Adaptive Scheduling of Web Transactions

Shenoda Guirguis 1 Mohamed A. Sharaf 2 Panos K. Chrysanthis 1 Alexandros Labrinidis 1 Kirk Pruhs 1

1CS Department, University of Pittsburgh
{shenoda, panos, labrinid, kirk}@cs.pitt.edu
2ECE Department, University of Toronto
msharaf@eecg.toronto.edu

Abstract—In highly interactive dynamic web database systems, user satisfaction determines their success. In such systems, user-requested web pages are dynamically created by executing a number of database queries or web transactions. In this paper, we model the interrelated transactions generating a web page as workflows and quantify the user satisfaction by associating dynamic web pages with soft-deadlines. Further, we model the importance of transactions in generating a page by associating different weights to transactions. Using this framework, system success is measured in terms of minimizing the deviation from the deadline (i.e., tardiness) and also minimizing the weighted such deviation (i.e., weighted tardiness).

In order to efficiently support the materialization of dynamic web pages, we propose ASETS*, which is a parameter-free adaptive scheduling algorithm that automatically adapts to, not only system load, but also transactions’ characteristics (i.e., interdependencies, deadlines and weights). ASETS* prioritizes the execution of transactions with the objective of minimizing weighted tardiness. It is also capable of balancing the trade-off between optimizing average- and worst-case performance when needed. The performance advantages of ASETS* are experimentally demonstrated.

I. INTRODUCTION

Web-database systems nowadays support the most prevailing e-services ranging from e-banking and stock trading, to e-commerce applications, to personalized news and weather services. In these applications, user-requested web pages are dynamically created from data in databases. Specifically, dynamic web pages are composed by a number of content fragments which define both the layout of the page as well as its content. A dynamic web page is generated by dynamically materializing each individual fragment by accessing local and remote databases and by executing lengthy code to produce HTML. Often, the content of the fragments composing a dynamic web page is interdependent, and that leads to dependencies among the web transactions which materialize the corresponding fragments. Moreover, different fragments might have different importance in generating a page.

In such highly interactive applications, user satisfaction or positive experience determines their success. Reportedly, more than 20 billion dollars in revenue are lost every year due to excessive delays in e-commerce web pages that lead clients to quit their sessions without completing a purchase [7]. Given the bursty and unpredictable behavior of web user populations, it is therefore crucial for such systems to adapt and scale automatically and efficiently to different workload’s settings, prioritizing resources as needed in order to keep users satisfied under varying workloads.

One way to quantify a user’s satisfaction is to associate a dynamic web page with a soft-deadline which defines an upper bound on the latency perceived by the end user accessing that page. This can be extended to the fragment-level where each content fragment in a dynamic web page is assigned its own deadline. In either case, the assigned deadline is a mapping from the service level agreements (SLAs) provided by the dynamic content service provider to the end user. Hence, the success of the system (i.e., the user satisfaction) is better measured in terms of minimizing the deviation from the deadline, that is, tardiness.

Unfortunately, minimizing tardiness is not a trivial goal, especially under high loads or strict deadlines. This goal is further complicated in the presence of dependencies between different fragments in dynamic web pages. Moreover, a fragment is often associated with some utility or weight which represents its importance in generating a page. The presence of these weights further adds to the complexity of the problem where the goal should be further extended to minimize the weighted tardiness. That is, transactions materializing more important fragments should experience less tardiness than those materializing less important ones.

In order for the service provider to meet its expected goals, it often employs a transaction scheduler which prioritizes the execution order of web transactions involved in the generation of dynamic web pages and their fragments. Toward this, several off-the-shelf policies have been used for transaction scheduling. However, these policies are either deadline-oblivious, or dependency-oblivious, or both. For example, the Earliest-Deadline-First (EDF) policy is often used for scheduling transactions according to their deadlines. However, EDF minimizes tardiness only if the system is lightly loaded and when precedence constraints between transactions are consistent with the transaction deadlines. This means that a dependent transaction cannot have a deadline which is earlier than the deadline of any transaction that precedes it. Unfortunately, this is not always the case since the precedence relationship between transaction does not necessarily lead to precedence in the associated deadlines.

Shortest-Remaining-Processing-Time (SRPT) is another policy which is often used for scheduling web transactions. Although SRPT is known to outperform EDF under high loads,
it performs far worse at light loads. Further, it is oblivious to dependency and precedence constraints between transactions.

In this paper, we model the dependency between transactions generating a web page as a set of workflows and quantify the user satisfaction by associating dynamic web pages with soft-deadlines. Further, we model importance of transactions in generating a page by associating different weights to transactions. Based on this model, we develop and experimentally demonstrate the performance advantages of a parameter-free adaptive scheduling policy, called ASETS*, that adapts to system load. ASETS* extends the ASETS policy [12] by exploiting the dependency between web transactions in order to minimize the perceived tardiness. It also leverages the weight assigned to each transaction so that to maximize the user satisfaction by minimizing weighted tardiness.

Specifically, ASETS* employs a novel adaptive policy that integrates EDF with the Highest Density First (HDF) policy, which is optimal for weighted transactions [2]. In the case where all weights are equal, HDF reduces to SRPT and ASETS* reduces to an integration of EDF and SRPT. In the absence of precedence constraints, ASETS* operates at transaction-level, while it operates at the workflow-level when they exist. This allows ASETS* to dynamically adjust to the workload and constantly minimize the perceived tardiness. In that sense, ASETS* is a parameter-free adaptive policy that adapts, not only to the system load, but also to the transaction characteristics, and decides at which level to operate: i.e., transaction-level or workflow-level.

Finally, ASETS* is also capable of balancing the trade-off between optimizing average-case and worst-case performance when needed by utilizing an aging scheme that recognizes deadlines.

