Achieving class-based QoS for transactional workloads

Bianca Schroeder Mor Harchol-Balter
Carnegie Mellon University
Department of Computer Science
Pittsburgh, PA USA
<bianca, harchol>@cs.cmu.edu

Arun Iyengar Erich Nahum
IBM T.J. Watson Research Center
Yorktown Heights, NY USA
<aruni,nahum>@us.ibm.com

Abstract

In e-commerce applications certain classes of users desire mean response time guarantees and are willing to pay for this preferential level of service. Unfortunately, todays commercial DBMS, which lie at the heart of most e-commerce applications, do not provide adequate support for class-based quality of service guarantees. When designing methods for providing such guarantees, it is furthermore desirable that they be effective across workloads and not rely on changes to DBMS internals for portability and ease of implementation. This paper presents an External Queue Management System (EQMS) that strives to achieve the above goals.

1. Introduction

Transaction processing systems lie at the core of modern e-commerce applications such as on-line retail stores, banks and airline reservation systems. The economic success of these applications depends on the ability to achieve high user satisfaction, since a single mouse-click is all that it takes a frustrated user to switch to a competitor. Given that system resources are limited and demands are varying, it is difficult to provide optimal performance to *all* users at all times. However, often transactions can be divided into different *classes* based on how important they are to the online retailer. For example, transactions initiated by a "big spending" client are more important than transactions from a client that only browses the site. A natural goal then is to ensure short delays for the class of important transactions, while for the less important transactions longer delays are acceptable.

It is in the financial interest of an online retailer to be able to ensure that certain classes of transactions (financially lucrative ones) are completed within some *target mean response time*. It is also financially desirable for the online retailer to be able to offer a Service Level Agreement (SLA) to certain customers, guaranteeing them some target mean response time that they desire (with possible deteriorated performance for customers without SLAs). This paper proposes and implements algorithms for providing such performance targets on a per-class basis

A guaranteed mean response time for some class of transactions is one form of a *Quality of Service (QoS) target*. In many situations it is useful to provide more general QoS targets such as *percentile targets*, where $x\%$ of response times for a class are guaranteed to be below some value y. Percentile targets are often demanded by clients as part of a Service Level Agreement (SLA), for example to ensure that at least 90% of the client's transactions see a response time below a specified threshold. In addition to per-class response time and percentile targets, another common QoS target is to provide low *variability* in response times. The reason is that users may judge a relatively fast service still unacceptable unless it is also predictable [5, 11, 25].

Because the dominant time associated with serving an e-commerce transaction is often the time spent at the back-end database (rather than the front-end web/app server), it is important that the QoS be applied to the backend database system to control the time spent there. Yet, commercial database management systems (DBMS) do not provide effective service differentiation between different classes of transactions.

In designing a framework for providing class-based QoS targets one strives for the following high-level design goals:

Diverse per-class QoS target metrics The system should allow for an arbitrary number of different classes, where the classes can differ in their arrival rates, transaction types, etc. Each class is associated with one or more QoS targets for (per-class) mean response time, percentiles of response time, variability in response time, best effort, or any combination thereof.

Portability and ease of implementation Ideally the system should be portable across DBMS, and easy to implement.

Self-tuning and self-adaptive The system should ideally have few parameters, all of which are determined by the system, as a function of the QoS targets, without intervention of the database administrator. The system should also automatically self-adapt to changes in the workloads and QoS targets.

Effective across workloads Database workloads are diverse with respect to their resource utilization characteristics (CPU, I/O, etc.). We aim for a solution which is effective across a large range of workloads.

No sacrifice in throughput & overall mean response time Achieving per-class targets should not come at the cost of an increase in the overall (over all classes) mean response time or a drop in overall throughput.

With respect to the above design goals, the prior work is limited. Commercial DBMS provide tools to assign priorities to transactions, however these are not associated with any specific response time targets. Research on real-time databases does not consider mean per-class response time goals, but rather looks only at how an individual transaction can be made to either meet a deadline or be dropped (we never drop transactions). The only existing work on per-class mean response time guarantees for databases is based on modified buffer pool management algorithms [6, 7, 14, 22]. These techniques are not effective across workloads, since they focus only on one resource: Tuning the buffer pool will for example have little effect on CPU-bound or lock-bound workloads. Moreover, they don't cover more diverse QoS goals such as percentile or variability goals. A major limitation of all the above approaches is that they rely on changes to DBMS internals. Their implementation depends on complex DBMS specifics and is neither simple, nor portable across different systems.

Our approach aims at achieving the above high-level design goals through an external frontend scheduler. The scheduler maintains an upper limit on the number of transactions executing simultaneously within the DBMS called the Multi-Programming Limit, or "MPL" (see illustration in Figure 1). If a transaction arrives and finds MPL number of transactions already in the DBMS, the arriving transaction is held back in an external queue. Response time for a transaction includes both waiting time in the external queue (queueing time) and time spent within the DBMS (execution time).

The immediately apparent attribute of our approach is that it lends itself to portability and ease of implementation as there is no dependence on DBMS internals. Also moving the scheduling outside the DBMS, rather than scheduling individual DBMS resources (such as the bufferpool or lock queues), makes it effective across different workloads, independent of the resource utilization.

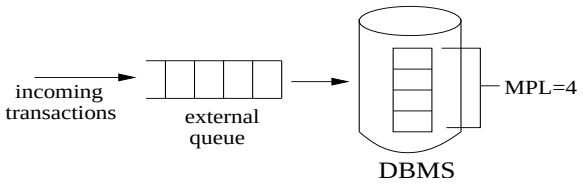

Figure 1. *Simplified view of mechanism used to achieve QoS targets. A fixed limited number of transactions (MPL=4) are allowed into the DBMS simultaneously. The remaining transactions are held in an unlimited external queue. Response time is the time from when a transaction arrives until it completes, including queueing time.*

With respect to obtaining diverse QoS targets, the core idea is that by maintaining a low MPL, we obtain a better estimate of a transaction's execution time within the DBMS, and hence we are able to maintain accurate estimates of the per-class mean execution times. This in turn gives us an upper bound on the queueing time for a transaction, which can be used by the scheduler in order to ensure that QoS targets are met. The actual algorithms that we use are more complex and rely on queueing analysis in order to meet a more diverse set of QoS targets, and behave in a self-adaptive manner.

