ratio and arrival time of transaction T respectively. Thus, the time constraint of transaction T, $C_T = D_T - A_T = SR_T \times SD_T$. In other words, SR_T determines the tightness of the deadline of T. Our workload parameters are summarized in table 2. Table 2 contains some

Parameter Type	**Notation**	**Description**
Resource Parameter	*NumCPU*	Number of CPUs
	NumDisk	Number of disks
	ProcCPU	CPU time /data page
	ProcDisk	Disk time/data page
Workload Parameter	*ArrivalRate*	Transaction Arrival Rate
	DBSize	Number of pages in the database
	WriteProb	Write probability/accessed page
	SizeInterval	Range of the number of pages accessed per transaction
	SRInterval	Range of slack ratio

Table 2: Input Parameters to our RTDBS Model

parameters not discussed so far. The *ArrivalRate* and *DBSize* parameters are self explanatory. The value of the *WriteProb* parameter denotes the probability with which each page that is read will be updated. *SizeInterval* denotes the range within which transaction sizes will uniformly belong. In other words, the arrival generator module generates transaction sizes by drawing from a uniform distribution whose range is specified by *SizeInterval*. Similarly, transaction slacks are generated by drawing from the uniform distribution whose range is specified by *SRInterval*.

5 Performance Analysis

5.1 Performance Metrics

We use the same performance metrics as [18]. There are two primary performance measures that are used to compare AAP and AEVD: *Normalized Miss Ratio* (NMR) and *Bias Factor* (BF). NMR captures the fraction of offered load that is not completed on time, weighted by transaction sizes. It is computed as:

$$NMR = \frac{\sum_{s \in SizeInterval}(s) \times MissRatio_s}{\sum_{s \in SizeInterval}(s)}$$

where $MissRatio_s$ denotes the miss ratio of transactions of size s and *SizeInterval* denotes the range of transaction sizes in the workload. NMR has been shown to be an unbiased performance metric for multiclass systems (see [18] for discussion).

The *Bias Factor*, or BF, is simply the slope of the regression line correlating miss ratio and transaction size. A negative BF denotes a bias in favor of long transactions, while a positive BF indicates favoring shorter transactions (see [18] for discussion).

5.2 Overview of Experiments

We wrote a simulator based on the RTDBS model described in section 4 to comparatively analyze the performance of ED[4], AEVD and AAP. The simulator was written in SIMPACK [7], a C based simulation toolkit. The mean CPU time to process a page was set to 10ms and the mean disk access time per page was assigned a mean value of 20ms. Our simulation considered 8 CPUs and 16 disks. The transaction sizes (for assigning deadlines) was drawn from a uniform distribution with a minimum value of 1 and a maximum value of 30. For AEVD, *HITbatch* was set to 60, which is also the same value used for presenting the results in [18].

5.3 Baseline Model

5.3.1 A Comparative Performance Study

We embark on our performance analysis with a baseline model. Subsequently, further investigations are reported by changing a few parameters at a time. In this model we explore the effects of resource contention on the performance of ED, AAP and AEVD while ignoring data contention. Thus *WriteProb* is set to zero. Results of data contention are reported in section 5.7 where *WriteProb* is set to a non-zero value. Within the context of this section, we also explore the effects of admission control in section 5.3.2. A comparison of the computational overheads of AAP and AEVD is presented in section 5.4. Table 3 shows the parameter settings used for the baseline model. In

Workload Parameter	Value	Resource Parameter	Value
DBSize	1000 pages	*NumCPUs*	8
WriteProb	0	*NumDisks*	16
SizeInterval	[1,30]	*ProcCPU*	10ms
SRInterval	[2.0,6.0]	*ProcDisk*	20ms

Table 3: Parameter Settings for the Baseline Model

[4]We also simulated the performance of ED to add perspective to our experiments. All our baseline model results include ED performance

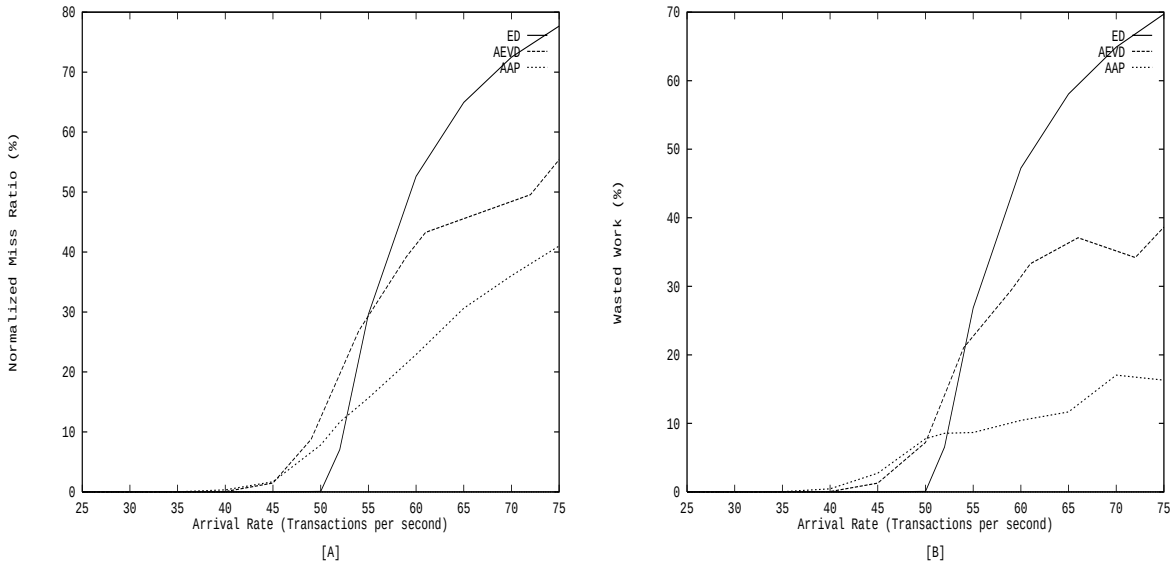

Figure 2: Normalized Miss Ratios and Wasted Work in the Baseline Model

addition to the above parameters, the load sensitivity factor λ is initially set to 1.5 and the re-evaluation frequency for λ, i.e., the parameter *SampleBatch*, is set to 100.

We first highlight the performance of the algorithms under different load conditions. Informally, when the system is able to finish all its jobs, it is underloaded. When it is not able to finish all its jobs it is overloaded. Figure 2A shows the NMRs of AAP, AEVD and ED under different transaction arrival rates.

