Optimizing I/O-Intensive Transactions in Highly Interactive Applications

Mohamed A. Sharaf
ECE Department
University of Toronto
Toronto, Ontario, Canada
msharaf@eecg.toronto.edu

Panos K. Chrysanthis
CS Department
University of Pittsburgh
Pittsburgh, PA, U.S.A.
panos@cs.pitt.edu

Alexandros Labrinidis
CS Department
University of Pittsburgh
Pittsburgh, PA, U.S.A.
labrinid@cs.pitt.edu

Cristiana Amza
ECE Department
University of Toronto
Toronto, Ontario, Canada
amza@eecg.toronto.edu

ABSTRACT

The performance provided by an interactive online database system is typically measured in terms of meeting certain pre-specified Service Level Agreements (SLAs), with expected transaction latency being the most commonly used type of SLA. This form of SLA acts as a soft deadline for each transaction, and user satisfaction can be measured in terms of minimizing tardiness, that is, the deviation from SLA. This objective is further complicated for I/O-intensive transactions, where the storage system becomes the performance bottleneck. Moreover, common I/O scheduling policies employed by the Operating System with a goal of improving I/O throughput or average latency may run counter to optimizing per-transaction performance since the Operating System is typically oblivious to the application high-level SLA specifications. In this paper, we propose a new SLA-aware policy for scheduling I/O requests of database transactions. Our proposed policy synergistically combines novel deadline-aware scheduling policies for database transactions with features of Operating System scheduling policies designed for improving I/O throughput. This enables our proposed policy to dynamically adapt to workload and consistently provide the best performance.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Transaction processing*.

General Terms

Algorithms, Design, Performance.

Keywords

Database Systems, Transaction Processing, I/O Scheduling.

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

1. INTRODUCTION

In today's highly interactive database applications, optimizing I/O intensive transactions is a challenge, as important now as it was at the emergence of database management systems. The reason for this is that today's highly interactive applications, such as Web applications driven by a database or a data warehouse, must meet the users' high performance expectations in light of the fact that these applications involve large number of data accesses. For example, in database-driven Web applications, a user during a single session accesses (interactively or at the "speed of thought") web pages which are dynamically created from data in databases. Typically, such a dynamic web page is composed by a number of content fragments, each of which is materialized at every request by running several I/O intensive queries and executing lengthy code to produce HTML. As reported in [17], I/O access can constitute up to 90% of the transaction execution time.

Similarly, in data warehousing applications, users interactively analyze massive amounts of data by means of OLAP (on-line analytical processing) tools. Such tools typically query the underlying data warehouse, retrieve and process large tables of data, and finally provide users with summarized aggregate information.

In such highly interactive applications, user satisfaction or positive experience determines the applications' success (and keeps the competitors "more than a click away" [25]). Service Level Agreements (SLAs) are used to quantify the user's satisfaction. Transaction latency expressed as a *deadline* is the most commonly used form of SLA, reflecting the user's expectation for the transaction to finish within a certain amount of specified time.

In order to maximize users' satisfaction, the underlying data management system should strive to maximally meet the pre-specified SLAs. However, this is a non-trivial task, especially given the bursty and unpredictable behavior of Web access which often leads to conditions of heavy load and high utilization. This goal is further complicated in I/O-bound transactions, where slow disk access becomes the performance bottleneck and traditional caching cannot fully mitigate the problem.

Previous work that attempts to meet the users' pre-specified SLAs focuses on scheduling multiple transaction access to CPU (e.g., [25, 24, 26, 10]) as well as CPU scheduling of sub-transactions to meet a global transaction deadline (e.g., [14, 15, 16, 27]). In these policies, as well as in most transaction scheduling policies, the scheduling of I/O requests is handled by the underlying operating sys-

tem (OS). However, the OS is typically oblivious to the application higher-level performance goals (i.e., SLAs). Even worst, the operating system usually employs scheduling policies which optimize the I/O throughput (e.g., using the *Shortest Seek Time First (SSTF)* policy) which under some workload conditions curtail the meeting of SLAs.

The lack of SLA-awareness at the I/O scheduling level, motivated us to investigate scheduling I/O page requests issued by multiple transactions in order to meet per-transaction deadline. Such a deadline could be a user specified transaction deadline (i.e., SLA) or in the case of sub-transactions, the global transaction SLA could be used to derive an intermediate deadline for each sub-transaction using schemes such as [14, 15, 16, 27].

The same observation above motivated the work in [8] which studied the scheduling of I/O operations in the context of Real-Time Database Management System (RTDBMS) with *hard deadlines*. In particular, [8] proposed a policy for scheduling I/O operations with the goal of maximizing transaction success rate (i.e., the number of transactions meeting their deadlines).

Contrary to RTDBMS where transactions are associated with hard deadlines, in highly interactive database applications, transactions are associated with *soft-deadlines* which express the performance expectations of the end-user and beyond which transactions are not dropped but are still processed to completion. Hence, contrary to RTDBMS, the success rate is not an appropriate performance goal in highly interactive database applications. Thus, in this paper, we argue that minimizing transaction *tardiness* (i.e., amount of deviation from deadline) is a more appropriate performance goal in online interactive systems where users' transactions are not dropped beyond deadlines but are still processed even if they passed those assigned deadlines. Towards this, we propose a parameter-free adaptive scheme, called *TraQIOS*, for scheduling I/O requests issued by transactions to disk-resident data. The target goal for TraQIOS is to automatically and dynamically adapt to the workload conditions so that to minimize the overall tardiness experienced in a database system executing multiple concurrent I/O-intensive transactions.

TraQIOS extends the impact of SLA-aware transaction scheduling policies to the I/O sub-system and performs two functions: (1) It assigns each I/O request an intermediate deadline given the transaction's global deadline and system status, and (2) it schedules pending I/O requests according to their intermediate deadlines using a novel SLA-aware I/O scheduling policy.

The I/O scheduling policy used in TraQIOS dynamically combines the features of scheduling policies designed for improving I/O throughput (e.g., SSTF) with those of deadline-aware scheduling policies (e.g., Earliest Deadline First (EDF)). This enables our proposed policy to dynamically adapt to the workload and constantly improve the provided performance. To that end, TraQIOS extends the synergy between SLA-awareness at the DBMS level to the I/O scheduling at the operating system level.

The **contributions** of this paper are summarized as follows:

1. We study the trade-off between several I/O scheduling policies in terms of optimizing tardiness of transactions under different workload conditions.

2. We introduce a new scheme, TraQIOS, as an integrated component of the DBMS data manager for handling I/O accesses to disk-resident data. TraQIOS adaptively combines the best features of existing I/O schedulers guided by the high-level SLA specifications assigned to transactions.

3. We provide an experimental evaluation of our proposed scheme which shows that the two components of TraQIOS, namely, 1) the mechanism for assigning deadlines to I/O requests, and 2) the policy for scheduling individual I/O requests, both outperform existing schemes.

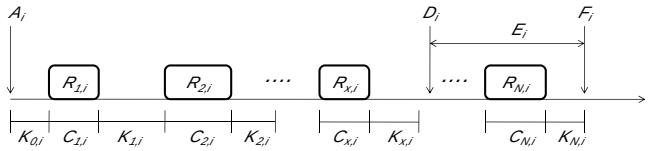

Figure 1: Transaction Model

Roadmap: The rest of this paper is organized as follows. Section 2 provides the system model. Our proposed TraQIOS scheme for SLA-aware I/O scheduling is presented in Section 3. Section 4 describes our experimental testbed, whereas Section 5 discusses our experiments and results. Section 6 surveys related work. The paper concludes in Section 7.

2. SYSTEM MODEL

In this work, we assume a traditional database management system (DBMS) which supports transaction execution with synchronous I/O operations that suspend the transaction execution until they are serviced.

We model each transaction (T_i) as a sequence of pairs of I/O requests $R_{i,x}$ and processing time $K_{i,x}$, starting with processing time $K_{i,0}$, where $R_{i,x}$ is the x^{th} I/O request in transaction T_i and $K_{i,x}$ models the amount of time spent processing the data fetched by $R_{i,x}$ or preparing new data to be read or written by $R_{i,x+1}$ in a synchronous fashion (Figure 1).

TraQIOS receives I/O requests as issued according to a transaction schedule generated by the employed concurrency control mechanism of the DBMS. The choice of the concurrency control mechanism bears no significance to *TraQIOS* which operates as a layer in between the database system and the I/O system where it handles I/O requests as they come. It could be two-phase locking (2PL) or multiversion [4], ensuring serializability, snapshot isolation [9] or some other weaker level of isolation [3].

In our model, each transaction T_i is associated with the following three parameters:

1. *Transaction Arrival Time* (A_i): The time when T_i has arrived at the DBMS.

2. *Transaction Deadline* (D_i): The SLA assigned to T_i which represents the ideal time when T_i should finish execution.

3. *Transaction I/O Length* (N_i): The number of I/O requests issued by T_i as estimated by the query optimizer.

Notice the N_i is the expected total number of page requests issued by transaction T_i, which includes the I/O operations required for accessing tables, indexes, as well as intermediate tables used internally by a transaction such as temporary storage allocated by join queries or external merge sort. However, the actual number of page I/O requests might often deviate from the one estimated by the query optimizer. This is mainly because of inaccuracy in the histogram-based statistics or lack of indexes on the accessed data. However, this inaccuracy is often limited which makes N_i a good estimate of the actual number of I/O requests issued by the T_i.