Road-map: The rest of the paper is organized as follows: Section II provides important background. The ASETS algorithm and its extensions are motivated and explained in Section III, and evaluated in Section IV. Section V discusses related work. We conclude in Section VI.

II. BACKGROUND

A. System Model

Typically, dynamic web pages are composed of a number of content fragments which define both the layout of the page as well as its content. The content of each fragment is materialized on the fly by dynamically executing a number of transactions on local and remote databases. Often, the contents of the fragments composing a dynamic web page are interdependent which in turn leads to dependencies among the corresponding web transactions. These dependencies are expressed in terms of transaction workflows which specify the relationship between the different transactions involved in creating a web page as well as a partial order of transaction execution. Specifically, in a workflow, if the output of transaction $T_x$ is an input to transaction $T_y$ (i.e., $T_x \rightarrow T_y$), then $T_y$ is a dependent transaction where $T_y$ depends on $T_x$, or equivalently, $T_x$ precedes $T_y$.

Clearly, the dependency property is transitive such that if $T_x \rightarrow T_y$ and $T_y \rightarrow T_z$, then $T_z$ depends on $T_x$ (i.e., $T_x \rightarrow T_z$). In general, a dependent transaction $T_i$ might depend on a set of one or more transactions. We call that set of transactions the dependency list of $T_i$, and it is denoted as $l_i$. If $l_i$ is the empty set (i.e., $l_i = \phi$), then $T_i$ is an independent transaction.

An independent transaction is ready to be executed whenever it is submitted to the back-end database. On the other hand, a dependent transaction is only ready for execution after all the transactions in its dependency list are executed first.

In the general case, generating the dynamic content of a single fragment $G_i$ requires the execution of a set of dependent and independent transactions. In this paper, for brevity and without loss of generality, we assume that a fragment $G_i$ is generated by a single transaction $T_i$. That is, we view that the set of transactions materializing a fragment as a single (long) transaction $T_i$ which performs all the tasks of the original set of transactions and inherits all the dependencies of the original transactions on transactions for other fragments. We also assume that there is a single backend database from which all fragments are generated.

Transaction $T_i$ is partially characterized by a soft-deadline $d_i$, which is the pre-specified SLA of the corresponding fragment $G_i$. In general, a transaction $T_i$ is characterized by the following parameters:

**Definition 1:** We define the characteristics of a transaction $T_i$ to be:
- **Arrival Time** ($a_i$): The time when $T_i$ has arrived at the database system.
- **Deadline** ($d_i$): The ideal time by which $T_i$ should finish execution.
- **Length** ($r_i$): The (remaining) processing time needed to execute $T_i$. We assume that if caching or materialization is utilized for fragments [8], then transactions’ lengths are adjusted accordingly.
- **Weight** ($w_i$): The weight associated with $T_i$, to reflect its importance.
- **Dependency List** ($l_i$): The list of transactions that precede $T_i$. We assume this information is available to the scheduler [10].

Given these parameters, at any given point of time, the slack of a transaction $T_i$ is defined as follows:

**Definition 2:** The slack $s_i$ of a transaction $T_i$ is the extra amount of time $T_i$ can wait before it has to execute in order to meet the deadline $d_i$. Specifically, at any given time $t$, $s_i = d_i - (t + r_i)$.

Given a set of interdependent transactions, a workflow is defined with respect to the dependency lists. Specifically, a workflow is defined for every transaction that does not appear in any dependency list. The workflow for transaction $T_i$ includes all transactions that appear in $l_i$, and recursively all transactions that appear in $l_j$ of each $T_j \in l_i$. Note that a transaction can belong to more than one workflow.
The system model is illustrated in Figure 1. The upper part of the figure shows the workflow-level, whereas the lower part of the figure shows the transaction-level. At the workflow-level, arrows represent the dependency or precedence constraints between transactions. At the transaction-level, we illustrate the per-transaction properties listed above.

In Figure 1, we illustrate a web page with two workflows: \(< T_1, T_m, T_n, T_o >\) and \(< T_1, T_i, T_j, T_k >\). Each workflow has at least one leaf transaction and a single root transaction. For instance, in workflow \(< T_1, T_m, T_n, T_o >\), \(T_1\) is the root transaction and \(T_o\) is the corresponding fragment.

In a workflow, a leaf is an independent transaction, whereas the root is a dependent transaction. However, the root transaction does not precede any other transaction in the network of workflows, i.e., a root transaction does not belong to any dependency list. Each transaction \(T_i\) in a workflow, the arrival time \(a_i\) and deadline \(d_i\) are available to the system once the transaction \(T_i\) is submitted. The length of the transaction \(r_i\) is typically computed by the system based on previous statistics and profiles of transaction execution.

B. Application Scenario

To illustrate the above concepts, consider a web application which provides users with web pages that are tailored to their interests and preferences. For instance, it provides users with information about the stock market, traffic conditions, weather conditions, etc. Further, assume that a user’s page contains four fragments for stock market information which are as follows: fragment 1: lists the prices for all stocks traded in the stock market, fragment 2: lists the prices of all stocks in the user’s portfolio, fragment 3: provides the current total value of the user’s portfolio, and fragment 4: lists alerts on stocks that meet certain conditions predefined by the user (e.g., the price of a certain stock has changes by more than 5%).

Clearly, the content of each of the stock fragments is dynamic. This requires running certain transactions against the backend database so that to retrieve the most current data needed for populating those fragments. Let’s assume these transactions \(T_1, T_2, T_3,\) and \(T_4\) where transaction \(T_i\) populates the corresponding fragment \(G_i\). These transactions clearly exhibit some dependency and form a workflow. In particular, \(T_2\) is dependent on \(T_1\) where \(T_1\) retrieves the current list of stock prices and \(T_2\) joins that list with the list of stocks in the user’s portfolio. Similarly, \(T_3\) is dependent on \(T_2\) where \(T_3\) runs some aggregate query on the output of \(T_2\). Similarly, \(T_4\) is dependent on \(T_2\) where \(T_4\) applies some predicates to filter the output of \(T_2\).