Load conditions are varied in our simulation by changing the transaction arrival rates. To illustrate an underloaded system, let us take an arrival rate of 50 transactions/second. ED has been shown to be close to optimal in underloaded conditions, and at 50 transactions/per second it has a NMR of 0. Both AEVD and AAP on the other hand perform worse than ED under these conditions, having NMRs of 8% and 12% respectively at an arrival rate of 50 transactions/second. However, it is noteworthy that even under such moderately loaded conditions AAP performs better than AEVD. However, as the system load is increased, there is a remarkable difference in behavior between AAP and the other two algorithms. Specifically the performance of ED and AEVD deteriorate much more rapidly than AAP. For example, for an arrival rate of 67 transactions/second, ED's NMR is 70%, AEVD's NMR is 50% while AAP's NMR is only 32%. This illustrates AAPs clear superiority under overload as far as miss ratios are concerned.

The reason why AAP is so remarkably superior to the other protocols is its explicit overload management policy which makes sure that transactions are only admitted to the system when there is enough resource capacity to process these transactions. ED and AEVD, on the other hand, do not have adequate mechanisms to react under overload. ED has no admission control policy whatsoever, while AEVD attempts to limit entry through assigning transactions to hit and miss groups. However, AEVD's mechanism is ad hoc, e.g., it is not sensitive to system load, resource capacities or incoming transaction profiles. As a result, both ED and AEVD suffer from poor overload management, resulting in significant amount of wasted processing. This is any processing done on transactions that eventually had to miss their deadlines and abort. A comparative study of wasted work is shown in figure 2B. In figure 2B, we plot percentage wasted work at various

transaction arrival rates. Wasted work is defined as the ratio between wasted processing and total processing in the system and is expressed as:

$$\text{wasted work } (\%) = \frac{\text{\# of disk accesses on aborted transactions}}{\text{total \# disk accesses}} \times 100$$

It should be noted that since there is no data contention in the baseline model, aborts happen only because of deadline misses. Thus the numerator of the above expression is a true indicator of the wasted work done in the system. The curves in figure 2B, as expected, exhibit the same general trend as the curves in figure 2A. What is worth noting however, is that at high overload conditions, while ED and AEVD expend a large fraction of their processing in working on jobs that are destined to fail, AAP expends a vast majority of its processing in doing useful work. For example, at an arrival rate of 70 transactions/second, ED wastes about 70% of its total processing, AEVD about 40%, while AAP wastes a remarkably low 16% of its processing doing unuseful work. This translates into the low NMRs of AAP and the relatively higher NMRs of ED and AEVD.

Producing low NMRs is just one indication of a protocol's goodness. For example, if the algorithm under investigation is highly discriminatory towards certain transaction classes, its effectiveness is lost to a large degree. A measure of a protocol's discriminatory behavior is its bias factor (BF). Thus, after looking at NMRs, we turn our attention to BFs of the various algorithms, presented in figure 3A.

Under underloaded conditions, the BFs of all three algorithms are around 0. Under overload conditions, ED's BF assumes a markedly sharp positive trend. For instance at an arrival rate of 52 transactions/second, ED and AEVD have a BF of .5, while AAP has BF around $-.5$. Under greater overloads ED's bias accelerates fast reaching a BF of 3.0 around 60 transactions/second. This denotes ED is highly biased against long transactions, reconfirming ED's discriminatory behavior reported variously elsewhere, e.g., [18]. Under more overloaded conditions, AEVDs BF curve has a trend towards higher positive BF values denoting mild increasing bias against larger transactions. On the other hand AAP's BF exhibits a trend towards lower negative values of BF indicating a mild bias against shorter transactions. What is noteworthy however, is that the *magnitude of the BF values are lower in AAP than in AEVD.* For example, at 65 transactions/second AEVD has a BF of 1.2, which is nearly twice the magnitude of the BF of AAP at $-.66$.

To explore the very different behavior of the algorithms at high load conditions, we examined the miss ratios of various transaction classes at overload conditions. Figures 3B and 3C plot miss ratios as a function of transaction size under overload conditions by setting the transaction arrival rate to 60 and 75 transactions/second respectively.

Figures 3B and 3C show that under overload conditions, ED discriminates heavily against longer transactions. In fact, in our simulations, on an average, ED missed almost 18 times as many transactions of size 30 than it did for size 5. This accounts for its high positive BF. Also, the high degree of misses (more than 55% for transaction sizes 20 and higher in figure 3A) accounts for its high overall NMR under overload. AEVD shows a low positive bias, explained by the comparatively milder (as compared to ED) upward trend of the AEVD curve in figures 3A and 3B. AAP shows a corresponding downward trend. However, on close observation of the AAP and AEVD curves, certain important differences emerge. Firstly, in both figures the AAP curve has a smoother shape than the AEVD curve, which shows a large number of "kinks". This is an indication that the degree of difference in handling different transaction classes is higher in AEVD than in AAP. Secondly, the AAP curves are flatter than the AEVD curves, explaining the lower magnitude of bias.