Also notice that a portion of the issued requests are served directly from the buffer pool cache if the corresponding pages are

available in memory. For pages not in the buffer pool, an I/O request is issued to the storage sub-system. We call the latter, physical I/O requests and these are the ones handled by TraQIOS. Each physical I/O request $R_{i,x}$ is characterized by the following parameters:

1. *Request Block Number* ($B_{i,x}$): The address of the block requested by $R_{i,x}$,

2. *Request Cost* ($C_{i,x}$): The disk access time incurred by request $R_{i,x}$.

Typically $C_{i,x}$ is composed of three components: (1) seek time, (2) rotational time, and (3) transfer time.

However, seek time is the most fluctuating component in computing the cost of a request since it involves the mechanical movement of the disk head to the respective address of the request. For this reason, the disk performance is typically captured by the average access time parameter $\overline{C}$ which is the expected time to access an arbitrary data block. Further, the main objective of disk scheduling policies is to minimize the total amount of seek time incurred in accessing blocks of data (e.g., using the *Shortest Seek Time First (SSTF)*. In this paper, we also focus on seek time and we will use the terms request cost and seek time interchangeably.

The time when transaction Q_i finishes execution is denoted as *finish time* (F_i). Ideally, F_i should be less than or equal to D_i. However, in the presence of multiple transactions issuing I/O requests concurrently, the I/O requests issued by Q_i might experience queuing delays in the storage system leading to delaying the finish time of T_i beyond D_i.

In our model, the system strives to finish executing each transaction T_i before its deadline. However, if T_i cannot meet its deadline, the system will still execute it but it will be "penalized" for the delay beyond the deadline D_i. This penalty per transaction is known as *tardiness* which is formally defined as:

DEFINITION 1. *Transaction tardiness, E_i, for transaction T_i is the total amount of time spent by T_i in the DBMS beyond its deadline D_i. That is, $E_i = 0$ if $F_i \leq D_i$, and $E_i = F_i - D_i$ otherwise.*

Hence, the system overall performance is measured using *average tardiness* which is defined as:

DEFINITION 2. *The average tardiness for N database transactions is: $\frac{1}{N}\sum_{i=1}^{N} E_i$.*

In order to gain a better understanding of the performance of I/O scheduling policies, in the rest of this paper, we assume I/O-bound transactions where I/O access is the performance bottleneck. Hence, the CPU is assumed to be lightly loaded and transactions never experience queuing delays at the CPU level. Consequently, the transaction scheduling policy used at the higher level does not bear any impact since I/O accesses are the dominant units of work.

3. I/O SCHEDULING FOR DATABASE TRANSACTIONS

In this section, we first define the problem of scheduling I/O requests for database transactions under pre-specified SLAs. Then, we present our proposed TraQIOS scheduling policy for improving the performance of I/O intensive transactions.

3.1 Problem Statement

As mentioned in the Introduction, under policies that attempt to meet the users' pre-specified SLAs (e.g., [25, 24, 26, 10]), the scheduling of I/O requests issued by a transaction is handled by the underlying operating system which is typically oblivious to the higher-level SLAs assigned to transactions. This highlights the need for a mechanism which properly maps a transaction global deadline to a local deadline associated with each I/O request as well as the need for an I/O scheduling policy which efficiently utilizes those local deadlines towards reducing the deviation from the global deadline.

The problem of assigning local deadlines to individual I/O requests is to some extent similar to that of setting local deadlines to sub-transactions in an RTDBMS so that to minimize the transaction drop rate (e.g., [14, 15, 16, 27]). However, as shown in [15], those proposed policies are highly sensitive to the workload conditions, where one might perform the best under one setting and the worst under another setting. Moreover, in this paper, our goal is to minimize transaction tardiness under soft deadlines as opposed to reducing drop rate. In Section 3.2.2, we show that this conflict in objective is efficiently addressed by our new SLA-aware slack budgeting scheme for assigning intermediate soft deadlines which is also adaptive to workload conditions.

Another key issue is that the OS typically schedules I/O requests so that to maximize throughput and/or to minimize the average latency per request. Specifically, one widely used I/O scheduling policy employed at the OS level is the *Shortest Seek Time First (SSTF)*. SSTF optimizes I/O latency by always serving the request closest to the disk head. Assuming that seek time is the only variable per disk access while rotational and transfer times are constants, SSTF could be considered as an instant of the more general *Shortest Job First* scheduling policy, which is known for optimizing average latency [23].

However, optimizing average latency is sometimes at odds with satisfying the pre-specified SLAs. In other words, minimizing latency might lead to unnecessary increases in tardiness. This is due to the fact that requests with short seek times might be given higher priority than requests issued by a transaction with an imminent deadline.

To that end, given the intermediate deadlines assigned to I/O requests, there is still the need for an efficient scheduling policy which enables minimizing the tardiness of each of those individual I/O request. From the above discussion, there are two obvious promising candidate policies to achieve that goal, namely, (1) SSTF, and (2) EDF. Formally, we define the two policies as follows:

1. **SSTF:** Each I/O request $R_{i,x}$ is assigned a priority equal to $\frac{1}{C_{i,x}}$, where $C_{i,x}$ is the cost of $R_{i,x}$.

2. **EDF:** Each I/O request $R_{i,x}$ is assigned a priority equal to $\frac{1}{D_i}$, where D_i is the deadline of R_i.

Under both policies, a scheduling point is reached whenever an I/O request is served. At that point, the priority of each I/O request is computed according to the policy used and the one with the highest priority is served first.

Studying the performance of the two policies above should clarify the trade-offs involved in the problem of scheduling I/Os with soft deadlines. Specifically, Figure 2 shows the performance of the two policies under our evaluation testbed, which is described in details in Section 4. The figure shows that EDF outperforms SSTF at low and medium disk utilization. However, at high utilization, SSTF clearly outperforms EDF.

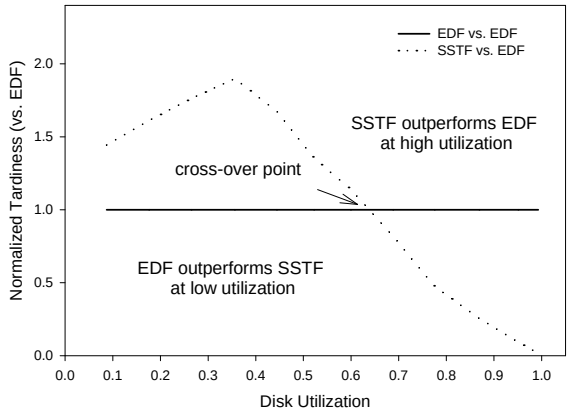

Figure 2: Trade-off between EDF and SSTF: at low-utilization, EDF reduces tardiness by up to 50%

The conflict between EDF and SSTF in the I/O scheduling context (as shown in Figure 2) is quite similar to the one observed in an earlier study [6] where both EDF and SJF were used to prioritize CPU scheduling of transactions. Specifically, as pointed out in [6], at high utilization, it is impossible to finish all transactions by the specified deadlines. Using an EDF scheduler in such high-load situations will have a substantial negative impact on the overall tardiness. This negative impact is known as the *domino effect*: transactions keep missing their deadlines in a cascaded fashion. The cause of the domino effect is that EDF might give high priority to a transaction with an early deadline that it has already missed, instead of scheduling another one which has a later deadline that could still be met. As a result, both transactions will miss their deadlines and accumulate tardiness.

In I/O scheduling, that negative impact is more prominent since it might involve a costly seek to serve a request with an early deadline instead of serving the one which is currently the closest to the disk head. On the other hand, at low utilization, EDF can afford making those costly "out-of-the-way" moves while still meeting most of the deadlines. To the contrary, SSTF might run into the problem of serving close by requests coming from transactions with long deadlines instead of scheduling requests which are relatively far from the head but they are issued by transactions with more imminent deadlines.

An obvious integration of EDF and SSTF assumes that one knows the "cross-over" point where SSTF is better than EDF [6]. However, such a point is not easy to determine since it is dependent on many factors including disk utilization, deadlines, data access pattern, and cache size. It is also expected the cross-over point will change drastically over time, as the workload changes. This motivates us to investigate a new adaptive policy for scheduling I/O requests in the presence of soft deadlines.

3.2 The TraQIOS I/O Scheduling Policy

In this section, we present TraQIOS, our scheme for scheduling I/O requests issued by transactions to disk-resident data. As mentioned in Section 1, TraQIOS performs the following two functions:

1. It assigns each I/O request an intermediate deadline given the transaction's global deadline and the system status, and
2. It schedules pending I/O requests according to their intermediate deadlines using an SLA-aware I/O scheduling policy.

To simplify the presentation, we will first describe our policy for scheduling individual I/O requests assuming that each transaction is accessing a single disk page. Then, in Section 3.2.2, we will extend our model for the general case where each transaction is accessing more than one disk page.

3.2.1 TraQIOS: Scheduling individual I/O Requests

TraQIOS assigns each pending I/O request a priority value and at each scheduling point, the request with the highest value is the one scheduled to be served. Specifically, our goal is to assign each pending I/O request a certain priority so that to minimize the average transaction tardiness when requests are served in descending order of their priority values.