The external scheduler achieves class differentiation by providing short queueing times for classes with very stringent QoS targets, at the expense of longer queueing times for classes with more relaxed QoS targets. There are no transactions dropped. One inherent difficulty in this approach is that not every set of targets is feasible, e.g., not every class can be guaranteed a really low response time. An external scheduler therefore also needs to include methods for determining whether a set of QoS targets is feasible.

The effectiveness of the external scheduling approach and whether it requires sacrifices in overall performance (e.g. throughput or mean response time) depends on the choice of the MPL. For scheduling to be most effective a very low MPL is desirable, since then at any time only a small number of transactions will be executing inside the DBMS (outside the control of the external scheduler), while a large number are queued under the control of the external scheduler. On the other hand, too low an MPL can hurt the overall performance of the DBMS, e.g., by underutilizing the DBMS resources resulting in a drop in system throughput. Therefore, another core problem an external scheduler needs to solve is that of choosing the MPL.

In this paper we propose and implement a unified external scheduling framework called *EQMS (External Queue Management System)* that addresses all of the above problems. Figure 2 gives an overview of the EQMS architecture.

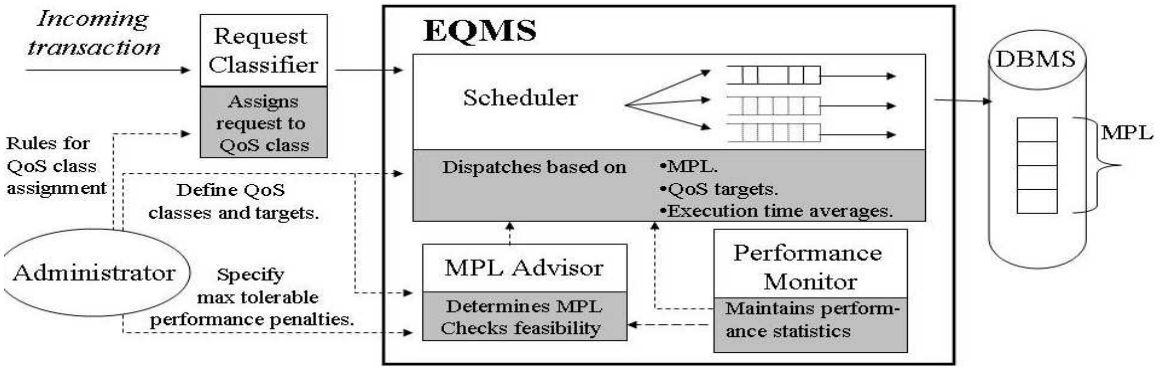

Figure 2. *Overview of the EQMS system.*

The EQMS takes as input a set of classes with one or several QoS targets for each class. These are specified by the online retailer and are not part of the EQMS. The core component of the EQMS is the *Scheduler* which decides on the order in which transactions are dispatched to the DBMS such that the associated QoS targets are met. The scheduler relies on the *MPL Advisor* to determine an MPL that provides sufficient scheduling control, while keeping performance penalties, such as loss in throughput, below a threshold defined by the DBA (database administrator). The MPL Advisor also checks for the feasibility of a given set of targets. The EQMS combines feedback control (based on information collected by the *Performance Monitor*) with queueing theory to operate in a self-tuning and self-adaptive fashion.

We demonstrate the effectiveness of our solution in experiments with two different DBMS, IBM DB2 and PostgreSQL. We create a range of workloads, including CPU-bound, I/O-bound, and high vs. low lock contention workloads, based on different configurations of TPC-C [23] and TPC-W [24]. We show that our solutions apply equally well across all workloads studied. The reason is that the core idea of limiting the MPL reduces contention within the DBMS at the bottleneck resource, independent of what the particular bottleneck resource is.

The paper is organized as follows: Section 2 reviews related work. Section 3 describes the experimental setup. Section 4 details the algorithms used by the Scheduler to achieve class-based mean response time targets and Section 5 explains how to schedule for more complex QoS goals, including percentile and variability goals. Section 6 describes the techniques used by the Scheduler and the MPL Advisor to adapt in dynamic environments. We conclude in Section 7.

2. Related work

When looking at prior work on providing QoS guarantees for DBMS transactions two points are apparent: First, prior work focuses on scheduling datbase internal resources and hence requires modifications to DBMS internals; our goal is to provide QoS guarantees externally, transparent to the underlying DBMS. Second, only per-class mean response time targets have been considered; our goal is to provide methods for a wider range of QoS targets, including variabiility or percentile targets.

Below we describe prior work on providing guarantees for DBMS transactions. Most of the work is in the area of real-time DBMS (RTDBMS), which is concerned with deadlines rather than targets involving mean response time. Commercial DBMS provide tools to assign priorities to transactions, however these are not associated with any specific response time targets. The little work that involves per-class guarantees is primarily simulation-only, and does not cover complex QoS goals such as percentile or variability goals, and is not portable in that it requires the modification of database internals (e.g. the bufferpool manager).

Work on RTDBMS

In Real-time DBMS, there is a deadline (typically a hard deadline) associated with each transaction. The goal of RTDBMS is to minimize the number of transactions which miss their deadlines. If a hard deadline is missed, the transaction is dropped. Examples of work in this area include: [1–4, 12]. This work is different from our own in that it does not allow for mean response time targets or variability targets. Also, in our work, no transactions are dropped. The RTDBMS typically involves using a specialized database engine, and the mechanism is implemented internally, making it less portable.

Commercial DBMS

As a testament to the importance of the problem of providing different service levels most commercial DBMS provide priority mechanisms in some form. For example, both IBM DB2 [16] and Oracle [20] offer CPU scheduling

tools for prioritizing transactions. Although different classes are given different priorities with respect to system resources, it is not clear how these priority levels relate to achieving specific response time targets. Towards this end, Kraiss et al. [15] try to map each class to some fixed priority such that scheduling based on priorities will meet the desired response time targets. Such an assignment of priorities to classes does not always exist.