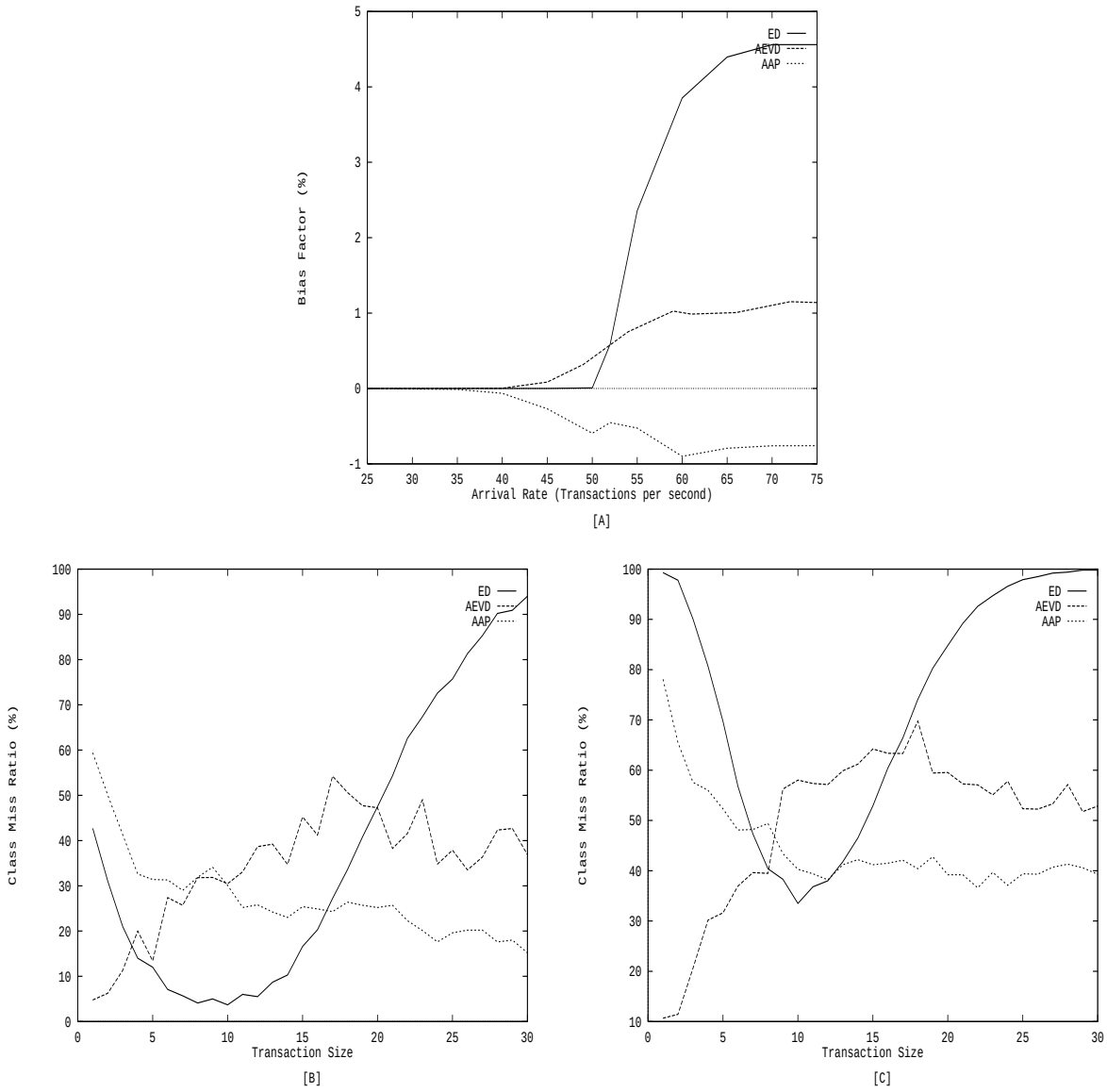

Figure 3: Bias Factors and Class Miss Ratios in the Baseline Model

5.3.2 Effect of Admission Control

One aspect of AAP that sets it apart from other scheduling algorithms is an admission control policy geared for explicit overload management and bias control. Therefore, we think it is important to show the effect of our admission control policy on AAP. A very natural way of studying this is to compare the performance of AAP with a version of AAP that has had the admission control component stripped, i.e., just the basic scheduling mechanism. We call this second, stripped down version of AAP, AAP$^-$. The $-$ signifies its stripped down status.

First we turn our attention to the overload management part and examine how AAP and AAP$^-$ react to high loads. This is shown in figure 4A, which plots the NMRs of AAP and AAP$^-$.

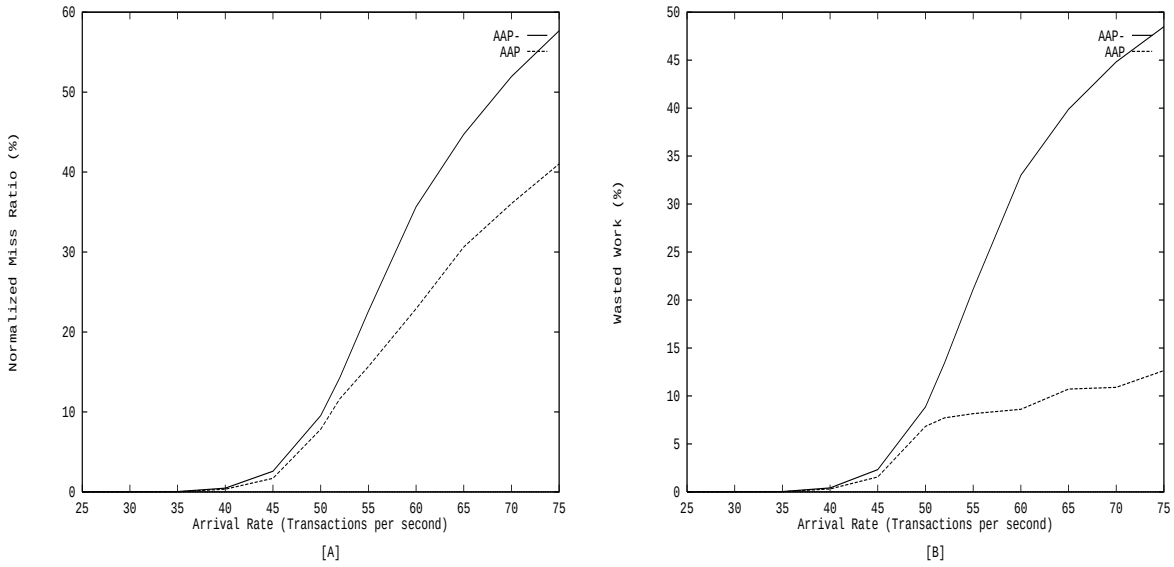

Figure 4: Effect of Admission Control on Normalized Miss Ratio and Wasted Work

As expected, AAP and AAP$^-$ behave virtually identically until true overload conditions kick in around an arrival rate of 50 transactions/second. This is so, as the admission control policy does not really work until overload is detected. Under overload conditions however, AAP performs significantly better, showing an NMR of 41% at 75 transactions/second against AAP$^-$'s 56%. This phenomenon, as explained earlier, occurs due to AAP's ability to regulate wasted work in the system by virtue of its admission control policy. To properly understand this, consider figure 4B, that plots the percentage wasted work in AAP and AAP$^-$ at various system loads. Again, from this figure it is apparent that both algorithms perform identically until overload conditions set in the system. Following this point, AAP$^-$ shows a steep curve signifying wasted work increases dramatically with system load. In AAP however, the curve is fairly flat, rising from about 8% at 50 transactions/second to 13% at 75 transactions/second. Figure 4B also validates the load sensitivity of AAP. Recall in section 2.3 we argued that any true overload management policy should be *load sensitive*, i.e., it should shield the system from the effects of changes in environment load. In AAP we attempted to do this by making the admission control criteria dependent on the system load, i.e., as load increases the stringency of the admission criteria increases. Figure 4B clearly shows that we have been quite successful by virtue of the fact that the wasted work curve remains virtually flat. This shows that the system resources were protected well from the effects of overload.

Now we revert our attention to the bias control component of our admission policy. We explore the effectiveness of our bias control mechanism in a manner similar to the one used to test the overload management component, i.e., through AAP and AAP$^-$. Consider figure 5, which plots bias factors of AAP and AAP$^-$ at various system loads. It may be easily observed from this figure that the bias control policy lowers the BF. Overall it may be said that the admission control policy has a marked effect on AAP with respect to both overload management as well as bias control.