In order to compute the priority of a transaction T_i under TraQIOS, let us first assume that transaction T_i has issued an I/O request $R_{i,1}$ (or just R_i since in this section we are considering single I/O transactions). Further, assume that C_i is the cost of request R_i and K_i is the processing time of the data fetched by R_i as defined in Section 2.

Additionally, let S_i be the current *slack* of transaction T_i if its I/O request is served immediately. Formally,

DEFINITION 3. *Transaction slack, S_i, for transaction T_i at time t is the amount of time between T_i's deadline (D_i) and its finish time if T_i is scheduled right now (i.e., it is scheduled at the current time t). Hence, $S_i = D_i - (t + C_i + K_i)$.*

As such, the tardiness of transaction T_i is dependent on its current slack S_i. In particular,

1. If $S_i < 0$, then T_i will pass its deadline even if R_i is served immediately, leading to T_i accumulating a tardiness of $|S_i|$ time units.
2. If $S_i \geq 0$, then T_i will meet its deadline if R_i is served immediately, leading to T_i finishing $|S_i|$ time units before its deadline.

Given the slack time S_i of Transaction T_i and the parameters of request R_i, we assign R_i a priority value which is computed as follows (for details on computing this priority, please see Appendix A):

$$P_i = \begin{cases} \frac{1}{C_i} & S_i \leq 0 \\ \frac{1}{C_i}(1 - \frac{S_i}{n\overline{C} - C_i}) & 0 < S_i \leq n\overline{C} - C_i \\ 0 & S_i > n\overline{C} - C_i \end{cases} \quad (1)$$

where $\overline{C}$ is the *average request cost* which is the expected time needed for the head to travel between two arbitrary blocks on the disk and n is the number of pending I/O requests.

To understand the intuition underlying our policy, notice that our priority function assigns R_i a priority of $\frac{1}{C_i}$ if $S_i < 0$ (i.e., if T_i cannot make its deadline). Clearly, this is the same priority that would have been assigned to R_i under the SSTF policy which orders request according to their seek time from the current head location.

As the value of S_i increases, our policy *reduces* the priority of R_i since it has some slack which allows serving requests from other transactions, since there is no reward if transaction T_i finishes before its deadline.

Our policy reduces the priority of R_i until $P_i = 0$ when $S_i > n\overline{C} - C_i$. The reason for that is easily explained considering some arbitrary request R_x which is competing with R_i over the disk access. For that request R_x, it takes on average $\overline{C}$ units to access

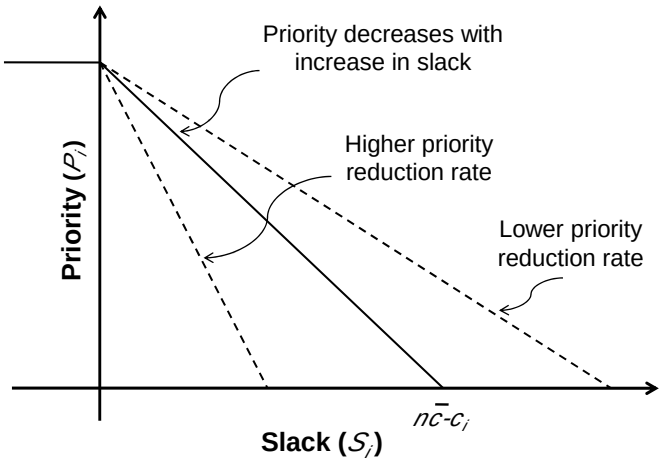

Figure 3: Priority as a function in slack

the data block requested by R_x and $\overline{C}$ more time units to move the head back and fetch the block requested by R_i. Hence, if R_x is scheduled first, then R_i is expected to be served in $2\overline{C}$ time units instead of C_i time units.

If the increase in R_i's serving cost is still lower than R_i's slack (i.e., $2\overline{C} - C_i < S_i$), then R_i can still meet its deadline even if another request (i.e., R_x) is scheduled first. Hence, R_i can safely allow R_x to be executed first by setting its own priority P_i to zero. If that increase is higher than R_i's slack (i.e., $2\overline{C} - C_i > S_i$), then R_i will lower its priority according to the ratio between its slack and the extra incurred cost.

In the above discussion, for simplicity, we have assumed that R_i is competing with a single arbitrary request R_x over the disk access. However, in reality, at each scheduling point R_i is competing with all the requests in the pending queue. Ideally, R_i should reduce its priority to zero if its slack can accommodate serving all the other pending requests first. In such case, R_i is guaranteed a zero tardiness even if all those pending requests are served first. Assuming that there are n pending requests including R_i, then the expected time needed to serve the other requests is: $(n-1)\overline{C}$. Hence, the increase in R_i's cost if those n requests are served first is: $(n-1)\overline{C} + \overline{C} - C_i$ which results in the priority function above (Eq. e:defInstant).

Under our priority function, the highest priority assigned to request R_i is $1/C_i$. This priority decreases linearly depending on the slack available for transaction T_i which issued R_i as shown in Figure 3. The rate (or slope) of priority reduction is determined by the ratio between S_i and $n\overline{C}$. Hence, the value $n\overline{C}$ acts as a knob which allows our policy to integrate EDF scheduling with SSTF scheduling. Specifically,

- If $n\overline{C}$ is high, the priority reduction rate is slow and our policy tends to behave like SSTF,
- If $n\overline{C}$ is low, the priority reduction rate is high and our policy tends to behave like EDF.

Naturally, n increases with the increase in I/O utilization moving our policy more towards SSTF-like scheduling. Similarly, n decreases with the decrease in utilization moving our policy more towards EDF-like scheduling. This allows our policy to dynamically adapt to the utilization as we will show experimentally in Section 5.

3.2.2 TraQIOS: Slack Budgeting Mechanism

In the previous section, we have discussed the adaptive policy employed by TraQIOS for scheduling I/O requests. Such policy is sufficient if each transaction is accessing a single disk page. However, in order to extend our scheme to handle the general case where each transaction issues multiple I/O requests, the main problem to address is how to set the slack value associated with each request (see Eq. 1). The general criterion is to set such slack so that to eventually minimize the transaction tardiness beyond its "global" deadline.

To this end, the slack assigning problem is similar to that of setting local deadlines to sub-transactions in an RTDBMS so that to minimize the transaction drop ratio under hard deadlines (e.g., [14, 15, 16, 27]). These schemes proposed in RTDBMSs are designed to work under the umbrella of the EDF policy. Given that our goal in this paper is to minimize transaction tardiness under soft deadlines, we are seeking a new slack budgeting schemes which works in synergy with our adaptive policy proposed above. Towards this goal, we need to answer the following two basic questions:

1. How much is the current slack S_i of transaction T_i?
2. How much of S_i is to be assigned to each request $R_{i,x}$ issued by T_i?

To answer the first question, consider a transaction T_i issuing requests $R_{i,0}$, $R_{i,1}$,..., R_{i,N_i}. Further, assume that at time t, T_i has issued its request $R_{i,x} (0 \leq x \leq N_i)$. Let $M_{i,x}$ be the number of remaining requests to be issued by T_i (i.e., $M_{i,x} = N_i - x$). Hence, at time t, T_i has $M_{i,x}$ more I/O requests to issue and it has $D_i - t$ time units until it reaches its deadline.

However, the amount of time $D_i - t$ is not all slack since the transaction will be performing more I/O accesses and CPU processing during that interval. Thus, the maximum slack available for T_i at time t is: $S_i = D_i - t - C_{i,x} - K_{i,x} - C_{i,x+1} - K_{i,x+1} - ... - C_{i,N_i} - K_{i,N_i}$, where $C_{i,x}$ is disk access cost of $R_{i,x}$ and $K_{i,x}$ is the amount of time spent processing the data fetched by $R_{i,x}$.

The processing times K_i are simply estimated either by the query optimizer or by monitoring the prior executions of the transaction T_i. For the disk access costs C_i, the actual costs of requests $R_{i,x+1}$, ..., R_{i,N_i} depend on their respective distance from the disk head at the time when they are issued, hence, we assume that the cost each of those requests is the average access time $\overline{C}$. Thus, the expected $S_i = D_i - t - C_{i,x} - K_{i,x} - M_{i,x}\overline{C} - \sum_{j=x+1}^{N_i} K_{i,j}$.

The remaining number of requests M_i is initially set to the expected transaction I/O length N_i as estimated by the database query optimizer. This includes the I/O operations required for accessing tables, indexes, as well as intermediate tables used internally by a transaction such as temporary storage allocated by join queries or external merge sort. Then, M_i is decremented whenever one of the following happens:

1. A page access is satisfied by a physical request to the I/O sub-system, or
2. A page access is satisfied by a logical request to the database buffer pool cache.

In both of the above cases, transaction T_i is accessing pages included in the initial estimate of the number of I/O requests (i.e., N_i). However, in the presence of inaccuracy in estimates and T_i ended up processing more pages than N_i, then every request beyond N_i will leave M_i equal to 1.