In the scenario above, all transactions are submitted to the database as the user logs onto the system. Moreover, each transaction is associated with an SLA which reflects its urgency. However, notice that the precedence relationship between transaction does not necessarily imply a precedence in the associated deadlines. This conflict between precedence constraints and deadline constraints is easily illustrated by considering the relationship between \(T_4\) and both \(T_1\) and \(T_2\). While \(T_4\) (i.e., alerts) is dependent on both \(T_1\) and \(T_2\), \(T_4\) might in fact have an earlier deadline than both since a user would most probably like to see the stock alerts first. Similarly, the same conflict might also arise between \(T_3\) (i.e., portfolio value) and both \(T_1\) and \(T_2\).

In addition to deadlines, each transaction \(T_i\) is also associated with a weight \(w_i\). In our example here, this weight can reflect the subscription level of the user, for example: gold, silver, or bronze, corresponding to how much money they paid. It might also reflect the importance of different transactions from the perspective of a single user. For instance, the stock workflow might be more important to the user than the traffic workflow, or similarly traffic might be more important than weather conditions.

For this particular web application, as well as any other service provider application, user satisfaction determines their success and thus the goal is to optimize the performance for user’s satisfaction. Transaction scheduling is one technique for achieving that aforementioned goal. However, an efficient transaction scheduler should consider both: 1) the transaction properties, and 2) the relationship between different transactions. In this paper, we propose such scheduling policy. However, before delve into the details of our proposed policy, we will first provide more insights into the desired performance goals in a database-driven web environment.

C. Performance Goals

Ideally, the finish time \(f_i\) of transaction \(T_i\) should be equal to the sum of its arrival time \(a_i\) and its length \(r_i\). However, this will only happen if transaction \(T_i\) does not experience any queuing delays or if it is the only transaction in the system, which is not the norm; a transaction will typically wait for other transactions to finish execution first, especially when the system is under high load.

In the soft-deadline model, the system strives to finish executing each transaction \(T_i\) before its deadline, \(d_i\). However, if \(T_i\) cannot meet its deadline, the system will still execute \(T_i\), but it will be “penalized” for the delay beyond the deadline \(d_i\). This delay is known as tardiness and is defined as follows:
Definition 3: Transaction tardiness, $t_i$, for transaction $T_i$ is the total amount of time spent by $T_i$ in the system beyond its deadline $d_i$. That is, $t_i = 0$ iff $f_i \leq d_i$, and $t_i = f_i - d_i$ otherwise.

Similarly, the overall system performance is measured in terms of average tardiness which is defined as follows:

Definition 4: The average tardiness for $N$ transactions is:

$$\frac{1}{N} \sum_{i=1}^{N} t_i$$

In the case where transactions are associated with weights, the system performance is naturally measured using weighted tardiness which is defined as:

Definition 5: The average weighted tardiness for $N$ transactions is: $\frac{1}{N} \sum_{i=1}^{N} (t_i w_i)$.

Web-databases typically employ a transaction scheduler which decides the execution order of transactions. One common and natural class of scheduling policies are called priority-based policies. In a priority-based policy, priority $P_i$ is assigned to each transaction $T_i$, and the highest priority transaction is always executed first. Different schedulers consider different parameters for computing the priority $P_i$. For example, $EDF$ uses $1/d_i$ as the priority, while $HDF$ uses $w_i/d_i$ as the priority. In the next sections, we describe some well-known policies for transaction scheduling, as well as our proposed $ASETS^*$ policy.

III. $ASETS^*$

In this section, we introduce $ASETS^*$, our proposed scheduling algorithm. For clarity of presentation, we first describe $ASETS^*$ for scheduling independent transactions in Section III-A. Next, we extend $ASETS^*$ to schedule dependent transactions with precedence constraints in Section III-B and present the general case of $ASETS^*$ for scheduling dependent transactions with associated weights in Section III-C. Finally, we show how $ASETS^*$ can balance the trade off between average- and worst-case performance in Section III-D.

A. Scheduling at the Transaction-Level

Before introducing $ASETS^*$, we illustrate the trade off between $EDF$ and $SRPT$ that motivated our work.

1) EDF vs SRPT: Earliest Deadline First ($EDF$), and Shortest Remaining Processing Time ($SRPT$) are two promising policies that have been proposed for minimizing average tardiness. Specifically, under $EDF$, a transaction with an early deadline receives a higher priority, whereas under $SRPT$, a transaction with a shorter processing time is the one which receives higher priority.

$EDF$ guarantees that all jobs will meet their deadlines if the system is not over-utilized. As such, the tardiness of the system is expected to be zero since all the transactions meet their deadlines. When the system is over-utilized, it is impossible to finish all transactions by the specified deadlines. So some transactions will experience tardiness. 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 where 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 contrast to $EDF$, $SRPT$ is the best policy to use when all transactions have already missed their deadlines. This is because the problem of minimizing tardiness in this case is the same as the problem of minimizing response time, for which $SRPT$ has been shown to be the optimal policy [11]. However, in the case when there are transactions that have not missed their deadline yet, $SRPT$ might run into the problem of assigning a high priority to a short transaction that has a long deadline instead of scheduling another one which is relatively longer but its deadline is imminent.

Example 1: To further illustrate the difference between the two policies, consider the example in Figure 2. The figure shows two sets of transactions $T_1$ and $T_2$ with deadlines $d_1$ and $d_2$, and processing times $r_1$ and $r_2$, respectively.