Towards per-class mean response time targets

Carey et al. [10] consider the situation of two classes, where they strive to make the mean response time for the high priority class as low as possible by scheduling internal DBMS resources on a read-only workload. Their work is a simulation study. In our recent work [18] we consider the same problem for a more general workload (TPC-C and TPC-W) under a variety of DBMS via an implementation of (DBMS internal) lock scheduling and CPU scheduling.

More closely related to our current work are the following papers, [6–8,22], all of which have multiple classes each with a different mean response time target. Other QoS targets are not considered. Their approach is to schedule internal memory (buffer pool management). The above are all simulation studies.

3. Experimental setup

As representative workloads for transactional web applications, we choose the TPC-C [23] and TPC-W [24] benchmarks. The TPC-C workload in this study is generated using software developed at IBM. The TPC-W workload is generated using the TPC-W Kit from PHARM [9], though minor modifications are made to improve performance, including rewriting the connection pooling algorithm to reduce overhead.

Different configurations of these workloads (number of warehouses, number of clients) result in different levels of resource utilization for the hardware resources: CPU and I/O. We experiment with 4 different configurations of TPC-C and TPC-W as shown in Table 1(top). We chose these configurations in order to cover different combinations of resource utilization levels (see Table 1(bottom)). In addition, varying the configuration will also result in different levels of lock contention [18]. For example, lock contention is a large component of a transaction's lifetime in workloads W_{I0} and $W_{I0>CPU}$, but not in the other workloads.

The TPC-C and TPC-W benchmarks are defined to be used as closed systems, and we use them this way. We assume a zero "think time" throughout. All results in the paper have been repeated with non-zero think times, and with open system configurations and results have been found to be similar. Due to a lack of space, unless otherwise stated, we show only the results for zero think times, allowing us to focus on the effect of varying the MPL in all the graphs.

Workload	Benchmark	Confi g	Data-base	CPU load	I/O load
W_{IO}	TPC-C	40 WH, 100 clients	4GB	low	high
$W_{IO>CPU}$	TPC-C	10 WH, 100 clients	1GB	med.	high
$W_{CPU>IO}$	TPC-W Shopping	100 EBs, 10K items, 140K customers	300MB	high	med.
W_{CPU}	TPC-W Browsing	100 EBs, 10K items, 140K customers	300MB	high	low

Table 1. *Description of the experimental workloads.*

The DBMS we experiment with are IBM DB2 [16] version 8.1, and PostgreSQL [19] version 7.3. *Due to lack of space, all results graphs throughout the paper pertain to the IBM DB2 DBMS. Results for PostgreSQL are very similar and we describe these in words only.* In all experiments the DBMS is running on a 2.4-GHz Pentium 4 with 3GB RAM, running Linux 2.4.23, with a buffer pool size of 2GB. The machine is equipped with two 120GB IDE drives, one of which we use for the database log and the other one for the data. The client generator is run on a separate machine with the same specifications as the database server, and is directly connected to the database server through a network switch.

4. Achieving response time targets

In this section we assume that each of the QoS targets is a specific mean response time target for each class. Specifically, class i transactions have a target mean response time of τ_i. After introducing some notation, we explain the algorithms used by the Scheduler to achieve the per-class response time targets and to determine whether a set of targets is feasible.

4.1. Notation

The notation we use in order to formally explain the external scheduling algorithms is summarized in Table 2, and is straightforward. The mean response time of transactions is denoted by T, and can be divided into T^Q and T^{DBMS}, where the former denotes the mean time the transactions spend queueing externally to the DBMS and the latter quantity is the mean time that the transactions spend within the DBMS. That is,

$$T = T^Q + T^{DBMS}$$

T^Q	Mean time transactions spend waiting in external queue in system with external scheduling
T^{DBMS}	Mean time transactions spend executing in the DBMS in system with external scheduling
T_i^Q	Mean time transactions in class i spend waiting in external queue
T_i^{DBMS}	Mean time transactions in class i spend executing in the DBMS in system with external scheduling
T	Overall mean response time, i.e. sum of T_Q and T_{DBMS}
R	Mean response time in original (no external scheduling) system
R_i	Mean response time of class i transactions in original (no external scheduling) system
τ_i	Mean response time target of class i
p_i	Fraction of transactions that are class i
t_{curr}	current time
$T_i^{x\%}$	x-th percentile of T_i
$T_i^{DBMS\,x\%}$	x-th percentile of T_i^{DBMS}
$\tau_i^{x\%}$	Target for x-th percentile of T_i

Table 2. *Notation*

Furthermore, we denote the per-class response times via a subscript i denoting class i, where T_i denotes the mean response time for class i transactions, and

$$T_i = T_i^Q + T_i^{DBMS}$$

Notice that the above notation is different from the τ_i's which denotes the i^{th} class' mean response time target. Lastly, we define R to be the mean response time in the original system, without external scheduling. The remaining notation will be explained as needed.

Measurements of the quantities introduced above, i.e. the queuing times, execution times, and total mean response times, both per class and aggregated, are needed by the Scheduler and the MPL Advisor and are therefore tracked by the Performance Monitor.

4.2. The basic algorithm

The Scheduler relies on the MPL Advisor to choose an MPL so that the time spent within the DBMS is low and predictable. In particular, since the Scheduler cannot control the time a transaction takes to execute inside the DBMS, the MPL has to be low enough such that for each class the expected time within the DBMS is lower than the class' response time target. Given n QoS classes with response time targets $\{\tau_1, \ldots, \tau_n\}$, the MPL needs to ensure that for each class i

$$T_i^{DBMS} < \tau_i$$

The main question for the Scheduler is then how to order the transactions within the external queue to achieve the targets. Observe that for each class i the Scheduler knows the mean target response time τ_i and can obtain the mean database execution time T_i^{DBMS} from the Performance Monitor. It can therefore determine how much "slack" it has in scheduling transactions from this class: transactions in class i can afford on average to wait up to but not more than

$$s_i = \tau_i - T_i^{DBMS}$$

time units in the external queue before they should start executing in the DBMS.