To answer the second question above on how much slack to assign each I/O request, recall that the amount of slack *advertised* by

an I/O request plays an important role in determining its priority. For instance, if request R_i advertises a high slack value, then its priority is reduced to allow other requests to use the slack available for transaction T_i. However, this might lead to delaying the future M requests to be issued by T_i and eventually causes T_i to experience high tardiness. On the other hand, advertising a low slack value deprives other transactions from taking advantage of whatever slack T_i might have. In summary, advertising a low slack value leads to an SSTF-like scheduler where slacks are not exploited, whereas advertising a high slack value leads to an EDF-like scheduler where slacks are used to accommodate more urgent requests.

To balance the trade-off in setting the slack value described above, we propose using a *slack budgeting* mechanism which allows each transaction to budget its slack given the current system conditions. Specifically, we propose setting the slack value $S_{i,x}$ assigned to I/O request $R_{i,x}$ as follows:

$$S_{i,x} = \frac{M^U_{i,x}}{M_{i,x}} S_i \tag{2}$$

where S_i is the total expected slack of T_i, $M_{i,x}$ is the number of remaining requests to be issued by T_i, and U is the average transaction *success rate* which is the percentage of transactions finishing before their deadlines as monitored by the system.

The intuition is that a high success rate is a good indication that an EDF scheduling policy will perform well. Hence, our scheduler tries to inject more EDF-like scheduling by making transactions advertise higher slacks. On the other hand, a low success rate indicates that EDF is entering the domino effect phase where transactions keep missing their deadlines. In that case, our policy moves more towards SSTF-like scheduling, where transactions are very strict about the amount of slack they advertise. For instance, at $U = 1$, each I/O request $R_{i,x}$ is associated with the entire slack remaining for T_i, whereas at $U = 0$, a transaction will budget its remaining slack uniformly over its remaining requests.

Notice that above two extremes (i.e., $U = 0$ and $U = 1$) are equivalent to the policies for assigning deadlines to sub-transactions under hard deadlines in RTDBMSs as proposed in [15]. Specifically, [15] proposed the Ultimate Deadline policy where a sub-transaction inherits the global deadline of its parent transaction. It also proposed the Equal Slack policy where it divides the total remaining slack equally among the remaining subtasks. However, it has been shown that both policies are very sensitive to the workload conditions. In particular, the Ultimate Deadline policy performs reasonably well under low utilization, whereas at high utilization, it is outperformed by the Equal Slack policy. Moreover, in [15], sub-transactions are scheduled using EDF, whereas under TraQIOS, I/O scheduling is performed according to our adaptive SLA-aware policy. In particular, given our slack budgeting policy, our general priority function for scheduling I/O requests is defined as follows:

$$P_{i,x} = \begin{cases} \frac{1}{C_{i,x}} & S_{i,x} \leq 0 \\ \frac{1}{C_{i,x}}\left(1 - \frac{S_{i,x}}{m\overline{C} - C_{i,x}}\right) & 0 < S_{i,x} \leq m\overline{C} - C_{i,x} \\ 0 & S_{i,x} > m\overline{C} - C_{i,x} \end{cases} \tag{3}$$

where $S_{i,x}$ is computed as in Eq.2 and m is the expected total number of remaining I/O requests issued by the set of transactions currently in the system.

3.2.3 Discussion

A remaining question is how TraQIOS can be implemented. There are several light-weight alternatives for the low-level physical I/O instrumentation of TraQIOS. Here, we discuss two of them.

The first one is to modify the existing Linux I/O deadline-based scheduler to accommodate general priority assignments beyond just deadlines. This has been the approach followed in [11] where each I/O request is assigned a deadline based on its priority. Similarly, in our case, we will implement a TraQIOS priority assignment where a request with a high priority under TraQIOS will receive a short deadline, whereas a request with a low priority will receive a long deadline.

Another alternative is to modify the *Anticipatory* scheduler [12] which is the default scheduler in Linux 2.6. One advantage of this approach is that the Anticipatory already computes the seek time for each request from the current head position. Hence, the intermediate deadline assigned to a request by TraQIOS is the only other parameter the scheduler needs in order to compute our proposed priority function. That intermediate deadline is simply passed from the storage manager (e.g., MySQL/InnoDB) through the standard arguments and system calls defined by POSIX.

4. EVALUATION TESTBED

We created a simulator to model transaction execution in a DBMS system. In this section, we describe the experimental setup, whereas and the experimental results are presented in Section 5.

Metrics: We mainly compare the performance of different policies in terms of the provided average tardiness as defined in Section 2. However, for the sake of completeness, we as also report results on the transaction miss ratio provided by TraQIOS and compare it against existing policies which are known for optimizing such metric.

Policies: We conducted several experiments to compare the performance of our proposed *TraQIOS* policy against the previously described *EDF* and *SSTF* policies. We have also compared TraQIOS against two hybrid scheduling policies, namely, *FD-SCAN* [1] and *SSEDV* [8].

FD-SCAN is a deadline-aware modification of the traditional elevator algorithm SCAN. Specifically, under FD-SCAN, the track location of the request with the earliest feasible deadline is used to determine the scan direction. The head seeks in the direction of that request serving all other requests along the way until it reaches the target track.

In SSEDV, each I/O request is assigned a priority which is defined as:

$$P_i = \alpha C_i + (1 - \alpha) l_i \tag{4}$$

where C_i is the seek time to serve the I/O request, l_i is the amount of time between the current time and R_i's deadline D_i, and α is a scheduling parameter. This allows SSEDV to perform as a hybrid between SSTF and EDF depending on the value of α. In our experiments, we set α to 0.8 as suggested in [8]. Moreover, SSEDV assigns each request a *step deadline*, where the step deadline assigned to the request number x in transaction T_i is set as:

$$d_i = A_i + \frac{x}{N_i}(D_i - A_i) \tag{5}$$

where D_i is the deadline, A_i is the arrival time, and N_i is the total number of I/O requests in T_i.

Disk: We simulated a disk drive as in [22, 8, 2]. The modeled disk has 1000 cylinders where the access time of an I/O request is

Parameter	Value	Default
Scheduling policies	EDF, SSTF, FD-SCAN, SSEDV, TraQIOS	
Disk Cylinders	1000	
Disk Average Access Time	5.8 & 9.4 mSec	9.4 mSec
Transactions	5000	
Requests per Transaction	1–1000	
System Utilization	0.1 – 0.99	
Database Layout	100 & 1000 cylinders	1000 cylinders
Data Access (p_s)	0.0 (Highly Random) & 0.8 (Highly Sequential)	Random
$SlackFactor_{max}$	2 – 6	4
Cache Hit Probability (p_h)	0% – 50%	0%
Inaccuracy in Optimizer I/O Estimates (p_l)	0% – ±50%	0%

Table 1: Simulation Parameters

defined as $a + b\sqrt{d}$, where a is the average rotational latency, b is the seek factor, and d is the number of cylinders between the current cylinder and the target cylinder. In our simulation, we set a to 4.0 mSec, whereas b is set to either 0.1 or 0.3. Setting $b = 0.3$ allows an average access time of 9.4 mSec which is around the standard for today's workstation disk drives, whereas $b = 0.1$ allows an average access time of 5.8 mSec which is the typical for disk drives used in large server farms.

Transactions: We generated 5000 transactions where the number of I/O requests issued by each transaction is picked uniformly from the range [1–1000]. After an I/O is served, the transaction will wait for a think time period before generating a new request. This interval simulates the processing performed by the transaction on the fetched data. In order to create a workload where I/O is the bottleneck, the think time is picked uniformly from the range [0.0–1.0] mSec. This, in combination with the disk average access times above, result in I/O forming 60% to 95% of the transaction total execution time. The generated transactions arrive at the system according to a Poisson distribution where we vary the inter-arrival time of the Poisson distribution to simulate different utilizations.

Data Access: In the default setting, the database pages are stored uniformly across the 1000 cylinders of the disk drive. Hence, the address of the data block accessed by each request is generated uniformly within that range. However, in order to examine performance in the presence of hot spots, we experimented with settings where the database occupy fewer cylinders. Moreover, we also simulated both random and sequential data access. Specifically, we use a parameter p_s which determines the probability for the next block accessed by a transaction to fall on the same cylinder as the previously accessed block.

Deadlines: Each transaction T_i is assigned a deadline $D_i = A_i + N_i\overline{C} + SlackFactor_i \times (N_i\overline{C})$, where for T_i, A_i is the arrival time, N_i is the number of I/O requests , $\overline{C}$ is the average access time, and $SlackFactor_i$ is a parameter which determines the ratio between the initial slack time of a transaction and its expected length.$SlackFactor_i$ is generated uniformly over the range [0.0–$SlackFactor_{max}$], where $SlackFactor_{max}$ is a simulation parameter.

Cache: In order to simulate the impact of the database buffer pool, we introduce a simulation parameter p_h which determines the probability of a cache hit. Specifically, whenever an I/O request is generated, a coin is tossed and if the coin value is less than p_h, then it is considered a hit and the request is served directly from memory. We find this approach general enough so that to decouple the performance evaluation here from the implementation details of particular cache replacement policies (e.g., LRU, LFU, etc.).