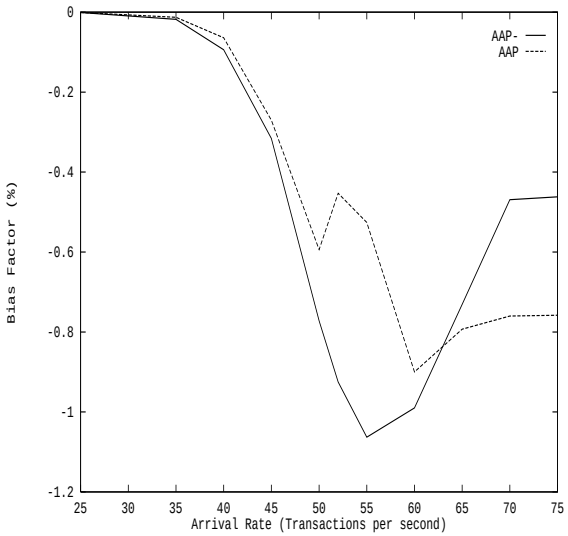

Figure 5: Effect of Admission Control on Bias Factors

5.3.3 Estimation of AP

One of the most critical components of the AAP algorithm is the estimation of AP. Its importance is rooted in the notion that our priority assignment parameter, DAPR, relies on AP to prioritize the transactions. Clearly, if our initial AP estimation is good, AAP would be expected to perform well and not otherwise. For this reason, we felt it was important to analyze how well our initial AP computation corresponded to the true transaction size value[5].

We evaluated this correspondence by an analysis of the errors that resulted as a consequence of our initial AP estimation, i.e., we noted the estimation error (ε) values for a particular simulation run, where ε for a transaction T is given as $\varepsilon_T = Size_T - AP_{T_{init}}$, where $Size_T$ denotes the size, or actual number of page accesses from disk done by T and $AP_{T_{init}}$ is the initial AP value of T. Figures 6A and 6B shows some of our results.

We start out by a simple plot of percentage error values against time, shown in figure 6A. Each time unit on the X-axis represents approximately 1000 simulated time units, and the data reflects 43000 error values, i.e., we analyze the prediction for 43000 transactions. Even though a few of our estimates are way off (e.g., 2500% error), the dark band around the 0% line reflects the fact that an overwhelming majority of our predictions had low error values. Figure 6A also reflects the fact that we overestimate more than we underestimate. This is demonstrated by the complete lack of large negative percentage values. This is also desirable, as gross underestimation would result in transactions being awarded lower priorities than they should, thereby increasing the likelihood of their premature demise. On the other hand overestimation translates into a conservative strategy. Figure 6A, also says something about α, the adaptive parameter that is used as a weight in the AP calculation. It can be seen, that whenever we have a large error value (except in the case when time = 1000), our model quickly adapts to reduce the error, shown by the cluster of points on the zero error line.

We now turn our attention to more rigorous analysis of our error results. Our goal is to provide

[5]Recall from section 1, that the initial AP value is the true transaction size prediction

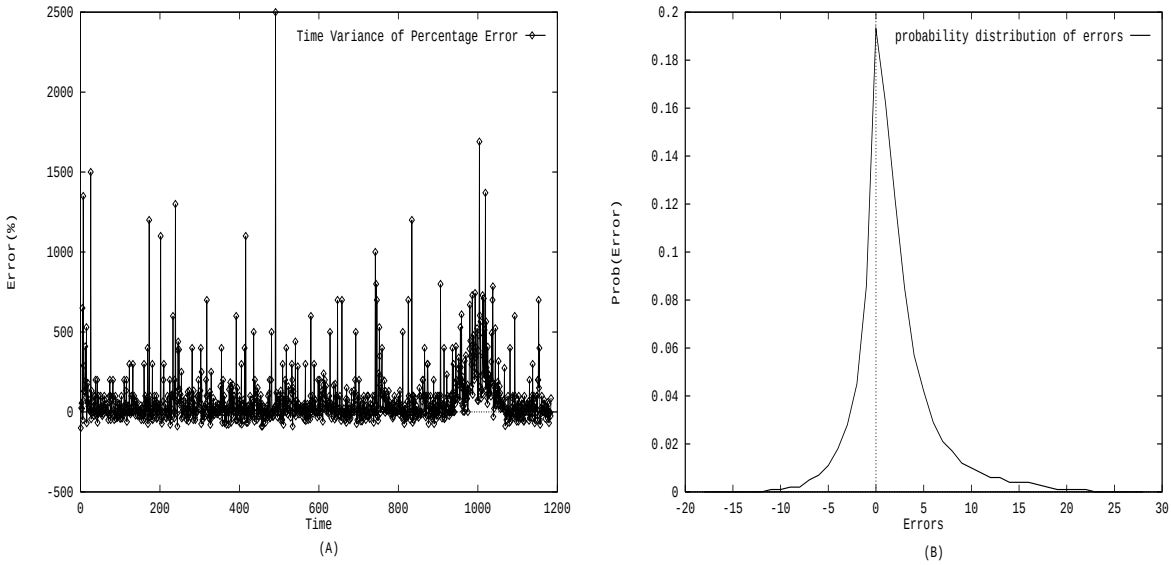

Figure 6: Goodness of AP Estimation

certain statistical bounds of estimation errors. First, we obtained certain summary statistics regarding estimation error of which we report the mean ($\hat{\mu}$), standard deviation (s) and the median (M) respectively: $\hat{\mu}_\varepsilon = 1.83$, $s_\varepsilon = 4.2$ and $M_\varepsilon = 0$. Since our transaction sizes were drawn from a uniform distribution with range [1,30], the error range is [-30,30]. Thus, $\hat{\mu} = 1.83$, indicates, on average, we were very close to the true value. Also, $M = 0$, indicates we overestimated as many times as we underestimated, confirming the lack of any inherent bias in our AP estimation mechanism. Recall, from the discussion in the previous paragraph, that our *magnitude* of overestimation was greater than the magnitude of underestimation.

Next we consider figure 6B, which is the most convincing proof of validity of our estimation mechanism. We introduce this discussion with the following statement: if our estimation is good and unbiased, it would make sense to expect the estimation errors to be distributed symmetrically with a mean of zero. In order to test the above, we extracted the probability distribution of the errors. This is presented in figure 6B. This plot almost perfectly fits a normal curve[6] with the mean very close to zero (more specifically, as reported before, $\hat{\mu} = 1.83$). This analysis also allows us to extract confidence intervals on the error value. At a 90% confidence level, we can assert that the true mean μ is going to lie in the interval $\hat{\mu} \pm 1.645 \times \frac{s}{\sqrt{n}}$. In our case, $\hat{\mu} = 1.83$, $s = 4.2$ and the sample size $n = 43000$. Thus the 90% confidence interval on μ_ε turns out to be [1.79,1.86], which promises to yield very low error values.

5.4 Computational Overhead

In this section we discuss the time and space overheads for AAP and AEVD.