Based on the slack of a transaction the Scheduler computes a timestamp for when the transaction should be dispatched out of the external queue and into the DBMS, which we call the *dispatch target time*. Formally, if a new transaction of class i arrives at time t_a its dispatch target t_d is

$$t_d = t_a + s_i = t_a + \tau_i - T_i^{DBMS}.$$

Whenever a transaction completes at the DBMS, and we have to pick the next transaction for execution from the external queue, we pick the transactions in increasing order of their dispatch targets (t_d value).

We demonstrate the viability of the above algorithm experimentally. The above high level description omits an important issue arising in practice: how does the Scheduler adjust to surges and fluctuations in system load which might make it impossible to achieve all the targets. This practical concern will be addressed in Section 6.

4.3. Feasibility of assignment

The Scheduler distinguishes two types of infeasible targets. The first one has already been explained above and includes per-class targets that are lower than the per-class mean execution time (i.e. the average time spent inside the DBMS). The mean response time of a class is the sum of its queueing time and its execution time, and can obviously not be smaller than either one of its components. The first condition for a target to be feasible is therefore the following:

$$T_i^{DBMS} < \tau_i$$

It is important to note that the mean execution time T_i^{DBMS} depends on the MPL: a smaller MPL leads to less contention at the DBMS and therefore to shorter execution times. It is therefore the goal of the MPL Advisor to recommend an MPL that meets the above condition, provided that this does not come at a performance penalty (e.g. in terms of throughput loss) beyond what the DBA has specified as tolerable. If the MPL Advisor cannot determine an MPL that satisfies the above condition, then the target τ_i is not feasible. The details of how the MPL Advisor works will be explained in Section 6.1.

The second type of infeasible targets comprises those that cannot be achieved due to a simple lack of system resources (e.g. suppose every class desires a really low response time guarantee). More precisely, we don't expect the overall mean response time under class-based prioritization to be lower than for the unprioritized system. That is the weighted average over all per-class mean response time targets is not expected to be lower than the mean response time in the original unprioritized system.

We describe a simple condition for determining whether a set of per-class mean response time targets is feasible, i.e., whether the set of targets is achievable via some algorithm.

We start by defining the overall target mean response time (aggregated over all classes):

$$\tau_{overall} = \sum_{i=1}^{n} p_i \cdot \tau_i$$

Recall R represents the mean response time in the original system (without scheduling).

Obviously a necessary condition for achieving the individual τ_i's (via some ordering of the external queue) is that

$$\tau_{overall} > R$$

We now argue (only intuitively) that this also represents a sufficient condition. The crux of the argument is that the external scheduling (with the limited MPL) does not increase the overall measured mean response time T as compared with the original R. That is, when the MPL is chosen carefully (as in Section 6.1) we have

$$T \approx R$$

Hence the above condition also implies

$$\tau_{overall} > T$$

which is sufficient.

4.4. Results for mean response time targets

We experimentally evaluate the accuracy of the Scheduler in achieving per-class mean response time targets using the four workloads in Table 1. In all experiments we use an MPL of 20, since for our workloads this MPL is high enough that neither throughput nor overall mean response time is sacrificed. Tables 3 and 4 show detailed results for W_{I0} and W_{CPU} respectively under IBM DB2. Results are shown in the "Measured" column corresponding to the QoS target specified in the previous two columns. Mean and max values are specified for a sequence of 10 experimental runs, each consisting of 25,000 transactions. At the moment, we are only concerned with the first two rows of these tables, which consider per-class response time targets. We have experimented with three different classes with different targets and frequencies. As shown in the tables, we are always able to achieve within 8% of the desired per-class response time targets for W_{IO} (Table 3) and within 12% of the desired per-class response time targets for W_{CPU} (Table 4). Recall that W_{CPU} corresponds to the TPC-W workload which is more variable. Results for $W_{I0>CPU}$ are very similar to those for W_{I0} with errors ranging between 0-5% and results for $W_{CPU>IO}$ are very similar to those for W_{CPU} with errors ranging between 1-11%. We also repeated all experiments for PostgreSQL, where results are slightly worse but still within 15% of the targets.

5. More complex QoS targets

In this section we consider more complex QoS targets. These include: Reducing overall variance in response times (aggregated over all class transactions)(Section 5.1); and achieving targets on the x^{th} percentile of response time for multiple classes (Section 5.2).

5.1. Reducing variance

In addition to desiring low mean response time, users are equally desirous of low variability in response times [5]. Both W_{CPU} and W_{IO} benchmarks are composed of a fixed set of transaction types. We find that although the variance within each transaction type is not too high, the overall variance in response time across all transaction types is quite high. Specifically, for W_{CPU}, the squared coefficient of variation (C^2) for individual transaction types ranges from $C^2 = 2$ to $C^2 = 5$; however over all transaction types, we measure $C^2 = 15$. W_{IO} is less variable. For individual transaction types we measure values ranging from $C^2 = .15$ to $C^2 = 0.8$, while looking across all transaction types we measure $C^2 = 2.3$. As a reference point, the exponential distribution has $C^2 = 1$.

Figure 3(left column) shows the (original) response times for the different transactions under W_{CPU} and W_{IO}, for IBM DB2. Our approach to combatting variability is to decrease the response time of the long transactions (by giving them priority) and in exchange increase the response

Experiment type	Class	Frequency	Priority	QoS target	Target	Measured		
						Mean	Max	Avg. error
Response times	C1	10%	1	Resp Time	0.7 sec	0.725	0.728	3.5%
	C2	20%	2	Resp Time	1 sec	1.02	1.029	2.0%
	C3	70%	N/A	Best effort	N/A	1.60	1.69	N/A
Response times	C1	40%	1	Resp Time	0.6 sec	0.653	0.657	8.0%
	C2	40%	2	Resp Time	1.3 sec	1.366	1.37	5.0%
	C3	20%	N/A	Best effort	N/A	2.54	2.59	N/A
Percentiles	C1	10%	1	80th %tile	1 sec	0.98	1.026	0%
	C2	10%	2	95th %tile	2 sec	2.03	2.159	1.0%
	C3	80%	N/A	Best effort	N/A	80th %tile: 1.96 95th %tile: 2.51	2.01 2.89	N/A N/A
Percentiles	C1	20%	1	80th %tile	1 sec	0.981	1.06	0%
	C2	20%	2	95th %tile	2 sec	2.002	2.018	.1%
	C3	60%	N/A	Best effort	N/A	80th %tile: 2.08 95th %tile: 2.69	2.18 2.77	N/A N/A
Variability	C1	100%	1	Reduce var	N/A	$C^2 = 0.108$ (before $C^2 = 2.3$)		
Combined	C1	10%	1	Resp Time	0.7 sec	0.73	0.78	4.0%
	C2	10%	2	Resp Time	1 sec	0.98	1.09	2.0%
	C3	10%	3	80th %tile	1 sec	0.978	0.99	2.0%
	C4	10%	4	95th %tile	2 sec	2.04	2.1	2.0%
	C5	60%	N/A	Best effort	N/A		1.7	