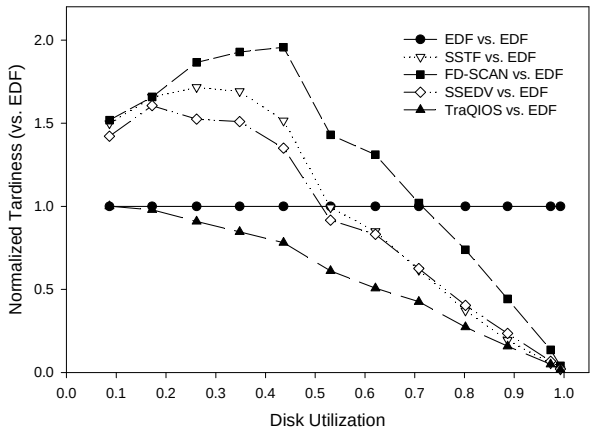

Figure 4: At utilization 0.53 (cross over point), TraQIOS reduces tardiness by 40% compared to EDF, SSTF, and SSEDV

Inaccuracy in Transaction Length: In order to model inaccuracies in the query optimizer estimates, we we introduce a new parameter p_l to control the percentage of deviation between the actual number of I/Os issued by a transaction and the estimated number of I/Os provided by the query optimizer. In our setting, p_l is in the range [0% – ± 50%] where a value of 0% indicates no deviation between the actual number of I/Os and the estimated one, whereas at a non-zero value, the actual number of I/Os increases or decreases accordingly.

In the next section, we present a representative sample of our experimental results under the settings summarized in Table 1.

5. EXPERIMENTS

In this section we present the performance results under the settings described in Section 4. The values reported here are the averages of at least 5 runs for each experimental setting.

5.1 Tardiness

In our first experiment, we measured the average tardiness for the scheduling policies mentioned above as the system utilization increases from 0.1 to 1.0, with $SlackFactor_{max} = 4.0$. The results for this experiment setting are shown in Figure 4 where we plot the performance of each policy normalized to the EDF policy. Hence, the lower the value, the better the performance compared to EDF. The figure shows that at low utilization, the system is able to

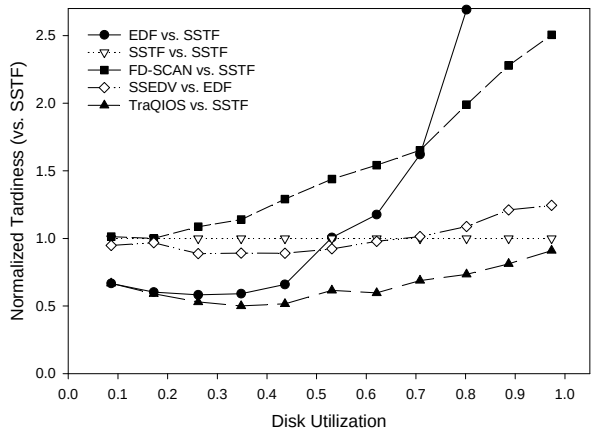

Figure 5: Performance normalized to SSTF shows the relative performance at high utilization

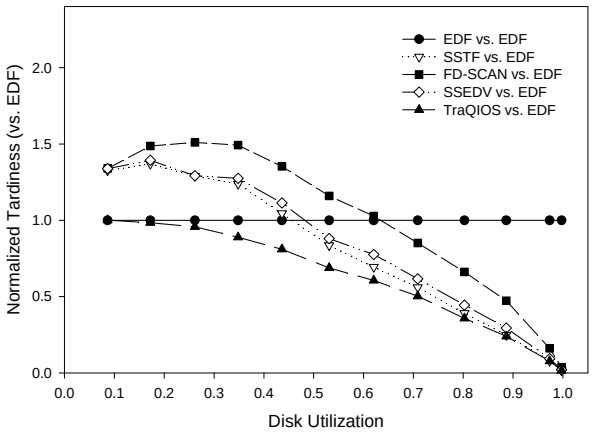

Figure 6: At Slack Factor = 2, the cross over point between EDF and SSTF moves to lower utilization of 0.43 where TraQIOS reduces tardiness by up to 20%

meet most of the deadlines, and hence, EDF performs better than SSTF, SSEDV, and FD-SCAN. As the utilization grows, the system cannot meet as much deadlines. As such, SSTF and SSEDV start to approach EDF until they both outperform it at utilization close to 0.53.

TraQIOS on the other hand, outperforms the other policies for all values of utilization. Notice that the maximum improvements provided by TraQIOS is around the cross over point between EDF, SSTF, and SSEDV, where TraQIOS reduces the average tardiness by 40% compared to SSEDV. However, TraQIOS still improves the performance at both ends of the utilization range. For instance, it reduces tardiness by 10% compared to EDF at utilization 0.25. TraQIOS also reduces tardiness by 20% compared to SSTF at utilization 0.9 (as shown in Figure 5).

The reason that TraQIOS still out performs EDF at low utilization, is that there are still intervals where the utilization increases significantly above the average due to the fact that we are using Poisson arrivals. At those high utilization intervals, TraQIOS automatically incorporates some SSTF-like scheduling to avoid the domino effect of EDF. Similarly, at high utilizations, TraQIOS outperforms SSTF as it incorporates some EDF scheduling as needed.

For SSEDV, in general, its performance follows the same pattern as SSTF. This is because its priority function gives high weight (i.e., $\alpha = 0.8$ as mentioned in Section 4) to the disk seek time which brings it very close to SSTF. However, SSEDV outperforms SSTF at lower utilization. This is due to the fact that SSEDV also considers the remaining time until a transaction's deadline when prioritizing I/O requests. But since the seek time still dominates the priority function, SSEDV is not able to outperform EDF or TraQIOS at low utilization.

From Figure 4, it might seem that at high utilization, SSEDV provides a performance similar to that of SSTF and TraQIOS. However, in order to study the performance at high utilization, we plot the performance of all policies normalized to SSTF in Figure 5. The figure shows that TraQIOS outperforms SSTF for all high utilization values until when utilization is equal to 1, that is when it provides the same performance as SSTF. However, SSEDV is not able to provide such performance since that even at high utilization, it still gives a constant weight to the transaction deadline which hurts its performance as it does to EDF. TraQIOS on the other hand, automatically controls the amount of EDF scheduling according to the workload conditions.

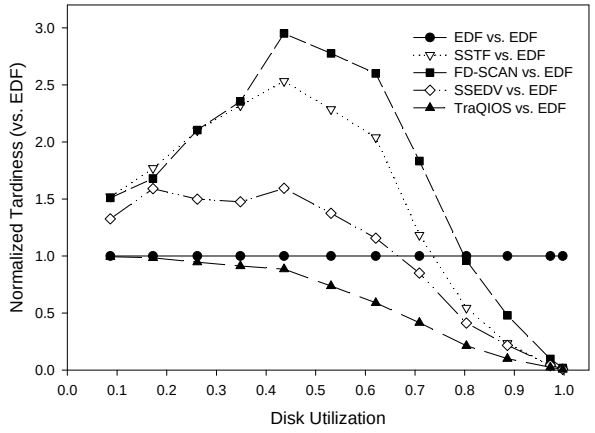

Figure 7: At Slack Factor = 6, the cross over point between EDF and SSTF moves to higher utilization of 0.75 where TraQIOS reduces tardiness by up to 50%

5.2 Impact of Slack Factor

This set of results shows the performance of our proposed algorithm under different deadline settings (i.e., under different values of $SlackFactor_{max}$). Basically, the value of $SlackFactor_{max}$ determines how tight or loose are the deadlines where the lower the value, the tighter the deadlines. Figures 6, and 7 show the results for $SlackFactor_{max}$ values of 2, and 6 respectively. This is in addition to the results of $SlackFactor_{max} = 4$ presented above in Figures 4 and 5.

The results show that TraQIOS constantly outperforms the other algorithms under the different settings, with the maximum gain be at the cross-over area. It is also interesting to notice that as we increase the value of $SlackFactor_{max}$ (i.e., relaxed deadlines), EDF's good performance at low utilization is more prominent where the magnitude of its gains compared to SSTF are higher and also where the cross over point between the two moves further to the right (i.e., higher utilization). The reason is that the more relaxed the deadlines are (i.e., larger $SlackFactor_{max}$), the more chances EDF can catch up if it missed deadlines. Hence EDF can cope with higher utilization and outperforms SSTF for a longer range of utilization.