[6] Although our error distribution is slightly skewed to the right owing to the larger magnitude of overestimation

5.4.1 Time Overhead

This is the computation time devoted to activities that are not directly related to scheduling[7], but are done by the algorithms to compute and update control variables, values and miscellaneous housekeeping activities (such as updating data structures). Examples include computing the pace-factor and virtual deadline in AEVD, and the DAPR and AP in AAP. Instead of simply enumerating the overhead tasks and attempting an analytical examination of these tasks, we undertook a thorough investigation of time overhead by *actually* measuring what AAP and AEVD does in performing these overhead activities. Specifically, we did the following: (a) We first enumerated the exact overhead computational activities; (b) We decomposed these activities into the basic arithmetic operations of addition, subtraction, multiplication and division; and finally (c) We measured precisely how many of these basic operations were performed by AAP and AEVD by augmenting our simulation program with appropriate counters. The result was that we obtained *precise* measurements of time overhead. Below we describe in detail the three specific activities enumerated above.

Overhead Activities: A careful examination of the two algorithms reveals that the overhead activities may be classified into three broad categories: (a) Dealing with a new transaction, i.e., setting up a new arrival for subsequent treatment by the scheduling algorithm; (b) progressing a transaction through its lifetime in the system; and (c) housekeeping activities, i.e., keeping track of and recomputing control variables and parameters needed for the above two categories. Below we enumerate, in tabular form, the specific computations inside these broad classes in table 4.

Subsequent to the above analysis, we decided to express time overhead in terms of the the number of basic arithmetic operations performed in completing the aforementioned overhead activities. To make things a little more manageable, instead of counting the four basic arithmetic operators, we classified operators into two basic units: (a) *additive operators*, i.e., addition and subtraction; and (b) *multiplicative operators*, i.e., multiplication and division. Then we obtained four different operation classes by combining these two operator classes with data types: integer additions, integer multiplications, float additions, float multiplications. Then we proceeded to map the numerical manipulations involved in AAP and AEVD into combinations of these four operation classes. These proved to be quite simple as a large majority of the actual computations were simple arithmetic operations. We just made the following two explicit mappings: (a) we counted a comparison operator as an addition (integer or float depending upon the operands); (b) we counted random number generation (in AEVD) as an integer multiplication, assuming a *multiplicative, linear congruential generator* [14].

Before presenting our results, we now briefly turn our attention to a intuitive comparison of the overhead activities in AAP and AEVD outlined above. Note that whenever possible, we assume the computationally best implementation for AEVD.

- In the *new transaction* phase, AAP's work involves computing AP and DAPR which are simple algebraic expressions and are thus performed in constant time. Validating admission control criteria also involves the computation of certain values and a few comparisons which are also performed in constant time. Likewise, for AEVD, computing the pace factor is simply performing certain arithmetic computations and is thus performed in small constant time. In addition to computing the pace factor, AEVD also has to assign transactions to the "hit" or "miss" group. This involves three steps: (a) generating a unique id, (b) insertion

[7] Such as maintaining priority queues at system resources

	AEVD	**AAP**
New Transactions	1. Assigning a new transaction to the "hit" or the "miss" group. This involves assigning a unique key value to the transaction, inserting it in a key ordered list of currently active transactions and comparing its position in that list to parameter *HITcapacity*. 2. Computing the pace factor for new transactions	1. Computing initial AP and DAPR 2. Validating admission control criteria
Transaction Progress	Repeated computation of *virtual deadlines* for each transaction as it proceeds towards its deadline.	Decrementing AP by 1 for each disk access and recomputing DAPR for each resource access.
Housekeeping	1. Keeping track of the greatest and least time constraints ($C_{T_{max}}$ and $C_{T_{min}}$) of the last $2 \times$ *HITbatch* transactions. 2. Adjusting adaptive control variable α, which needs recomputation of the bias factor every *HITbatch* transactions. 3. Recomputing *HITcapacity* after every *HITbatch* transactions.	1. Adjusting adaptive control variable α every *SampleBatch* transactions that arrive. 2. Adjusting adaptive control variable λ every *SampleBatch* transactions that are actually *allowed* into the system.

Table 4: Description of Overhead Computational Activities

in a ordered list of ids, and (c) a comparison with the *HITCapacity* parameter to perform the final assignment. Operation (c) is a small constant time operation. Operation (a), by choosing a random number generator with a long period may also be performed in constant time. Operation (b) however requires $\log n$ time. At high arrival rates n could get large. However, assuming the best possible implementation, it may be said that for this phase the time overheads for AAP and AEVD are fairly identical[8].

- The admission control work performed in the *new transaction* phase by AAP, more than pays off in the second phase (i.e., the *transaction progress* phase). From the descriptions above it may be easily seen that the work in re-computing the virtual deadline (presented in section 3) and the work in recomputing AP and DAPR (presented in section 2) are very comparable in computational intensity. In other words, on a per transaction basis, AAP and AEVD would perform about the same amount of work. However, as shown in section 5.3.2, AEVD has a *much* greater magnitude of wasted work than AAP. This means that AEVD performs computation on a lot of transactions that will eventually fail to make their deadlines. This leads to *vastly reduced total* overhead in this phase in AAP than in AEVD (more on this below).

[8]Note that we give AEVD substantial advantage here

- In the *housekeeping* phase, the recomputation of α in both AAP as well as AEVD is very comparable in computational intensity as well as re-evaluation frequency, as they both need to perform a linear regression. Similarly, recomputation of λ and *HITcapacity* are nearly identically computationally intensive. However, the computation of the maximum and minimum time constraints of the last $2 \times$ *HITbatch* transactions is a *very* computationally intensive operation. This is so as we have to keep track of the time constraints of the last $2 \times$ *HITbatch* transactions, and perform frequent re-evaluations of the max and the min values. Thus even in the housekeeping phase we expect AEVD to have substantially higher overhead than AAP.

Putting the above three points together we conclude that on the whole AEVD should have *significantly* higher overhead than AAP. Note that the above analysis attributes the best possible implementations to AEVD. In fact, AEVD's relative high overhead *does not* arise due to implementation reasons at all, but rather due to AAP's efficiency in general and its ability to reduce wasted processing in particular. For example, in the transaction progress phase, both AAP and AEVD have comparable (constant time) overhead on a per transaction basis. However, AAP's efficient overload management drastically reduces the number of transactions that start but eventually miss deadlines (see figure 4B for a comparative estimate of wasted work). As a result, AEVD wastes a lot more computation in this phase than AAP, leading to a substantial cost advantage for the latter.

We now present the results of our overhead study. We counted the number of times the four operation classes outlined above were performed in the context of overhead activities. In addition, for comparative results, the operation classes were then benchmarked as follows:

1. Each integer addition was assigned a weight of 1,
2. each integer multiplication was assigned a weight of 10,
3. each floating-point addition was assigned a weight of 3, and
4. each floating-point multiplication was assigned a weight of 20.

The results are presented in tables 5 and 6.