In Figure 2(a), the tardiness of running the transactions using the $EDF$ policy ($=t_2$) is less than that of using the $SRPT$ policy ($=t_1$). The reason for $SRPT$’s higher tardiness is giving higher priority to $T_2$ which has the shorter processing time ($r_2$) but a longer deadline ($d_2$). For the other set of transactions in Figure 2(b), $EDF$ provides higher tardiness ($=t_1 + t_2$). This is because it scheduled $T_1$ first which is already past its deadline leading to missing $T_2$’s deadline as well.

As it is obvious from this example, there is no clear best policy for scheduling transactions with deadlines that minimizes tardiness under all workloads. Generally speaking, $EDF$ performs well at low utilization, whereas at high utilization, $SRPT$ performs better than $EDF$.

One possibility is to select the policy dynamically based on the load of the system. However, measuring the load with reasonable accuracy may require non-trivial resources. More importantly, when jobs have deadlines, measuring the load
does not only involve considering the processing requirements of the transactions, but also the relationships between processing times and deadlines. For example, a batch of transactions with very low processing requirements but very tight deadlines will lead to an overloaded condition.

2) The ASETS* Policy: We propose a hybrid policy for transaction scheduling called Adaptive SRPT EDF Transaction Scheduling (ASETS) [12]. ASETS is the core of ASETS* which is a parameter-free adaptive policy that integrates the advantages of both the SRPT and EDF policies and automatically adapts to system load.

Under ASETS*, the scheduler maintains two priority lists. In the first list, called EDF-List, transactions are ordered according to their deadlines as in the EDF scheduling policy. That is the priority of each transaction in EDF-List is \( p_i = 1/d_i \). In the second list, called SRPT-List, transactions are ordered according to their remaining processing time as in the SRPT scheduling policy. That is the priority of each transaction in SRPT-List is \( p_i = 1/r_i \).

The first list, EDF-List, contains all transactions that can still make their deadlines, if they start execution right now.

Definition 6: A transaction \( T_i \) with deadline \( d_i \) is included in EDF-List iff, \( t + r_i \leq d_i \), where \( t \) is the current time.

The second list, SRPT-List, contains all transactions that already missed their deadlines.

Definition 7: A transaction \( T_i \) with deadline \( d_i \) is included in SRPT-List iff, \( t + r_i > d_i \), where \( t \) is the current time.

Notice that each transaction starts in the EDF-List then it might move to the SRPT-List if it misses its deadline while waiting in the EDF-List. Given the above two lists, at each scheduling point ASETS* selects for execution either the transaction at the top of EDF-List or the one at the top of SRPT-List. For convenience, we will call these two transactions: \( T_{i, EDF} \) and \( T_{i, SRPT} \), respectively.

To decide between \( T_{i, EDF} \) and \( T_{i, SRPT} \), given their remaining processing and slack times, ASETS* utilizes a simple greedy heuristic under, scheduling \( T_{i, EDF} \) for execution if:

\[
r_{i, EDF} < r_{i, SRPT} - s_{i, EDF},
\]

otherwise, \( T_{i, SRPT} \) is the one scheduled for execution.

The premise underlying this heuristic is to schedule the transaction with the least “negative” impact on the total tardiness. In particular, if \( T_{i, EDF} \) is scheduled first, its negative impact is to increase \( T_{i, SRPT} \)’s tardiness by \( r_{i, EDF} \). On the other hand, if \( T_{i, SRPT} \) is scheduled first, its negative impact is to increase \( T_{i, EDF} \)’s tardiness by \( r_{i, SRPT} \) minus the amount of slack \( s_{i, EDF} \) that \( T_{i, EDF} \) currently has, as illustrated in Figure 3.

To make it clearer, if the system has only these two transactions \((T_{i, SRPT}, T_{i, EDF})\), whichever order will lead to a minimal tardiness is the order that ASETS* follows. In other words, the top transaction on SRPT-List \( (T_{i, SRPT}) \) will be selected if the transaction on top of EDF-List \( (T_{i, EDF}) \) can still meet the deadline if it ran right after \( T_{i, SRPT} \). Otherwise, the top transaction on EDF-List will be selected to run first. Next we provide two examples of \( T_{i, EDF} \) and \( T_{i, SRPT} \) transactions, where \( T_{i, SRPT} \) is selected to run first in one example, while \( T_{i, EDF} \) is selected to run first in the second example.

Example 2: Figure 4 illustrates an example where the negative impact of running \( T_{i, SRPT} \) is less than the negative impact of running \( T_{i, EDF} \) first. The parameters of \( T_{i, SRPT} \) are: remaining processing time \( r_{1, SRPT} = 3 \) and deadline \( d_{1, SRPT} = 1 \). The parameters of \( T_{i, EDF} \) are: remaining processing time \( r_{1, EDF} = 2 \), deadline \( d_{1, EDF} = 2 \), and slack \( s_{1, EDF} = 2 \).

In Figure 4(c), clearly, if \( T_{i, SRPT} \) runs now, it will be tardy. So, the negative impact of running \( T_{i, EDF} \) before \( T_{i, SRPT} \) is to increase the tardiness by at least \( r_{i, EDF} = 5 \). On the other hand, running \( T_{i, SRPT} \) first as shown in Figure 4(d),
In order to accommodate dependency between transactions in a workflow, a simple yet naive way to extend $ASETS^*$ is to add a third Wait queue in addition to the $EDF$-List and the $SRPT$-List. A transaction $T_i$ is added to the Wait queue if it is still waiting for the execution of any the transactions that precede it (i.e., $l_i \neq \phi$). The rest of the transactions that are ready to execute are placed normally either in the $EDF$-List or $SRPT$-List. Under this approach, which we call $Ready$, a transaction $T_i$ would move from the Wait queue to the appropriate queue, i.e., $EDF$-List or $SRPT$-List, once all the transactions in its dependency list $l_i$ finish execution.