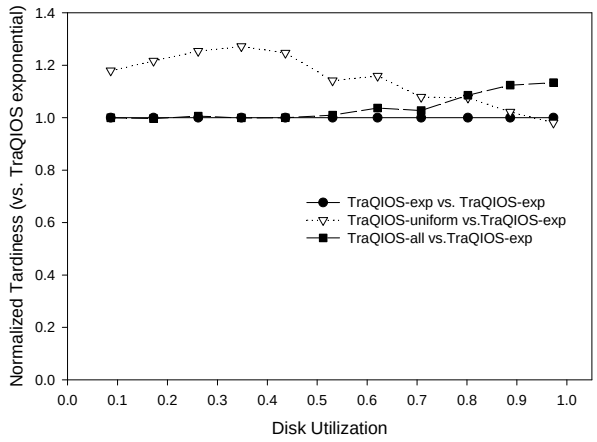

Figure 8: The trade-off between alternative slack budgeting mechanisms

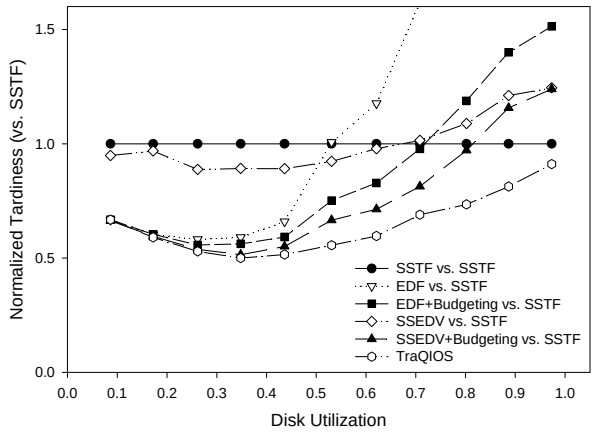

Figure 9: Improving the performance of EDF and SSEDV by augmenting them with the exponential slack budgeting scheme

5.3 Alternative Schemes for Slack Budgeting

In this experiment, we evaluate the performance of three alternative schemes for setting the slack associated with each I/O request issued by a transaction as discussed in Section 3. The three settings are as follows:

1. *Exp:* is the slack budgeting scheme we proposed in Section 3 where the slack associated with each I/O request is a function in the success rate (i.e., $S_{i,x} = \frac{M^U_{i,x}}{M_{i,x}} S_i$),

2. *All:* is an alternative budgeting scheme where the slack associated with each I/O request is equal to the transaction's current slack (i.e., $S_{i,x} = S_i$), and

3. *Uniform:* is an alternative budgeting scheme where the slack associated with each I/O request is inversely proportional to the number of remaining I/O requests (i.e., $S_{i,x} = \frac{1}{M_{i,x}} S_i$).

In Figure 8 we plot the average tardiness provided by each scheme normalized to TraQIOS-exp, hence, the lower the value the better the relative performance. The figure shows that TraQIOS-all outperforms TraQIOS-uniform at low utilization where TraQIOS-all behaves more like EDF and efficiently utilizes the large advertised slacks. At high utilization, TraQIOS-uniform outperforms TraQIOS-all by advertising a small slack with each I/O request and hence behaving like SSTF. The figure also shows that TraQIOS-exp is able to balance the trade-off between the two extremes since it considers the current system success ratio for slack budgeting. This allows TraQIOS-exp to reduce tardiness by to 13% compared to TraQIOS-all at high utilization and by up to 27% compared to TraQIOS-uniform at low utilization.

In order to distinguish between the gains provided by our policy for scheduling I/O requests and the gains provided by our slack budgeting scheme, we modified the deadline-aware I/O scheduling policies to work under our slack budgeting scheme. Specifically, we extended EDF so that the deadline for each request is assigned according to our exponential slack budgeting policy. Similarly, we replaced the step deadline scheme in SSEDV with our scheme.

Figure 9 shows the performance results for the modified policies (i.e., EDF+Budgeting and SSEDV+Budgeting) together with the original versions as well as TraQIOS where all are normalized to SSTF. The figure shows the significant improvement in the performance of each policy under our proposed budgeting scheme. However, TraQIOS still outperforms the two policies since in addition to the budgeting scheme, it also employs an adaptive I/O request scheduling policy.

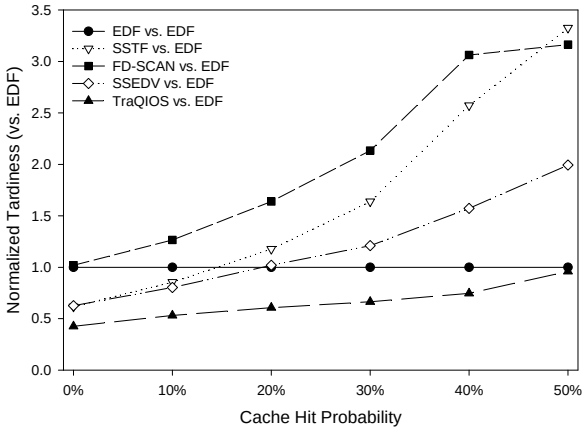

Figure 10: Normalized Tardiness at utilization 0.8: increasing the cache hit rate is equivalent to reducing the disk utilization

5.4 Impact of Cache Hit Rate

In this experiment, the value of the cache hit rate parameter (p_h) is in the range of [0%–50%] as opposed to the default value of 0%. Figure 10 shows the tardiness provided by each policy at utilization 0.8 normalized to that of EDF. Notice that the absolute tardiness decreases for all policies with increasing p_h (which is not shown in Figure 10).

The figure shows that at low hit rate, SSTF and SSEDV outperform EDF, whereas at high hit rate, EDF outperforms them both. The reason for such performance pattern is that increasing the cache hit rate is to some extent equivalent to decreasing the disk utilization as it reduces the amount of I/O requests submitted to the disk. As we have already seen in the previous experiments, EDF performs relatively well at low utilization, hence, it also performs well in the presence of large cache or a highly cachable workload. For TraQIOS, at low hit rate (i.e., high utilization), it employs some SSTF-like scheduling which allows it to outperform EDF, whereas

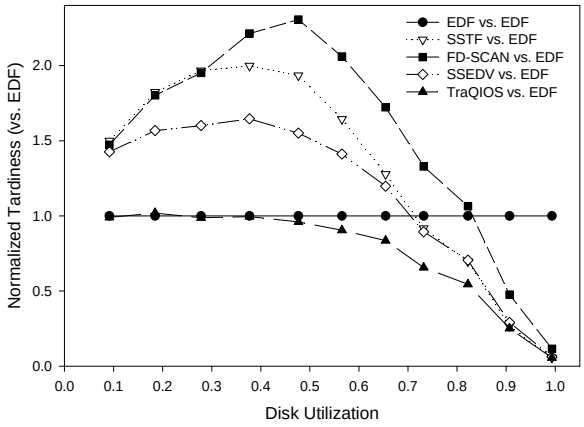

Figure 11: Performance under low seek time: disk access time = 5.8 mSec

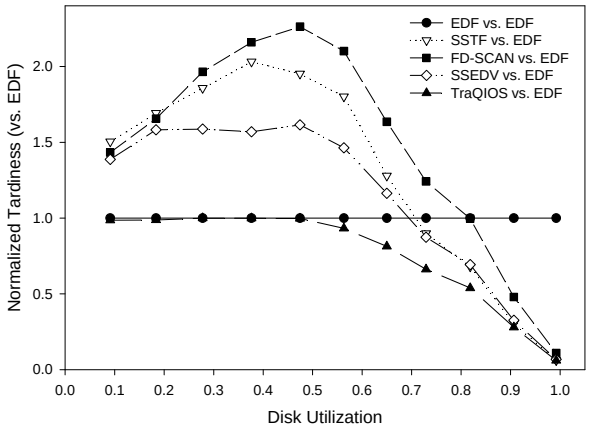

Figure 12: Performance under clustered data access: database occupies 10% of available disk cylinders

at high hit rate (i.e., low utilization), TraQIOS adapts and becomes mostly EDF, hence providing the same performance as EDF.

5.5 Impact of Seek Time

In this experiment, $SlackFactor_{max}$ is set to its default value of 4, whereas the seek factor parameter is set to 0.1 as opposed to the default value of 0.3 resulting in an average access time of 5.8 mSec as opposed to 9.8 mSec.

Figure 11 shows the results under this setting where all policies exhibit the same performance as in the default setting. However, notice that cross over point between EDF, SSTF, and SSDEV has moved to a higher utilization of about 0.75 as opposed to 0.53 (Figure 4). The reason is that reducing the seek time has the effect of reducing the *ratio* between the maximum and minimum time to serve a request which plays an important role in shaping the performance. In particular, the minimum service time is when the block is on the same cylinder as the current head location, whereas the maximum service time is when the head has to travel from the first cylinder to the last one. For seek factor 0.3, that maximum to minimum ratio is 3.3, whereas at seek factor 0.1 that ratio is only 1.7.

Given the above ratios, at a low seek factor (i.e., 0.1), EDF can serve a request which is close to its deadline yet far from the current head location (i.e., relatively large seek time) without severely penalizing those pending requests with short seek time. In particular, the worst delay incurred by a request will be 2×1.7 times its service time. To the contrary, at a higher seek factor, serving a request with a longer seek time results in a larger penalty for all the other pending requests. By balancing the trade-off between seek time and deadlines, TraQIOS is able to provide the best performance under low seek time as it does under high seek time. For instance, at the cross over point of 0.75, TraQIOS reduces latency by up to 30%.

5.6 Impact of Data Access Patterns

Clustered data access results in the same performance pattern exhibited when reducing the seek time as explained in the above the experiment. This is shown in Figure 12 where we use the default seek factor (i.e., 0.3), however, the database is clustered in only 100 cylinders of the total 1000 cylinders available on the disk. Hence, the maximum distance the disk head might travel is reduced and accordingly the ratio between the maximum and minimum request service time is reduced. This results in a performance similar to that when reducing the seek time with the cross over point moving to higher utilization of 0.75.