Arrival Rate	**Integer Additions**	**Integer Multiplications**	**Floating-point Additions**	**Floating-point Multiplications**	**Total Overhead**
25	2032	4	113	84	4091
31	2035	4	115	84	4100
35	2035	4	116	83	4083
41	2034	4	117	85	4125
46	2041	4	119	86	4158
50	2147	4.2	150	86	4359
55	2510	5.2	188	96	5046
59	2526	5.4	191	97	5093
64	2837	7	234	116	5929
70	3540	9	301	146	7453
75	4194	10	361	165	8827

Table 5: Overhead per Successfully Completed Transaction - AEVD

Finally figure 7 presents the overall overhead comparison of AEVD and AAP with the basic operations being weighted as shown above.

Arrival Rate	Integer Additions	Integer Multiplications	Floating-point Additions	Floating-point Multiplications	Total Overhead
25	17	0.2	29	8	271
35	18	0.2	28	8	266
40	19	0.2	28	8	264
45	22	0.2	28	8	268
50	27	0.2	29	8	281
52	30	0.2	29	8	287
55	30	0.3	29	9	289
60	37	0.3	31	9	312
65	40	0.3	31	9	321
70	60	0.4	32	8	345
75	128	0.4	28	8	384

Table 6: Overhead per Successfully Completed Transaction - AAP

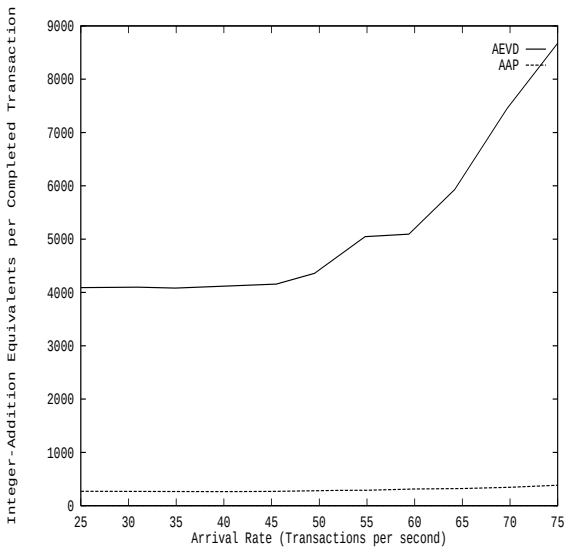

Figure 7: Computational Overhead

5.5 Space Overhead

The space overhead is the amount of memory consumed by the data structures essential to the algorithms. Examples include the transaction structures in the algorithms (including information each transaction must carry with it, such as it's virtual deadline and pace actor in AEVD, the DAPR and AP in AAP, etc). In AEVD, these data structures include the following:

1. A transaction structure consisting of fields: virtual deadline, pace factor, and member hit/miss group.

2. An integer array of the last $2 \times HITBATCH$ transactions' time constraints.

The AAP algorithm, on the other hand, required only the following data structures:

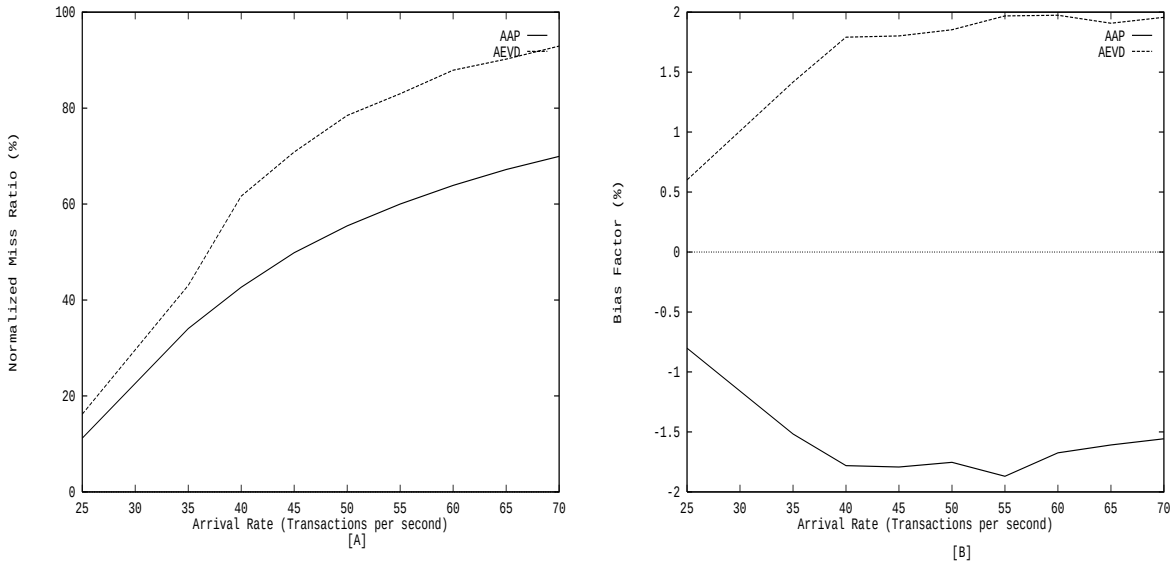

Figure 8: NMR and BF under increased resource contention

1. A field containing the current and initial values of the AP parameter in each transaction,
2. A set of counters used in computing the adaptive factors α and λ.

We now turn our attention to exploring the performance of AAP by varying certain parameter values in the baseline model. Specifically, we look at the effects of varying levels of resource and data contention. In the following sections however we do not report ED performance[9] for the sake of brevity, as well as the fact that we conclusively proved in the baseline section that AAP and AEVD substantially outperform ED. Other experiments simply reconfirm the baseline results and thus reporting these results amounts to repeating ourselves.

5.6 Effect of Resource Contention

We now turn our attention to exploring the sensitivity of the algorithms to the level of physical resource contention in the system. In the baseline experiment, we purposely kept resource contention low to study the effects of overloading. Now we report the results of an experiment that seeks to increase resource contention levels by decreasing the numbers of CPUs and disks. Specifically, for the curves shown below, *NumCPU* and *NumDisk* were set to 4 and 8 respectively. The NMR and BF for this experiment are shown below in figures 8A and B respectively. The trends shown in these figures are identical to the ones in the baseline experiments (see figures 2A and 3A) and similar reasoning may be applied to explain them. Basically, under tighter resource contention, both AAP and AEVD miss more transactions. One noteworthy aspect of the NMR curves is that the performance of AEVD, in percentage terms, is worse than AAP with respect to the baseline experiments. In other words, there is a larger separation of the curves. This is because AAP adapts better to resource restriction than does AEVD.