Table 3. *Summary of results for different QoS targets for W_{IO}.*

Experiment type	Class	Frequency	Priority	QoS target	Target	Measured		
						Mean	Max	Avg. error
Response times	C1	10%	1	Resp Time	3 sec	3.37	3.570	12%
	C2	20%	2	Resp Time	6 sec	6.564	6642	9%
	C3	70%	N/A	Best effort	N/A	10.6	10.8	N/A
Response times	C1	25%	1	Resp Time	2.5 sec	2.79	3.15	11%
	C2	25%	2	Resp Time	6.5 sec	6.6	7.4	1.5%
	C3	50%	N/A	Best effort	N/A	12.9	14.08	N/A
Percentiles	C1	10%	1	80th %tile	3 sec	3.068		2.7%
	C2	10%	2	95th %tile	12 sec	5.9	6.2	0%
	C3	80%	N/A	Best effort	NA	80th %tile: 16.1 95th %tile: 19.6	16.5 21.4	N/A N/A
Percentiles	C1	20%	1	80th %tile	3 sec	2.7	2.89	0%
	C2	20%	2	95th %tile	9 sec	7.9	8.4	0%
	C3	60%	N/A	Best effort	N/A	80th %tile: 12.01 90th %tile: 19.3	12.08 20.0	N/A N/A
Variability	C1	100%	1	Reduce var	N/A	$C^2 = 0.19$ (before $C^2 = 15$)		
Combined	C1	10%	1	Resp Time	3 sec	3.271	3.520	9%
	C2	10%	2	Resp Time	6 sec	6.285	6.678	6%
	C3	10%	3	80th %tile	2.5 sec	2.701	2.945	8%
	C4	60%	N/A	Best effort	N/A	80th %tile 11.8 95th %tile 19.2 Mean 10.5	12.4 20.4 10.9	N/A N/A N/A

Table 4. *Summary of results for different QoS targets for W_{CPU}.*

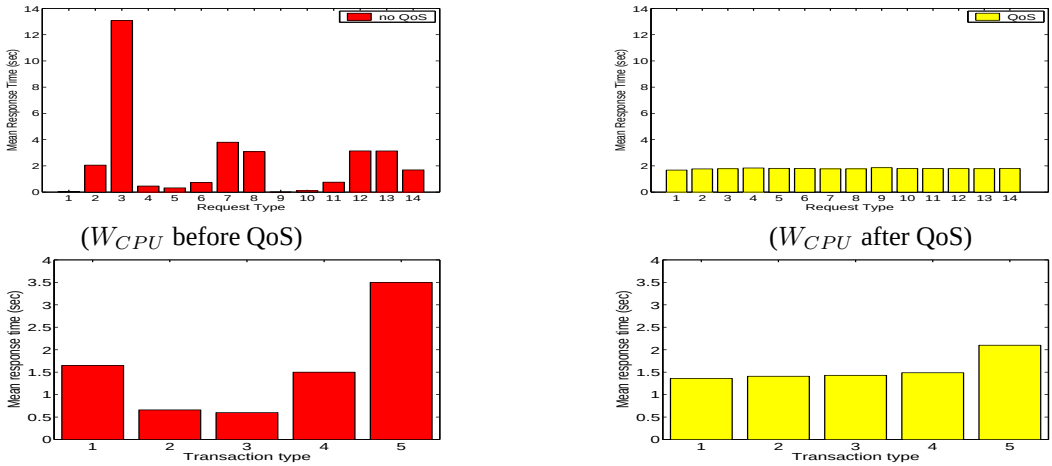

Figure 3. *QoS target reducing variability: Results for W_{CPU}(top) and W_{IO}(bottom).*

time of the short transactions, where the goal is to make all transaction response times as close to the overall mean response time as possible. This turns out to be possible because in typical workloads the fraction of transactions with very long response times is quite small as compared with the fraction of transactions with short response times (as in Pareto's "80-20 rule"). Figure 3(right column) shows the results of equalizing the response times, hence greatly decreasing variance. Under IBM DB2, for W_{CPU} we are able to decrease C^2 from 15 to 0.19. For W_{IO} we are able to decrease C^2 from 2.3 to 0.11. These results are summarized in Tables 3 and 4. Under PostgreSQL, for W_{CPU} we are able to decrease C^2 from 14 to 0.09. For W_{IO} we are able to decrease C^2 from 1.6 to 0.8.

The exact algorithm for reducing variability is easy to implement within our external scheduling framework. We start with the measured overall mean response time of the original system R. We denote the mean response time for the i^{th} transaction type by T_i. Initially some of the T_i's are higher than R and some are lower. To make the system more predictable, we assign type i transactions a target mean response time of $\tau_i = R$. We then apply the standard method for achieving per-class target mean response times.

For this method to work, it is important to note that it is desirable that the variability within each type is low, so that each type is more predictable. For many OLTP servers, e.g. the database backend of a Web site, this is the case: There are a limited number of possible transaction types that the user interface allows for, e.g. ordering, product search, retrieving shopping cart contents, and these transaction types are each limited in scope, resulting in low response time variability within each type.

5.2. Meeting xth percentile targets

Mean target response times are loose in that they can be heavily influenced by a small percentage of transactions. It is conceivable that some customers might prefer stronger guarantees, namely that 90%, say, of their transactions have response times strictly below some target. In this section we describe how to obtain per-class percentile target guarantees.