Under the $Ready$ approach, the scheduler is oblivious to any information on the dependent transactions which are concealed in the Wait queue. This leads to partially-informed scheduling decisions which are merely based on the transactions in both the $EDF$-List and $SRPT$-List. However, the Wait queue might contain a valuable transaction with an urgent deadline and/or high utility which ideally should be leveraged to boost the priority of those transactions which precede it in the workflow.

Toward a well-informed scheduling decision in the presence of dependency, transactions in the $EDF$-List and $SRPT$-List should inherit the most valuable characteristics of transactions in their respective workflows. To achieve this, we extend $ASETS^*$ so that it considers workflows rather than transactions. Specifically, at each scheduling point, for a workflow $K_A$, we make the distinction between the following two special transactions:

Definition 8: The Head Transaction $(T_{head,A})$: is a transaction that belongs to workflow $K_A$ and is ready for execution (i.e., $l_{head,A} = \emptyset$).

In other words, the head transaction is the first transaction in the workflow which is ready for execution. This is either because it initially had an empty dependency list or because all the transactions on its dependency list have been executed. The head transaction of the workflow changes over time.

Definition 9: The Representative Transaction $(T_{rep,A})$: is a virtual transaction which captures the properties of the remaining transactions in workflow $K_A$.

A representative transaction $T_{rep,A}$ is characterized by the following parameters:

- Deadline $(d_{rep,A})$: The minimum (earliest) deadline among all the remaining transactions in $K_A$.
- Remaining Processing Time $(r_{rep,A})$: The minimum remaining processing time among all the pending transactions in $K_A$.
- Weight $(w_{rep,A})$: The maximum weight among all the remaining transactions in $K_A$.

The representative transaction allows $ASETS^*$ to see beyond the $EDF$-List and $SRPT$-List into the Wait queue. This allows $ASETS^*$ to recognize any valuable dependent transactions and in turn adjust the priority of its corresponding head transaction. In particular, $ASETS^*$ decides if a workflow $K_A$ should be placed in the $EDF$-List or the $SRPT$-List according
to its corresponding representative transaction \( T_{rep,A} \), where a workflow \( K_A \) is placed in \( EDF-List \) iff \( T_{rep,A} \) can still meet the deadline if it starts execution now. Otherwise, it is placed in the \( SRPT-List \). Formally, a workflow \( K_A \) is placed in the \( EDF-List \) iff \( t + r_{rep,A} \leq d_{rep,A} \), where \( t \) is the current time, \( r_{rep,A} \) is the processing time of the representative transaction of workflow \( K_A \), and \( d_{rep,A} \) is its deadline. Otherwise, \( K_A \) is inserted in the \( SRPT-List \). Moreover, the \( EDF-List \) and \( SRPT-List \) are sorted based on the representative deadline \( d_{rep,A} \) and the representative processing time \( r_{rep,A} \), respectively.

In order to decide which transaction to run, we consider the negative impact of the head transaction of the workflow at the top of \( EDF-List \) (say \( K_A \)) and that of the workflow at the top of \( SRPT-List \) (say \( K_B \)).

In general, the negative impact of running workflow \( K_A \) on workflow \( K_B \) is calculated as the negative impact of running the head transaction of \( K_A \) (\( T_{head,A} \)) on the representative transaction of \( K_B \) (\( T_{rep,B} \)). The intuition behind that is that the representative transaction represents the most important transactions in a workflow, while the transaction that will actually get to run is the head transaction. Clearly running the head transaction of workflow \( K_A \) has negative impact on all transactions of workflow \( K_B \). However, the representative transaction, could be used to represent (or estimate) the maximum negative impact on the workflow. More precisely, the negative impact of running workflow \( K_A \) before workflow \( K_B \) is to increase the tardiness of the representative transaction of \( B \) (i.e., \( T_{rep,B} \)) by \( r_{head,A} \) minus whatever slack \( T_{rep,B} \) has. Thus, workflow \( K_A \) runs first iff \( r_{head,A} - s_{rep,B} \leq r_{head,B} - s_{rep,A} \).

**Example 4:** Let \( K_A \) and \( K_B \) be the two winner workflows of \( EDF-List \) and \( SRPT-List \), respectively, as illustrated in Figure 6. Both \( K_A \) and \( K_B \) consist of two transactions. The figure shows the deadline and the remaining processing time values of each transaction. As shown in the figure, the head transactions of \( K_A \) and \( K_B \) are \( T_{head,A} \) and \( T_{head,B} \), respectively. Also the representative transactions of \( A \) and \( B \) are \( T_{rep,A} \) and \( T_{rep,B} \), respectively. Thus, the negative impact of running \( K_A \) first is calculated as: \( r_{head,A} - s_{rep,B} = 2 - 2 = 0 \). That is, the processing time of \( T_{head,A} \) minus the slack of \( T_{rep,B} \). On the other hand, the negative impact of running \( K_B \) first is calculated as: \( r_{head,B} - s_{rep,A} = 3 - 0 = 3 \). Thus, \( K_A \) gets to run first, which means that \( T_{head,A} \) is to run until it finishes execution, or a new transaction arrives in the system.

**C. ASETS\(^*\): The General Case**

In this section, we generalize the ASETS\(^*\) policy to handle the general case where transactions are assigned independent weights.

As discussed in Section II-B, some transactions might be more important than others from the users’ perspective. Thus, when transactions are assigned different weights, the right performance metric becomes the average weighted tardiness and the objective is then to minimize the average weighted tardiness, as defined in Definition 5.