[9] Though we have conducted extensive experiments

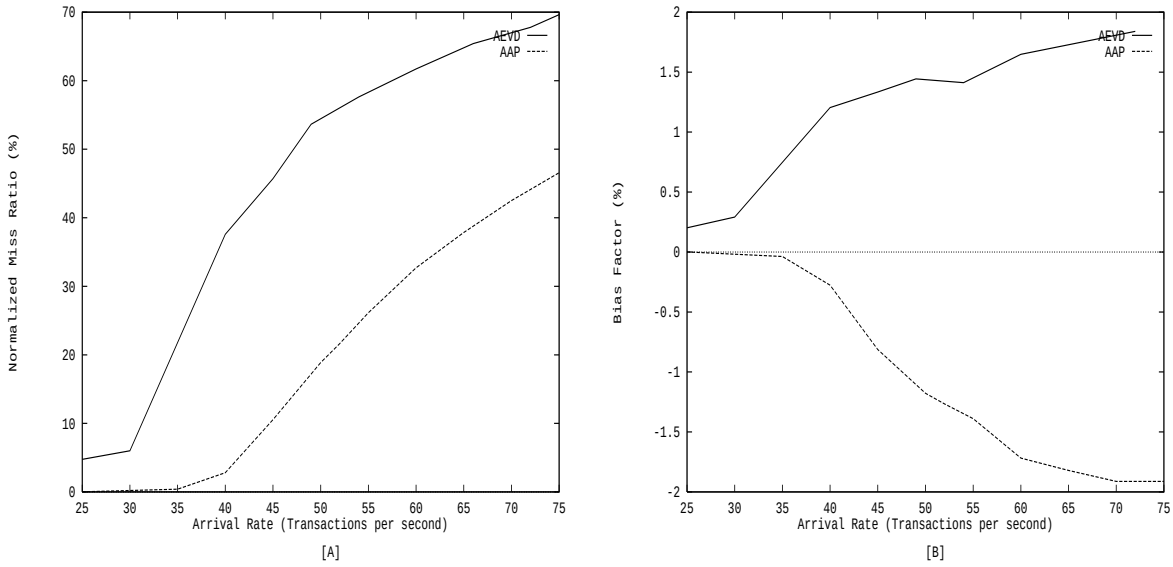

Figure 9: Effect of Data Contention

5.7 Effect of Data Contention

Our next experiment investigates situations where both resource as well data contention causes the system to miss deadlines. Data contention is achieved by setting the value of the system parameter *WriteProb* (write probability) to .25, which signifies that 25% of the data items accessed are going to be written as well. For the concurrency control algorithm we used the *Broadcast Commit* variant of the classical optimistic concurrency protocol (OPT-BC). This choice was made as OPT-BC has been shown to produce low miss ratios under heavy loads [9].

The optimistic protocol allows transactions to execute without interference until they are ready to commit, at which time a validation phase is done. If a transaction fails its validation check, it is restarted. In the broadcast commit variant, a committing transaction notifies other concurrently executing transactions with which it has conflicts, when it is ready to commit. This allows these other transactions to restart right away. For an extended discussion of OPT-BC, see [17].

Our results for this experiment are presented in figures 9A and B. In the presence of data contention, both AAP and AEVD miss more transactions as is easily seen by comparing figure 9A to the baseline results shown in figure 2. However, what is remarkable about these results is what they reveal regarding the behavior of AAP under varying load conditions. To properly understand this, consider figures 10A and B below, that compare the performance of AAP and AEVD with and without data contention. It can be easily seen that figures 10A and B do not contain any new information, but are realignments of the curves from figures 2A and 9A.

Figure 10A reveals that under low to moderate transaction loads there is a clear difference in performance of AAP depending on the presence or absence of data contention. For example, at a transaction load of 45 arrivals/second, in the absence of data contention, AAP shows a NMR of 2%. For the same arrival rate with data contention the NMR is 11%, i.e., more than 5 times as much. With increasing transaction load, this difference gradually decreases. At 60 transactions/second, the NMR is 22% in the baseline model while it is 32% with data contention. At 75 transactions/second, which constitutes high overload, the NMR values are practically similar:

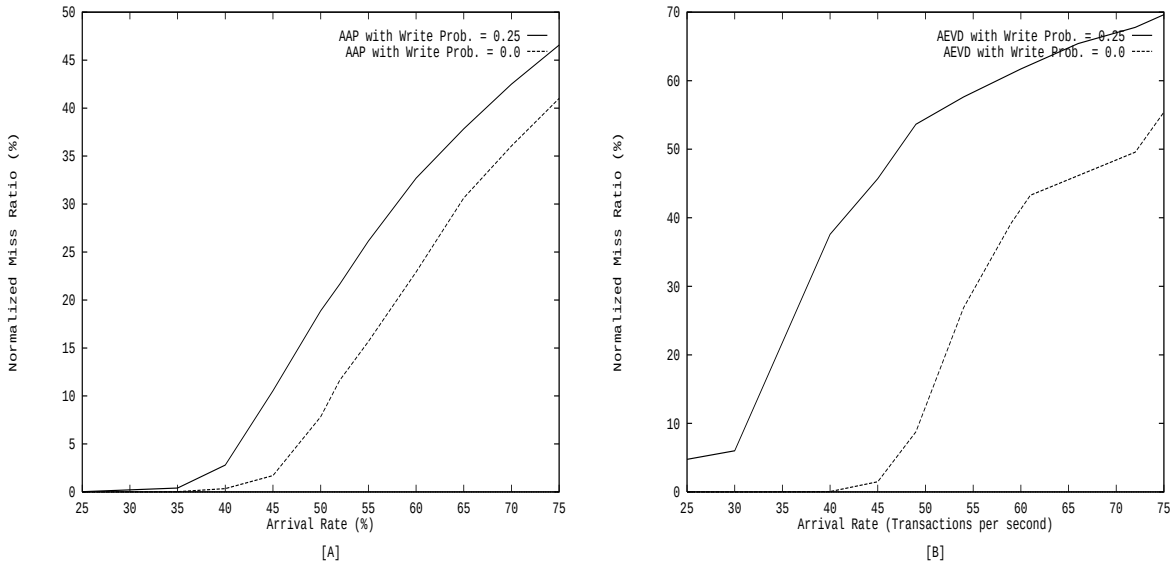

Figure 10: Comparison of Normalized Miss Ratios with and without Data Contention

43% without data contention and 48% with data contention. To summarize the above discussion, it may be said that as transaction load increases, the system performance with data contention almost becomes the same as that without data contention. The explanation of this phenomenon is the following: at low transaction loads, the effects of data contention dominates the effects of resource contention. Consequently, AAP performance degrades substantially from that of the baseline model. However, increasing transaction load has the impact of increasing the effect of resource contention, compared to that of data contention. With increasing loads, therefore, our admission control policy kicks in, which is equipped to handle resource contention. Thus, as resource contention starts to have an impact on the system, AAP performs better and better. Finally at high loads, when resource contention clearly dominates data contention, AAP's admission control mechanism makes sure that there is practically no difference in performance relative to the baseline model. On the other hand, figure 10A reveals the substantial difference in AEVD's behavior in the presence of data contention. For example, at 50 transactions/second AEVD's NMR is a mere 10% without data contention but a significant 50% with data contention. The above description may be easily visualized simple by looking at the separation of the curves in figures 10A and 10B. In 10A, the curves are fairly close, signifying AAP does not lose much of its edge given data contention. In 10B on the other hand, the curves are much farther apart signifying a substantial loss of effectiveness of AEVD in the presence of data contention.