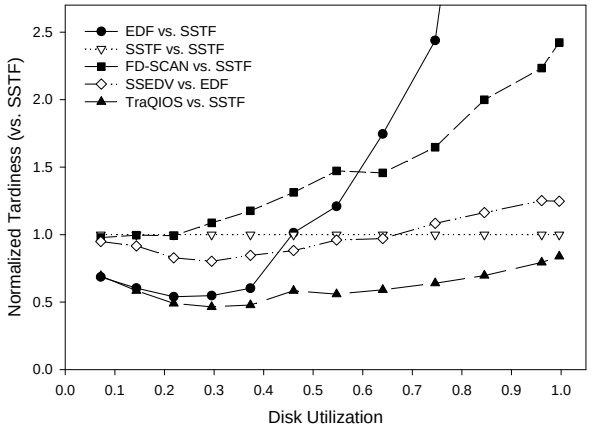

Figure 13: Performance under highly sequential data access

In Figure 13, we show the results for an experimental setting where the database is distributed over the disk 1000 cylinders, however the data access is highly sequential. In particular, we set the data access parameter p_s to 0.8 so that whenever a transaction issues a request, a coin is tossed and if the coin value if less than 0.8, then the accessed block is on the same cylinder as the last blocked accessed by that transaction. Figure 13 shows that under such pattern seek-time-aware policies (i.e., SSTF, SSDEV, and TraQIOS) remarkably outperform EDF. This is manifested by the cross over point between EDF and SSTF moving to a low utilization of 0.45 as opposed to 0.53 in default setting (Figure 4).

The reason is that under SSTF, the head makes very short moves between the successive sequential requests issued by a transaction, say T_s. Hence, by the time T_s issues a new sequential request, the head will most likely be very close to that cylinder sequentially accessed by T_s. This is in contrast to EDF which might move the head far away from that cylinder. Hence, under this setting, TraQIOS employs more SSTF-like scheduling which allows it to provide the best performance where it reduces tardiness by up to 40%.

5.7 Impact of Inaccuracy in I/O Length

In all the previous experiments, we have assumed that the actual number of I/O requests generated by transaction T_i is equal to the number of I/O requests estimated by the query optimizer (i.e., N_i).

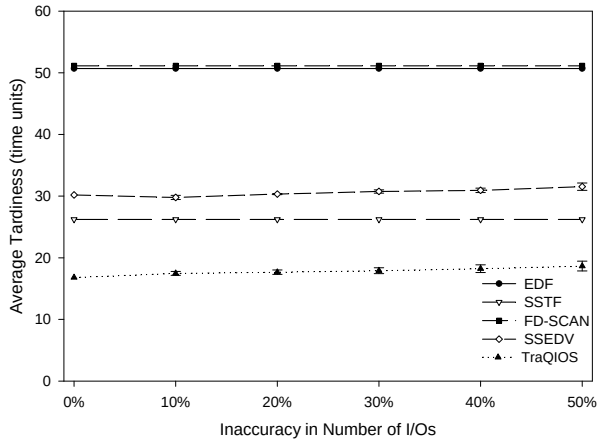

Figure 14: Average increase in tardiness vs. inaccuracy in Transaction I/O Length (inaccuracy uncorrelated with transaction length)

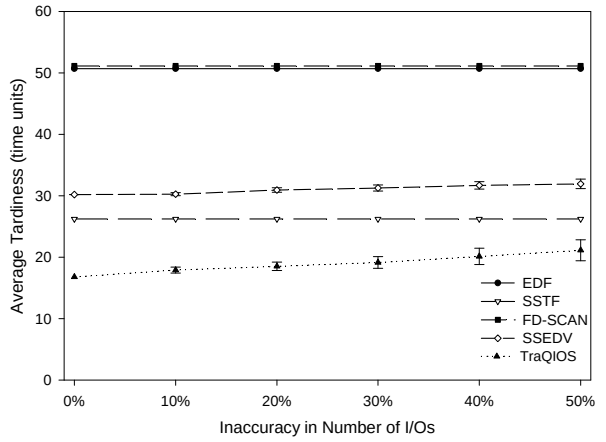

Figure 15: Average increase in tardiness vs. inaccuracy in Transaction I/O Length (inaccuracy correlated with transaction length)

In this experiment, we study the performance in the presence of inaccuracy in the query optimizer estimates. Specifically, in this experiment, the actual number of I/O requests generated by a transaction deviates from the estimated number by a certain percentages which simulates the inaccuracy of the query optimizer. However, the scheduling policy still assumes the estimated number of I/O requests and computes its priority accordingly. Notice that this will only affect the performance of SSEDV and TraQIOS since they are the two policies which utilize the estimated number of I/Os.

In Figure 14, we plot the performance at utilization 0.8 where the inaccuracy is in the range [0% – $\pm$50%]. In order to account for the randomness in inaccuracy across multiple experimental runs, we also report the confidence interval for a 95% confidence level where the lower the confidence interval, the closer the expected tardiness to the reported average.

The figure shows that the average tardiness provided by both SSEDV and TraQIOS increases with increasing the inaccuracy. It also shows a wider confidence interval in reported tardiness for both policies with increasing the inaccuracy. However, that increase in tardiness is insignificant to the performance especially to TraQIOS which still outperforms the other policies. For instance, at 0% inaccuracy, compared to SSTF, TraQIOS reduces tardiness by 36% compared to 29% reduction at 50% inaccuracy.

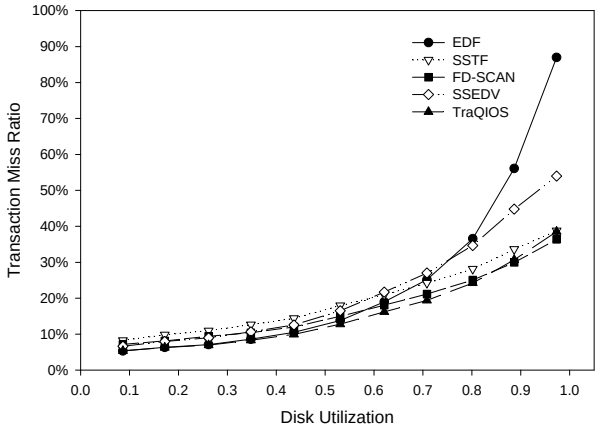

Figure 16: Performance under the Miss Ratio metric

The negligible degradation in performance though of the high inaccuracy in estimations is due to several reasons including the distribution of inaccuracies, and the impact of the other parameters involved in the priority function. However, the most important reason is the fact that for priority scheduling, it is the relative order of priority values that matters the most not the exact values. For example, in an extreme case where the actual number of I/Os is always higher than the estimated one, the relative order of transaction priorities will be the same using either the estimated or actual numbers. However, inaccuracies might result in a significant problem when it leads to priority reversals. For instance, when long transactions are missestimated to be short and vice versa.

In order to study such priority reversal extreme scenario and to stress test our proposed TraQIOS policy, we repeated this experiment where we set the inaccuracy in estimation to be inversely correlated with the actual transaction length with a correlation factor of -0.5. Hence, there is a higher probability for a short transaction to be assigned positive error, leading to an estimated number of I/Os higher than the actual one. Similarly, long transactions are assigned negative errors with a higher probability than negative ones. Figure 15 shows the performance under such setting where the increase in tardiness is more significant with the increase in inaccuracy. However, under this highly unrealistic scenario, TraQIOS still achieves the best performance compared to the other policies since N_i is only one of several parameters involved in the priority function.

5.8 Transaction Miss Ratio

For the sake of completeness, in addition to the previously reported tardiness results, in Figure 16 we measure the miss ratio provided by the different studied policies under our default experimental settings. However, notice that under our performance model a transaction is still executed to completion even after it passes its deadline. This performance model is different from the one used in RTDBMS where transactions have hard deadlines and a transaction is dropped if it misses the deadline. This difference in performance measuring might lead to results different from those in the RTDBMS literature.

Figure 16 shows that TraQIOS provides a very acceptable performance under our miss ratio measure. For instance, as utilization approaches 1.0, it reduces miss ratio by 50% compared to EDF while it increases it by only 12% compared to FD-SCAN. This in-

crease in miss ratio is for the sake of providing a tardiness 60% less than that of FD-SCAN at the same utilization point of 1.0 (as shown in Figure 4.

6. RELATED WORK

Disk scheduling is known to be an NP-complete problem. Thus, several heuristics [13] have been proposed to approximate the scheduling problem where the objective is mainly to optimize throughput or per request latency. Examples of these policies include SCAN, Shortest Seek Time First (SSTF), and Shortest Positioning Time First (SPTF).

Besides optimizing disk throughput, scheduling disk access under hard deadlines has received considerable attention which resulted in extending the traditional Earliest Deadline First (EDF) policy for scheduling transactions to schedule I/O requests. Examples of such policies include FD-SCAN [1] and SCAN-EDF [21]. However, these policies try to maximize the number of I/O requests meeting their deadlines as opposed to tardiness. Moreover, they work at the request level where the deadline of each request has to be explicitly specified as opposed to our problem in this paper where the deadline is specified at the transaction level.

In addition to EDF, several other scheduling policies for transactions have been proposed that attempt to maximize success rate in the presence of hard deadlines and explicit transaction priorities in Real-time Database Management Systems (RTDBMS). For instance, the work in [2] proposes prioritizing transactions according to the Least Slack policy where I/O requests generated by a transaction inherit its assigned priority. Moreover, it proposes policies for scheduling lock access based on the transaction specified priority. Meanwhile, notice that in our model, we are assuming a traditional database system where there is no notion of transaction priority and priorities are only assigned at the I/O-level to requests which already hold locks. That is based on the assumption that all transactions are of the same importance and the goal is to schedule I/O requests to minimize tardiness as opposed to maximizing success rate.