Consider the example of setting a 90^{th} percentile target denoted by $\tau_i^{90\%}$ for the transactions in class i. Our approach for mean response time targets doesn't apply to percentile targets. Thus we need a new approach. Our percentile target approach has two parts: First the MPL Advisor determines an MPL value which ensures a 90^{th} percentile target on just the execution time T_i^{DBMS}, i.e.

$$T_i^{DBMS\,90\%} < \tau$$

where $T_i^{DBMS\,90\%}$ denotes the 90^{th} percentile of execution times. We next define an algorithm for scheduling the external queue that uses $T_i^{DBMS\,90^{\%}}$ to achieve a 90^{th} percentile target on the response time for class i.

The second step of our approach is to convert $T_i^{DBMS,90\%}$ into a 90^{th} percentile result for response time. Observe that if the queueing time T_i^Q is bounded by some c, the resulting 90^{th} percentile response time is bounded by

$$T_i^{90\%} \leq c + T_i^{DBMS,90\%}$$

That means, when scheduling a transaction, in order to ensure a given percentile target $\tau_i^{90\%}$, i.e. ensure that

$$T_i^{90\%} \leq \tau_i^{90\%}$$

the amount of slack we have in scheduling this transaction is

$$\tau_i^{90\%} - T_i^{DBMS,90\%}$$

We can hence translate percentile targets to dispatch targets as follows: assign a transaction with target target $\tau_i^{90\%}$ the dispatch target of:

$$t_d = t_{curr} + \tau_i^{90\%} - T_i^{DBMS,90\%}$$

As before we schedule transactions from the queue in order of increasing dispatch targets.

Tables 3 and 4 show results for various experiments with per-class percentile target targets under IBM DB2 for W_{IO} and W_{CPU}. As shown, in all experiments, for both workloads, we are able to achieve our percentile response time targets usually within 3%. Results for $W_{IO>CPU}$ and $W_{CPU<IO}$ are comparable, with errors in achieving percentile targets ranging from 1-4%. For our experiments with PostgreSQL this number becomes 10%.

5.3. Combination targets

Finally, it is quite plausible that (i) different customer classes might desire different types of QoS targets, and (ii) a given customer class might simultaneously request multiple targets (e.g., a target for the mean and a percentile target). Both of these "combination" scenarios are easy to achieve in our external scheduling framework since all targets are immediately mapped to dispatch targets and transactions are then pulled from the external queue in order of these targets. A transaction having multiple QoS targets is assigned the most stringent of all of its corresponding dispatch targets.

Some results involving combination targets are shown for IBM DB2 in Tables 3 and 4, and all targets are achieved with very high accuracy.

6. Making the EQMS self-tuning and adaptive

This section details the self-tuning and self-adaptive features of the EQMS that make it robust to dynamic situations. We first explain how the MPL Advisor tunes and dynamically adapts the MPL, the most important parameter of the EQMS. We then discuss how the Scheduler copes with varying load conditions, especially with sudden load surges.

6.1. Details of the MPL Advisor

One big advantage of the EQMS approach is that there are very few parameters to tune. Essentially, the one most important parameter is the MPL. The proper choice of the MPL is crucial, though, since too high an MPL will provide the scheduler with only little control on class differentiation, while a too low MPL can harm the overall system performance. Below we first describe how the MPL Advisor determines a lower bound on the MPL (to limit loss in throughput, and increase in response time) and then how it determines an upper bound on the MPL (to allow sufficient control for achieving a given set of QoS targets).

Determining a lower bound on the MPL

There are two potential risks involved in choosing the MPL too low. First, a low MPL may cause the DBMS resources to be underutilized, leading to loss in throughput. Second, a low MPL may increase overall mean response time, since it enforces a stricter queueing of transactions, resulting in short transactions being forced to queue behind long ones (Head-of-Line-Blocking). Our experimental results across various workloads and system configurations show that the wrong choice of MPL can result in a drop in throughput by a factor of 10 and a more than tenfold increase in overall mean response time.

What the MPL Advisor does is to find a lower bound on the MPL which limits the above problems such that throughput loss and increase in mean response time are within tolerable range (where "tolerable" is specified by the DBA). Finding a good lower bound is a difficult problem (and is left as an open problem even in recent publications [13]). Our basic approach is to use a control loop augmented with queueing theoretic guidance. We develop queueing theoretic models that capture the basic properties of the relationship between system throughput and response time and the MPL. Analysis of the models (parameterized based on the given system and workload) provides us with a good initial MPL value, which we then fine-tune through a control-loop. "Jump-starting" the control-loop with a close-to-optimal starting value provides fast convergence times, even when using only small conservative constant adjustments in each iteration. The details are involved and addressed in a separate paper [21], also in submission.

Determining an upper bound on the MPL

In Section 4.3 we have identified several necessary conditions for a set of QoS targets to be feasible. In particular, for each class i that has a mean response time target associated with it the following condition needs to hold

$$T_i^{DBMS} < \tau_i$$

and for each class j with a percentile target

$$T_j^{DBMS\,90^{\%}} < \tau_j^{90^{\%}}$$

needs to hold. Since both T_i^{DBMS} and $T_j^{DBMS\,90^{\%}}$ are affected by the MPL (a higher MPL will lead to higher contention at the DBMS and therefore to higher T_i^{DBMS} and $T_j^{DBMS\,90^{\%}}$ values) the feasibility of a set of targets

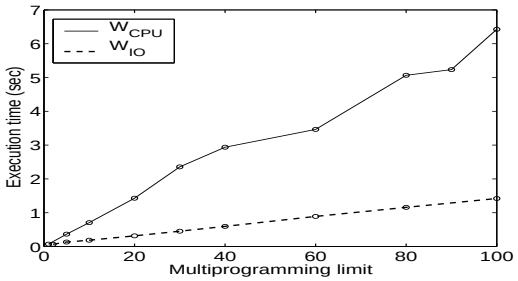

Figure 4. *The mean execution time as a function of the MPL under W_{IO} and W_{CPU}.*

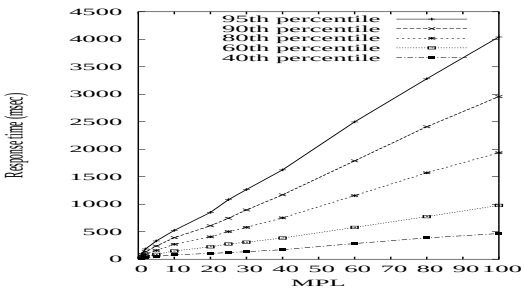

Figure 5. *Percentile of the execution time at the server for different MPLs for W_{IO}.*

depends on the MPL. The goal of the MPL Advisor is to choose an MPL such that the above conditions are met.