Recall that ASETS\(^*\) is essentially a hybrid policy between \( EDF \) and \( SRPT \) policies. However, this is the case when all transactions are equally weighted. The first question then is: what is the natural extension of \( EDF \) and \( SRPT \) whenever transactions are assigned different weights? In the extreme case under high system utilization, when all transactions have already missed their deadlines, the Highest Density First (HDF) policy is known to be optimal [2]. HDF assigns a
priority to each transaction that equals to its weight divided by its remaining processing time, that is $p_i = w_i/r_i$. Given this priority assignment, if all weights are equal, $HDF$ reduces to $SRPT$. On the other hand, if all transactions can still meet the deadline, $EDF$ is still the optimal policy. Because there is no tardiness at all, the weight does not play any role, since the final average weighted tardiness is zero.

Thus, in the general case, $ASETS^*$ is a hybrid policy that integrates $EDF$ and $HDF$. It reduces to an integration of $EDF$ and $SRPT$ in the case where all weights are equal. In that sense, $ASETS^*$ is a parameter-free adaptive policy that adapts, not only to the system load, but also to the transaction characteristics, and decides at which level to operate: i.e., transaction-level or workflow-level.

The general $ASETS^*$ policy employs two lists: the $EDF$-List and the $HDF$-List (which reduces to $SRPT$-List in case all weights are equal). The representative transaction is used as described in Section III-B to determine whether a workflow belongs to the $EDF$-List or to the $HDF$-List.

In the general $ASETS^*$ policy, the heuristic for deciding which winner to run is modified to reflect the fact that transactions are of different weights. Specifically, we need to scale the magnitude of the negative impact incurred by a workflow $K_A$ by the weight of that workflow (i.e., $w_{rep,A}$). Thus, we calculate the negative impact as before, then multiply it by the weight of the workflow, where the workflow inherits the maximum weight of its transactions. The algorithm pseudo-code is illustrated in Figure 7. In the Figure, the $Head()$ function takes a workflow reference as input and returns the Head transaction of that workflow, while $Representative()$ function takes a workflow reference as input and returns the representative transaction.

D. Balancing the Trade-off between Average- and Worst-case Performance

Some applications might require the scheduling policy to balance the trade-off between average- and worst-case performance. For such applications, $ASETS^*$ can be easily modified to achieve that required balance. In particular, $SRPT$ suffers from starvation. Starvation can be handled using an aging scheme that schedules the longest transaction after some time. However, in our case, there is a natural aging scheme captured by the missed deadline. That is, the oldest transaction is the one that has the earliest deadline. Hence, our simple balance-aware $ASETS^*$ would periodically run the transaction with the highest weight to deadline ratio which we call $T_{old}$. By running $T_{old}$, probably earlier than when it is scheduled to run according to $ASETS^*$, we minimize the starvation of high utility transactions and in turn improve the worst-case performance. However, this is expected to come at the expense of an increase in the overall average weighted tardiness (i.e., average-case performance). To balance the trade-off between the average- and worst-case performance, the frequency of selecting and running $T_{old}$ is controlled via an activation rate parameter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_i$</td>
<td>transaction length</td>
<td>Zipf($\alpha$) over [1 - 50]</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>skewness of job length distribution</td>
<td>0.5</td>
</tr>
<tr>
<td>$K_A$</td>
<td>slack factor</td>
<td>$[0.0 - \kappa_{max}]$</td>
</tr>
<tr>
<td>$\alpha_t$</td>
<td>arrival time</td>
<td>Poisson process with arrival rate = $2\cdot systemUtilization/\text{avg_transaction_length}$</td>
</tr>
<tr>
<td>SystemUtilization</td>
<td></td>
<td>$[0.1 - 1.0]$</td>
</tr>
<tr>
<td>Weight</td>
<td></td>
<td>$[1 - 10]$</td>
</tr>
</tbody>
</table>

TABLE I
SUMMARY OF EXPERIMENTAL PARAMETERS

In this paper, we distinguish two possible types of activation rates: time-based and count-based. Using the time-based activation rate means that every $P^t$ time units, a $T_{old}$ transaction is selected and executed. While using the count-based period means that a $T_{old}$ transaction runs every $P^c$ scheduling points. In Section IV-F we study how this balance-aware $ASETS^*$ performs for different values of the activation rate parameter.

IV. PERFORMANCE EVALUATION

We have conducted multiple experiments to evaluate the performance of our proposed scheduling policies. We describe the settings for these experiments in Section IV-A and the experimental results in Section IV-B.

A. Experimental Setup

Testbed: We created an RTDBMS simulator using C++. The simulator takes as input the system parameters, and generates the workload based on these parameters. The workload is a set of transaction properties; i.e., processing requirements, deadlines, dependencies, etc. We conducted several experiments to evaluate the performance of our proposed policies and compare them to other policies that were all implemented in our developed simulator.

Policies: At transaction level, we compared the $ASETS^*$ policy against the previously described $EDF$ and $SRPT$ policies. For completeness, we have also compared it against the traditional First Come First Served (FCFS) and the Least Slack (LS) policy [1], where under LS, the priority of transaction $T_i$ is set to $1/s_i$. At the workflow level, when all weights are equal, we compared $ASETS^*$ against Ready described in Section III-B.

As for $ASETS^*$ in the general case, i.e., at the workflow level when weights are different, we compared $ASETS^*$ against $EDF$ and $HDF$. We finally demonstrate the worst- and average-case performance of the balance-aware $ASETS^*$, which balances the trade-off between the worst- and average-case performance.

Transactions/Queries: We created a transaction workload similar to those in [5], [1]. Specifically, we generated 1000 transactions where the transaction length $l_i$ is generated according to a Zipf distribution over the range [1–50] time units.
with the default Zipf parameter for skewness ($\alpha$) set to 0.5 and it is skewed toward short transactions.

**Workload:** Transactions were generated first as described above, then based on a desired system utilization; arrival times of transactions were assigned according to a Poisson process. The arrival rate of the Poisson distribution is set equal to $\text{SystemUtilization} \div \text{AvgTransactionLength}$, where $\text{SystemUtilization}$ is a simulation parameter that takes the values 0.1, 0.2, 0.3, ... 1.0.