Figure 9B again illustrates AAP's bias against longer transactions by having a negative bias factor. Interestingly, at low arrival rates, when data contention dominates resource contention, AAPs bias towards long transactions is offset somewhat by OPT-BC, which has been shown to favor short transactions [10]. However, under overload conditions, resource contention dominates, accounting for the higher negative bias values. In other words, at high loads AAP's inherent bias towards longer transactions more than compensates for OPT-BC's bias towards short transactions, by giving longer transactions higher priority at CPUs and disks.

6 Conclusions

In this paper we presented *Adaptive Access Parameter* (AAP), a protocol for scheduling transactions in real-time database systems (RTDBSs) under overload conditions. The workload was assumed to be multiclass, where the classes are categorized by their sizes. In addition to meeting time constraints, a second goal of AAP was to exhibit non-discriminatory behavior towards transactions of all classes. Overload management and bias control are two issues that have been subjected to comparatively little examination in the RTDBS transaction scheduling context, as opposed to the intense investigation of different methods of priority assignments for transaction scheduling. Basically, AAP is a two stage algorithm: the first stage constitutes admission control, which does overload management as well as bias control; and the second stage is a prioritized scheduling policy where we propose a new priority assignment policy based on the tightness of a transaction's time constraint.

This paper contributes to the RTDBS literature in two ways. Firstly, it proposes, as far as we know, the first explicit overload management mechanism reported in the literature, with respect to RTDBSs. Our policy is load sensitive, i.e., it attempts to shield the system from the effect of the environment load. Thus as, transaction load waxes and wanes, our admission control policy reacts by becoming more or less restrictive, respectively. AAP also performs explicit bias control, with a goal of inducing non-discriminatory behavior towards different transaction classes. Last but not the least, our admission control policy is based on a simple idea: it is counterproductive to load the system beyond its processing capacity. Our second contribution is in the priority assignment policy of AAP. Unlike static priority assignment heuristics, AAP is dynamic – it attempts to control the rate at which transactions progress towards their deadlines. The fundamental principle of AAP is that transactions are prioritized based on the *amount of unprocessed work per unit time remaining till deadline expiry.* This principle is quantified using the notion of *Deadline Access Parameter Ratio* (DAPR). The essence of AAP is that the algorithm monitors transactions as they progress to their completion, and awards resources based on DAPR values. A feedback mechanism is implemented that ensures DAPR is properly updated to reflect the rapidity of a transaction's progress. AAP does not assume any a priori knowledge of transaction sizes or time constraints (which are assumed to be positively correlated in AEVD), but uses the "canned transaction" assumption.

Finally, a few words regarding our results. The outcome of our simulation experiments unequivocally demonstrate AAP's effectiveness. As a comparative platform we chose *Adaptive Earliest Virtual Deadline* (AEVD), which is recognized as the state of the art as far as overload management is concerned even though AEVD controls system load *implicitly* [22]. We conclusively showed that AAP has *substantially* lower transaction misses than AEVD, while enjoying a lower bias. In other words, AAP misses fewer transactions, while exhibiting lesser discriminatory behavior than AEVD. What makes AAP even more attractive is its *significantly* lower overhead than AEVD, which shows that we obtained our performance improvements at a substantially lower cost.

In the process of our experiments it turned out that AAP does favor long transactions somewhat over short transactions. This is a weakness of our algorithm and we hope to rectify this in an extension to this paper. In the future, we plan on making AAP more efficient by forcing the system to be even more conscious of the load. Another idea to make AAP more efficient is to have it perform better at lightly loaded conditions, when our studies showed that ED performed slightly better. This may be achieved by augmenting AAP such that it behaves like ED at low loads, but switches modes under overload conditions.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of the 15th VLDB*, 1989.

[2] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: Performance Evaluation. *ACM Transactions on Database Systems*, 1992.

[3] P. Bernstein, V. Hadzilakos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[4] Sprint Network Management Center. Site Visit, April 1992.

[5] S. Chakravarthy, D. Hong, and T. Johnson. Real-time transaction scheduling: A framework for synthesizing static and dynamic factors. Technical Report UF-CIS-TR-94-008, Dept. of CIS, University of Florida, 1994.

[6] A. Datta. Research Issues in Databases for Active Rapidly Changing data Systems (ARCS). *ACM SIGMOD RECORD*, 23(3):8–13, September 1994.

[7] P.A. Fishwick. Simpack: Getting started with simulation programming in C and C++. Technical Report TR92-022, Computer and Information Sciences, University Of Florida, Gainesville, Florida, 1992.

[8] J. Haritsa, M. Livny, and M. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1991.

[9] J.R. Haritsa, M.J. Carey, and M. Livny. Dynamic Real-Time Optimistic Concurrency Control. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1990.

[10] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In *Proceedings of the 17th Intl. Conf. on Very Large Data Bases*, 1991.

[11] J. Huang, J. Stankovic, D. Towsley, and K. Ramamrithnam. Experimental Evaluation of Realtime transaction processing. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1989.

[12] R.A. Johnson and D.W. Wichern. *Applied Multivariate Statistical Analysis.* Prentice Hall, 1992.

[13] G. Koren and D. Shasha. D^{over}: An optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 290–299, August 1992.

[14] A.M. Law and C.S. Larmey. *An Introduction to Simulation Using Simscript II.5.* CACI Products Company, 1984.

[15] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, January 1973.

[16] C.D. Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, Computer Science Department, Carnegie-Mellon University, 1986.

[17] D. Menasce and T. Nakanishi. Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management. *Information Systems*, 7(1), 1982.

[18] H. Pang, M. Livny, and M.J. Carey. Transaction scheduling in multiclass real-time database systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.

[19] B. Purimetla, R.M. Sivasankaran, J.A. Stankovic, K. Ramamritham, and D. Towsley. Priority Assignment in Real-Time Active Databases. Technical report, Computer Sciences Department, University of Massachusetts, 1994.

[20] K. Ramamritham. Real-Time Databases. *Distributed and Parallel Databases: An International Journal*, 1(2):199–226, 1993.

[21] Lui Sha, J.P. Lehoczky, and Ragunathan Rajkumar. Solutions for some Practical Problems in prioritized pre-emptive scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1986.

[22] P.S. Yu, K-L. Wu, K-J. Lin, and S.H. Son. On Real-Time Databases: Concurrency Control and Scheduling. *Proceedings of the IEEE, Special Issue on Real-Time Systems*, 82(1):140–157, 1994.

Recommended by Ozgur Ulusoy and Patrick O'Neil, Co-Editors