The basic mechanism is a control loop that dynamically adjusts the MPL based on measurements provided by the Performance Monitor; if the measured T_i^{DBMS} is higher than desired, the MPL is reduced (provided the lower bound on the MPL determined in the previous subsection is not violated). In order to decide how much the MPL needs to be adjusted the MPL Advisor uses queueing theoretic guidance to provide fast convergence. The details of the algorithm are explained below, first for response time targets then for percentile targets.

Determining the right MPL for ensuring the feasibility of response time targets would be trivial if we had a function that describes the exact relationship between the MPL and the mean execution time T_i^{DBMS}. While we cannot know the exact function, queueing theory reveals a crucial property of this function: According to Little's law [17] the expected execution time T is linear in the MPL value. Figure 4 verifies [1] this law in experiments for W_{IO} and W_{CPU}.

Based on this observation, we can tune the MPL in a control loop similar to the following:

1. Periodically monitor the per-class execution times T_i^{DBMS}.
2. Adjust the MPL if for any class i the mean execution time increases above the per-class target, i.e.
$$T_i^{DBMS} > \tau_i$$
3. Assuming that for some class the execution time is a factor of $f > 1$ times the per-class target target, i.e.
$$T_i^{DBMS} = f \cdot \tau_i$$
we adjust the MPL as follows:
$$MPL_{new} := MPL_{old} \cdot 1/f$$
provided that MPL_{new} does not violate the lower bound on the MPL.

The above algorithm involves multiplicative adjustments. In practice, combining this with small constant adjustments, when close to the target, works well. We find that for obtaining a good estimate of the mean execution time in step 1) it suffices for the Performance Monitor to sample a few hundred transactions in the case of W_{IO} and a few thousand transactions in the case of W_{CPU} (due to the inherently higher variability of workload W_{CPU}). In our experiments the above tuning algorithm finds the optimal MPL with at most 10 iterations.

Moving from mean response time targets to percentile targets the above approach does not work anymore, since Little's law does not apply to percentiles of response time. We solve this problem by approximating the DBMS by a Processor-Sharing (PS) server. While this approximation might be too crude for exact predictions, the hope is that it is still useful for providing intuition on the relationship between the MPL and $T_i^{DBMS\,90^{\%}}$. From queueing theory [2] it follows that in a PS server the percentiles of response time scale linearly with the number of transactions at the server. This result enables us to use the same method we used for mean response time targets for percentile targets as well.

The effectiveness of the above approach for percentile targets hinges on the assumption that our DBMS behaves similarly to a PS server with respect to the linear scaling of percentile response times as a function of MPL. Our experimental results in Figure 5 show that this assumption is in fact valid for our workloads running under IBM DB2. Although not shown, we find that the same result holds for PostgreSQL. We add a disclaimer that the above algorithm works best when all classes have a similar mix of transactions. This was not a necessary condition on all of our

[1] In the case of W_{CPU}, the line is not as straight as for W_{IO}, since the workload is created using TPC-W which exhibits a very high variability in service times, leading to higher variability in experimental results.

[2] This is based on the queueing-theoretic result that under M/G/1/PS, for a given transaction, its expected response time is proportional to both the number of transactions at the server and its service demand.

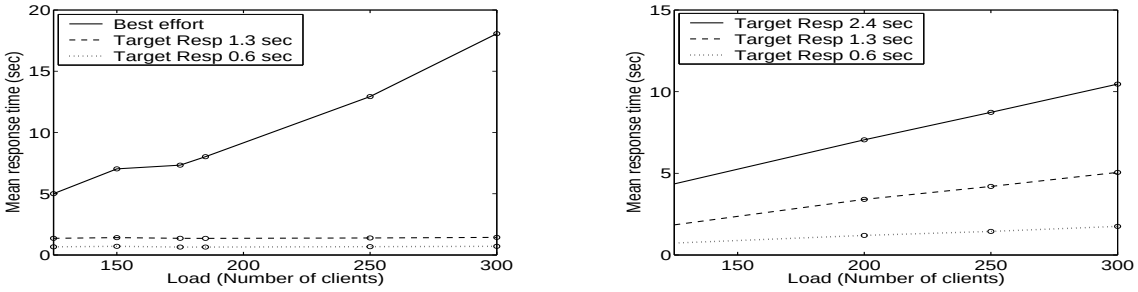

Figure 6. *The graphs show the response times for three classes and workload W_{IO} with increasing load (i.e. increasing number of clients).*

previous QoS algorithms, and is needed only here. If that condition is not met, the control loop that adjusts the MPL needs to be more complex.

6.2. Adapting to load fluctuations

The results in Sections 4 and 5 have assumed a system in steady state with a stable arrival process. During the day, however, the system load may fluctuate. Assuming that the system is never in overload, there should be no need to drop transactions, but the load might rise too high for the current set of targets to be feasible, i.e.

$$T > \tau_{overall}$$

The EQMS offers two approaches for handling this case.

The first approach assumes it is more important for some classes to stay within their target than for others. The DBA indicates this by specifying a priority for each class, in addition to specifying per-class response time targets. These priorities only become effective when load conditions make the per-class targets no longer feasible (i.e. $T > \tau_{overall}$). We detect this situation by checking whether there are any "late" transactions in the external queue, i.e., transactions that have already missed their dispatch target. If t_{curr} is the current time and t_d is the transaction's dispatch target time, then late transactions as those for whom

$$t_{curr} > t_d$$

Whenever we have to choose a new transaction for execution in the DBMS from the external queue, we first check whether there are transactions that are late. If there are, we pick the transaction with the highest priority among the late transactions. If there are no late transactions in the queue, we schedule as usual in the order of the dispatch targets.