**Deadlines:** Each transaction is assigned a deadline $d_i = a_i + l_i + k_i \times l_i$ where $k_i$ is a factor that determines the ratio between the initial slack time of a transaction and its length. $k_i$ is generated uniformly over the range $[0.0-k_{max}]$, where $k_{max}$ is a simulation parameter with default value of 3.0.

**Workflows:** We generated workflows using two parameters: the maximum workflow length and the maximum number of workflows. The maximum workflow length sets an upper bound on how long the workflow could be. The maximum number of workflows sets an upper bound on the number of workflows a transaction might belong to at one time. The actual workflow length, and number of workflows are uniformly drawn between one and the corresponding upper bound. We varied the maximum workflow length from three to ten, and varied the maximum number of workflows from one to ten.

**Weights:** Each transaction is assigned a weight randomly drawn between one and ten.

The values of performance metrics reported in the next section (i.e., average-tardiness, average weighted-tardiness and maximum weighted-tardiness) are the averages of five runs for each experiment setting. We have conducted multiple experiments to examine all possible parameters values that are summarized in Table IV-A. In all our experiments, ASETS* policy significantly outperformed the other scheduling policies and exhibited the same performance as the sample of representative results presented below.

**B. Experimental Results**

We first present the performance evaluation of ASETS* at the transaction level (Section IV-C). Then, Section IV-D presents the evaluation of ASETS* at the workflow level while all weights are equal. The evaluation of ASETS* in the general case is given in Section IV-E. Finally, the evaluation of balance-aware ASETS* is demonstrated in Section IV-F.

**C. ASETS* at the Transaction Level**

In our first experiment, we measured the average tardiness for the five scheduling policies mentioned above as the system utilization increases from 0.1 to 1.0, with Zipf’s parameter $\alpha = 0.5$ and $k_{max} = 3.0$.

The results for that experiment setting are shown in Figures 8 and 9. Figure 8 shows the average tardiness at low utilization while Figure 9 shows the average tardiness at high utilization (we split the utilization across two figures to zoom in for better understanding of the system behavior). Specifically, at low utilization (Figure 8), the system is able to meet most of the deadlines, and hence, $EDF$ performs better than $SRPT$. As the utilization grows, the system cannot meet all the deadlines, and $SRPT$ starts to approach $EDF$ until it outperforms it at utilization 0.6.

$ASETS^*$ on the other hand, outperforms both $EDF$ and $SRPT$ for all values of utilization. Notice that the maximum improvements provided by $ASETS^*$ is around the cross-over point between $EDF$ and $SRPT$ where it reduces the average tardiness by up to 30%. This is further illustrated in Figure 10, where we plot the average tardiness of $ASETS^*$ normalized to that of $EDF$ as well as $SRPT$.

Figure 10 also shows that $ASETS^*$ outperforms $EDF$ even at very low utilization values. The reason is that though the overall average utilization is low, 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, $ASETS^*$ automatically incorporates some $SRPT$ scheduling to avoid the domino effect of $EDF$. Similarly, at high utilization, $ASETS^*$ outperforms $SRPT$ as it incorporates some $EDF$ scheduling as needed.

The next set of results shows the performance of our proposed algorithm under different deadline settings. Specif-
ically, we compared the performance of ASETS$^*$ to SRPT and EDF under different values of $k_{max}$. Figures 11, 12, and 13 show the results for $k_{max}$ values of 1, 2, and 4, respectively. These are in addition to the results of $k_{max} = 3$ presented above in Figure 10. These results show that ASETS$^*$ constantly outperforms the other two algorithms under the different settings, with the maximum gain be at the cross-over area. It is also interesting to notice that the cross-over point moves further to the right (i.e., higher utilization) as we increase the value of $k_{max}$. The reason is that the more loose the deadlines are (larger $k_{max}$) the more chances EDF has to catch up if it missed deadlines. Hence, EDF can cope with higher utilization and outperforms SRPT for a longer range of utilization.

Finally, we examined the performance of ASETS$^*$ under different transaction length distribution skewness, while fixing the deadline slackness parameter $k_{max} = 3.0$. Specifically we changed the Zipf skewness parameter $\alpha$. We omit the plots here due to space limitations. We encountered the same behavior that ASETS$^*$ constantly outperforms both SRPT and EDF under all utilizations. We also observed that the more skewed the transaction length distribution, the earlier (i.e., at lower utilization) the cross-over point between EDF and SRPT. This is because the deadline is relative to the transaction length. Thus, the more skewed the distribution of transaction lengths, the tighter the deadlines, which leads to a higher level of system utilization, and this lets SRPT take the lead earlier.

D. ASETS$^*$ at the Workflow Level

In this section, we present a sample of the results evaluating the performance of ASETS$^*$. In Figure 14, we compare the average tardiness of ASETS$^*$ to that of Ready. The results show that ASETS$^*$ outperforms Ready by improving the average tardiness between 28% and 57%. In this experiment, the maximum number of workflows was set to one. Similarly, maximum workflow length was set to five in this experiment.

We also conducted several experiments with different values for the maximum workflow length and maximum number of workflows. In all cases we found similar and even better performance than the presented sample, i.e., ASETS$^*$ outperforms Ready under all cases. The percentage improvement in average tardiness was 44% on average.

E. ASETS$^*$: the General Case

In this section, we demonstrate the performance of ASETS$^*$ in the general case. Recall from Section III-C that ASETS$^*$ handles the case when both precedence constraints exist, and each transaction is assigned a weight. The objective here
is to minimize the weighted average tardiness as defined in Section III-C.