Minimizing tardiness in the presence of soft deadlines has been addressed by several research efforts in various types of multi-user systems. For example, the theoretical problem of scheduling jobs with the objective of minimizing tardiness has been addressed in [28, 19] which propose a parametrized offline scheduling policy for solving the single machine tardiness problem. Minimizing tardiness has also been studied in database systems (e.g., [7, 5, 25, 24, 26, 10]). For example, the work in [25, 24] proposes to limit the number of concurrent transactions within the DBMS by using an external scheduling mechanism which dispatches transactions according to an EDF-like policy. Also, the work in [26, 10] proposes a CPU scheduling policy for Web and in-memory database systems where transactions are assigned priorities using a hybrid policy which combines the advantages of the EDF policy together with the Shortest Job First policy or High Density First policy depending on whether or not transactions are associated with weights or importance.

The work in [28, 19] motivated the parameter-free online SAAB policy for minimizing tardiness in a wireless data broadcasting environment [20] as well as our parameter-free online TraQIOS policy proposed in this paper. However, the problem of scheduling transaction I/O requests addressed by TraQIOS is particularly different from that of CPU and broadcast scheduling. Specifically, the priority assigned by TraQIOS accounts for the fact that the cost of each I/O request is dynamic and it depends on the sequence of I/O requests already served since that sequence determines the current head location. Moreover, [19] and [20] studied scheduling independent jobs, whereas under TraQIOS, I/O requests are generated by user transactions which created the need for our slack budgeting schemes so that to map the transaction global deadline to a per-request intermediate deadline.

The z/OS workload manager (WML) [18] also recognizes the need for optimizing I/O access so that to meet higher level SLAs. Specifically, the WML approach is based on monitoring performance and accordingly adjusting transactions priorities. Though of the simplicity of that approach, it tends to be more reactive rather than proactive where priorities are only adjusted at the end of a sampling cycle. Moreover, WML sets priority at the transaction level rather than at the request level, hence, all requests from the same transaction are assigned the same priority regardless of the accessed block location. As such, it misses opportunities for optimizing I/O access by interleaving requests from different transactions to minimize seek time as in by SSTF and TraQIOS.

The closest work to ours is SSEDV [8], which we consider as part of our evaluation. SSEDV appeared in the context of RTDBMS where it studied the scheduling of I/O operations in order to maximize transaction success rate. Meanwhile, in this paper, we argue that minimizing transaction tardiness (i.e., amount of deviation from deadline) is a more appropriate performance goal in online interactive systems.

7. CONCLUSIONS

We introduce *TraQIOS*, a novel SLA-aware I/O scheduling policy for database transactions. Traditional I/O scheduling policies optimize for per-request deadlines or disk seeks in a transaction-oblivious fashion. Traditional DBMS transaction scheduling policies, on the other hand, are oblivious to I/O optimizations. *TraQIOS* extends the impact of SLA-aware transaction scheduling policies to the I/O subsystem.

The key idea underlying TraQIOS is to use on-line, SLA-aware slack budgeting and advertisements for I/O requests in each transaction to guide dynamic adaptation between a policy that optimizes disk seeks and an earliest deadline first policy. Our adaptive I/O scheduling policy exploits available slack for a given transaction to accommodate more urgent requests, while improving disk seeks under high I/O utilization.

In our experiments, we evaluated the sensitivity of TraQIOS to different workload settings including utilization, deadlines, cache, disk characteristics, data access patterns, and inaccuracies in query optimizer estimates. In all the conducted experiments, TraQIOS significantly outperforms the existing I/O scheduling schemes.

Acknowledgments: We would like to thank the anonymous reviewers for their thoughtful and constructive comments. The first author is supported in part by the Ontario Ministry of Research and Innovation Postdoctoral Fellowship. This work is also partially supported by NSF under project AQSIOS (IIS-0534531) and Career award (IIS-0746696). Finally, partial support has also been provided by Early Researcher Award (ERA), Ontario Centers of Excellence (OCE), NSERC, IBM Research, IBM CAS, and Intel.

8. REFERENCES

[1] R. K. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: A performance evaluation. In *RTSS*, 1990.

[2] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *TODS*, 17(3):513–560, 1992.

[3] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] K. P. Brown, M. J. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. In *VLDB*, 1993.

[6] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *RTSS '95*.

[7] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. In *VLDB*, 1989.

[8] S. Chen, J. A. Stankovic, J. F. Kurose, and D. F. Towsley. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Real-Time Systems*, 3(3):307–336, 1991.

[9] A. Fekete, E. J. O'Neil, and P. E. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Record*, 33(3):12–14, 2004.

[10] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Adaptive scheduling of web transactions. In *ICDE*, 2009.

[11] C. Hall and P. Bonnet. Getting priorities straight: Improving linux support for database I/O. In *VLDB*, 2005.

[12] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *SOSP*, 2001.

[13] E. G. C. Jr. and M. Hofri. On the expected performance of scanning disks. *SIAM J. Comput.*, 11(1):60–70, 1982.

[14] B. Kao and H. Garcia-Molina. Subtask deadline assignment for complex distributed soft real-time tasks. In *ICDCS*, 1994.

[15] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Trans. Parallel Distrib. Syst.*, 8(12):1268–1274, 1997.

[16] V. C. S. Lee, K. yiu Lam, B. Kao, K.-W. Lam, and S. lun Hung. Priority assignment for sub-transaction in distributed real-time databases. In *RTDB*, 1996.

[17] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *ICDE*, 2004.

[18] A. S. Meritt, J. A. Staubi, K. M. Trowell, G. S. Whistance, and H. M. Yudenfriend. z/OS support of the IBM TotalStorage enterprise storage server. *IBM Systems Journal*, 42(2):280–301, 2003.

[19] P. S. Ow and T. E. Morton. The single machine early/tardy problem. *Manage. Sci.*, 35(2):177–191, 1989.

[20] A.-D. Popescu, M. A. Sharaf, and C. Amza. SLA-aware adaptive on-demand data broadcasting in wireless environments. In *MDM*, 2009.

[21] A. L. N. Reddy, J. C. Wyllie, and R. Wijayaratne. Disk scheduling in a multimedia I/O system. *TOMCCAP*, 1(1):37–59, 2005.

[22] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[23] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Inter. Tech.*, 2006.

[24] B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. Nahum. Achieving class-based QoS for transactional workloads. In *ICDE*, 2006.

[25] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to determine a good multi-programming level for external scheduling. In *ICDE*, 2006.

[26] M. A. Sharaf, S. Guirguis, A. Labrinidis, K. Pruhs, and P. K. Chrysanthis. ASETS: A self-managing transaction scheduler. In *ICDE Workshops*, 2008.

[27] R. M. Sivasankaran, J. A. Stankovic, D. F. Towsley, B. Purimetla, and K. Ramamritham. Priority assignment in real-time active databases. *VLDB J.*, 5(1):19–34, 1996.

[28] A. P. J. Vepsalanen and T. E. Morton. Priority rules for job shops with weighted tardiness costs. *Manage. Sci.*, 33(8):1035–1047, 1987.

APPENDIX

A. COMPUTING I/O REQUEST PRIORITY

In order to ensure that a global schedule is optimal, every local schedule has to be optimal as well so that no improvement is achieved when exchanging the order of two tasks in the schedule. Hence, assume two I/O requests R_1 and R_2 with costs C_1 and C_2 respectively. Further, assume that S_1 is the current slack of R_1 and S_2 is the current slack of R_2. Finally, assume that time required to move the head from the block requested by R_1 to the one requested by R_2 is L.

In a schedule X, where R_1 is followed by R_2, the total tardiness (E_X) is computed as follows:

$$E_X = max(0, -S_1) + max(0, C_1 + L - C_2 - S_2)$$

Similarly, for an alternative schedule Y, where R_2 is followed by R_1, the total tardiness (E_Y) is computed as follows:

$$E_Y = max(0, -S_2) + max(0, C_2 + L - C_1 - S_1)$$

For E_X to be less than E_Y, then:

$$\frac{1}{C_1 + L - C_2}\left(\frac{-max(0, -S_1)}{C_2 + L - C_1} + max(0, 1 - \frac{S_1}{C_2 + L - C_1})\right)$$

has to be greater than

$$\frac{1}{C_2 + L - C_1}\left(\frac{-max(0, -S_2)}{C_1 + L - C_2} + max(0, 1 - \frac{S_2}{C_1 + L - C_2})\right)$$

where the first term represents the priority assigned to R_1 and the second term is the priority assigned to R_2.

In general, in order to compare R_i to any arbitrary request, we substitute L and C_2 with the expected seek time $\overline{C}$. Hence, the priority of R_i is computed as:

$$\frac{1}{C_i}\left(\frac{-max(0, -S_i)}{2\overline{C} - C_i} + max(0, 1 - \frac{S_i}{2\overline{C} - C_i})\right)$$

or equivalently:

$$P_i = \begin{cases} \frac{1}{C_i} & S_i \leq 0 \\ \frac{1}{C_i}(1 - \frac{S_i}{2\overline{C} - C_i}) & 0 < S_i \leq 2\overline{C} - C_i \\ 0 & S_i > 2\overline{C} - C_i \end{cases} \quad (6)$$