An alternative approach is to carry the burden of the excess load equally among all the classes. The burden will be proportionately shared between the classes in the following way: For each transaction we compute its *lateness*, ℓ, as

$$\ell = t_{curr} - t_d$$

When scheduling late transactions, we consider the *relative lateness*, ℓ_{rel}, of the transaction normalized by its target response time τ. Relative lateness is defined as

$$\ell_{rel} = \ell/\tau = (t_{curr} - t_d)/\tau$$

Whenever we choose a transaction for execution in the DBMS from the external queue, we first check whether there are transactions that are late. If there are late transactions in the queue, we pick the transaction with the largest relative lateness, ℓ_{rel}, for execution. If there are no late transactions in the queue, we schedule as usual solely based on the dispatch targets.

We implement both approaches for dealing with fluctuating load described above and experimentally evaluate them on the workload corresponding to the second row in Table 3. In the experiments we vary the load by increasing the number of clients from 100 to 300. The results are shown in Figure 6. In the first approach, the targets for the first two classes are maintained despite the load increase, while only the third (best effort) class suffers. These results for IBM DB2 are shown in Figure 6 (left). Observe that the first two classes have nearly constant mean response times across all loads. In the second approach, we share the burden across all the classes. The results are shown in Figure 6 (right), for the case where the third class has target response time 2.4 seconds. In this figure, the mean response times of all classes increase by the same factor as load increases.

7. Conclusion

Many time-sensitive applications rely on a DBMS backend. From the perspective of these applications, the DBMS is a mysterious black box: Transactions are sent into the DBMS and may take either a very short time (msec) or a

very long time (tens of secs), depending largely on the *other* transactions concurrently in the DBMS. The application has no control over which transactions will take long and which will take a short time.

This paper provides mechanisms and algorithms for controlling the time different transactions spend at a database backend. We provide methods for creating different QoS classes and for meeting specified per-class QoS targets. QoS targets can be mean response time targets, percentile targets, variability targets, or a combinations of targets.

Our solution is an external scheduling mechanism which limits the number of concurrent transactions (MPL) within the DBMS, holding all remaining transactions in an external queue. We find that for our workloads there is a good range of MPL values which allows us to achieve class targets without hurting overall system performance with respect to throughput and overall mean response time.

The algorithms needed to achieve the QoS targets are non-obvious and rely on queueing theory results and analyses. We demonstrate the effectiveness of the algorithms on several benchmark based workloads, including CPU bound, I/O bound and lock bound workloads, in situations with multiple classes and multiple targets per class. However, it is desirable to experiment with other real workloads to further validate the algorithms.

Our external scheduling approach is extremely portable, not just to different DBMS, but also to other types of backend servers. The queuing theoretic arguments in this paper do not depend on the server being a DBMS and apply to general systems as well.

References

[1] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions. In *Proceedings of SIGMOD*, pages 71–81, 1988.

[2] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of Very Large Database Conference*, pages 385–396, 1989.

[3] R. K. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: A performance evaluation. In *IEEE Real-Time Systems Symposium*, pages 113–125, 1990.

[4] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *Transactions on Database Systems*, 17(3):513–560, 1992.

[5] A. Bouch and M. Sasse. It ain't what you charge it's the way that you do it: A user perspective of network QoS and pricing. In *Proceedings of IM'99*, 1999.

[6] K. P. Brown, M. J. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. In *Proceedings of Very Large Database Conference*, pages 328–341, 1993.

[7] K. P. Brown, M. J. Carey, and M. Livny. Goal-oriented buffer management revisited. In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pages 353–346, 1996.

[8] K. P. Brown, M. Mehta, M. J. Carey, and M. Livny. Towards Automated Performance Tuning For Complex Workloads. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 72–84, Santiago, Chile, 1994.

[9] T. Cain, M. Martin, T. Heil, E. Weglarz, and T. Bezenek. Java TPC-W implementation. http://www.ece.wisc.edu/ pharm/tpcw.shtml, 2000.

[10] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. In *Proceedings of Very Large Database Conference*, pages 397–410, 1989.

[11] B. Dellart. How tolerable is delay? Consumers evaluation of internet web sites after waiting. *Journal of Interactive Marketing*, 13:41–54, 1999.

[12] J. Huang, J. Stankovic, K. Ramamritham, and D. F. Towsley. On using priority inheritance in real-time databases. In *IEEE Real-Time Systems Symposium*, pages 210–221, 1991.

[13] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of* ACM SIGMETRICS '04, pages 37 – 48, 2004.

[14] K. D. Kang, S. H. Son, and J. A. Stankovic. Service differentiation in real-time main memory databases. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 29 2002.

[15] A. Kraiss, F. Schoen, G. Weikum, and U. Deppisch. With heart towards response time guarantees for message-based e-services. In *VIII. Conference on Extending Database Technology (EDBT 2002)*, pages 732–735, 2002.

[16] I. T. Lab. IBM DB2 universal database administration guide version 5. Document Number S10J-8157-00, 1997.

[17] J. Little. A proof of the theorem $L = \lambda W$. *Operations Research*, 9:383 – 387, 1961.

[18] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *20th IEEE Conference on Data Engineering (ICDE'2004)*, 2004.

[19] PostgreSQL. `http://www.postgresql.org`.

[20] A. Rhee, S. Chatterjee, and T. Lahiri. The Oracle Database Resource Manager: Scheduling CPU resources at the application. High Performance Transaction Systems Workshop, 2001.

[21] B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. Nahum. How to determine a good multi-programming level for external scheduling. In *submission to ICDE '06 (Paper 597)*.

[22] M. Sinnwell and A. Koenig. Managing distributed memory to meet multiclass workload response time goals. In *15th IEEE Conference on Data Engineering (ICDE'99)*, 1997.

[23] Transaction Processing Performance Council. TPC benchmark C. Number Revision 5.1.0, December 2002.

[24] Transaction Processing Performance Council. TPC benchmark W (web commerce). Number Revision 1.8, February 2002.

[25] M. Zhou and L. Zhou. How does waiting duration information influcence customers' reactions to waiting for services. *Journal of Applied Social Psychology*, 26:1702–1717, 1996.