In Figure 15, the average tardiness of \textit{ASETS}^* is compared to that of \textit{EDF} and \textit{HDF}. \textit{EDF} is handling low system utilization better, while \textit{HDF} is the optimal policy under high system utilization. As can be seen from the figure, \textit{ASETS}* outperforms both \textit{EDF} and \textit{HDF} under all system utilization, combining the advantages of both algorithms (as was the case for \textit{ASETS}* compared to \textit{EDF} and \textit{SRPT}).

\textbf{F. ASETS*: Balance-aware}

Finally, we show the trade-off between worst- and average-case performance of \textit{ASETS}* (balance-aware). Note that \textit{ASETS}* here is working at the workflow level, with different weights being assigned to the transactions. We ran this experiment for different activation rate values. We changed the time-based activation rate from 0.002 to 0.01, and the count-based activation rate from 0.02 to 0.1. Same behavior was obtained in both cases, we present here the time-based case only to avoid repetition.

Figure 16 shows the maximum weighted tardiness of \textit{ASETS}* (balance-aware) in comparison to that of \textit{ASETS}* for the same activation rate variation. Again, as expected, balance-aware \textit{ASETS}* increases the average weighted tardiness, and the gap increases as the activation-rate increases. However, the increase is up to 5% (at activation-rate of 0.01) while the improvement in the maximum weighted tardiness is up to 27% at same activation rate, with the minimum improvement 7%.

We see that \textit{ASETS}* (balance-aware) decreases the maximum weighted average tardiness over \textit{ASETS}*. As expected, the improvement increases as the activation rate increases.

Clearly the reason behind this behavior is that the lower the activation rate, the less number of transactions that get to run out of \textit{ASETS}* order. Thus, the worst case performance is closer to that of \textit{ASETS}*, while it deviates (improves) as the activation rate increases.

On the other hand, Figure 17 shows the average weighted tardiness of \textit{ASETS}* (balance-aware) and that of \textit{ASETS}* for the same activation rate variation. Again, as expected, balance-aware \textit{ASETS}* increases the average weighted tardiness, and the gap increases as the activation-rate increases. However, the increase is up to 5% (at activation-rate of 0.01) while the improvement in the maximum weighted tardiness is up to 27% at same activation rate, with the minimum improvement 7%.

V. RELATED WORK

Our proposed novel parameter-free adaptive scheduling policy for web-transactions builds on previous work on Web-Databases and Real-Time scheduling. In this section, we review this work and contrast it with ours.

Previous research efforts have proposed several hybrid approaches for scheduling web-requests and real-time transac-
ations (e.g., [13], [3], [5], [6], [4]). However, these approaches have mainly focused on maximizing the hit-ratio (i.e., the number of transactions that meet their deadlines) or maximizing the system gain when each transaction is associated with a value, which could represent the popularity of a web-page. Below, we discuss some of these hybrid approaches since they share some features with our proposed ASETS∗ policy.

For instance, the work in [3] studies the performance of algorithms that use deadlines only, values only, or a mix of both in assigning transaction priorities. Specifically, it studies the Highest Value First (HVF) and the Highest Density First (HDF) policies. It also proposes a hybrid policy called MIX which uses a linear combination between the value and the absolute deadline in order to maximize the hit-ratio. Although it seems similar to our proposed ASETS∗, there are two main distinctions. First, ASETS∗ automatically adapts to different workloads, switching between HDF and EDF while MIX statically combines both of them using a system parameter. Second, ASETS∗ optimizes for average weighted tardiness, while MIX optimizes for Hit-Value Ratio.

Also, toward maximizing the hit-ratio, the work in [5] proposes a hybrid algorithm to schedule real-time transactions. The algorithm divides transactions into two sets, one to be scheduled using EDF, and another to be scheduled randomly where the size of each list is determined based on feedback of the achieved hit-ratio. The work in [5] further extends the proposed approach to maximize system gain (weighted hit-ratio) when transactions are associated with values. In [4], another hybrid approach is proposed to schedule web-broadcasts. [4] proposes MIA which is a hybrid approach between SRPT and EDF that also considers the popularity of broadcast items to maximize the total system gain.

Scheduling real-time transactions under precedence constraints (i.e., at the workflow level) was studied in [13]. It was shown in [13] that EDF is optimal if precedence constraints are consistent. They also provided general necessary and sufficient conditions for scheduling under precedence constraints.

Previous work has also studied the interaction between transaction scheduling and concurrency control as in [9] while ASETS∗ assumes query-transactions only. Also the trade-off between QoS and QoD was studied in [7]. ASETS∗ captures this trade-off by optimizing for weighted average-tardiness.

VI. Conclusions

This work was motivated by the need for an adaptive parameter-free scheduling policy that automatically adapts to different load settings for web-databases. In this paper we modeled the interrelated transactions generating a web page as workflows and quantified user satisfaction by associating dynamic web pages with soft-deadlines. Further, we modeled importance of transactions in generating a web page by associating different weights to transactions. Finally, we proposed a new scheduling algorithm called ASETS∗. ASETS∗ is a parameter-free adaptive scheduling algorithm which integrates EDF and HDF/SRPT. ASETS∗ prioritizes the execution of web-transactions with the objective of minimizing weighted tardiness. Also, we demonstrated how ASETS∗ is capable of balancing the trade-off between optimizing average- and worst-case performance.

We evaluated ASETS∗ at the different operation levels experimentally and showed that our proposed policy significantly outperforms the best known scheduling policies in terms of average tardiness or average weighted tardiness by up to 57%.

ACKNOWLEDGMENTS

This work was partially supported by NSF under project AQSIOS (#IIS-0534531) and Career award (#IIS-0746696). The second author is supported in part by the Ontario Ministry of Research and Innovation Postdoctoral Fellowship. The authors would like to thank the anonymous reviewers for their insightful comments and the members of the Advanced Data Management Technologies Laboratory for their help and, in particular, Brian Wongchaowart for his feedback on earlier drafts of this paper.

REFERENCES