Priority Mechanisms for OLTP and Transactional Web Applications

David T. McWherter Bianca Schroeder Anastassia Ailamaki[*] Mor Harchol-Balter[†]

Carnegie Mellon University, Dept. of Computer Science

Abstract

Transactional workloads are a hallmark of modern OLTP and Web applications, ranging from electronic commerce and banking to online shopping. Often, the database at the core of these applications is the performance bottleneck. Given the limited resources available to the database, transaction execution times can vary wildly as they compete and wait for critical resources. As the competitor is "only a click away," valuable (high-priority) users must be ensured consistently good performance via QoS and transaction prioritization.

This paper analyzes and proposes prioritization for transactional workloads in conventional DBMS. This work first conducts a detailed bottleneck analysis of resource usage by transactional workloads on commercial and noncommercial database systems (IBM DB2, PostgreSQL, Shore) under a variety of configurations. Our first contribution is a demonstration that for TPC-C workloads, under all of the above DBMS, transaction execution times are dominated by time spent waiting on locks, whereas for TPC-W workloads, CPU largely dominates transaction execution times. The second component of this work is an implementation and evaluation of several preemptive and non-preemptive prioritization algorithms in PostgreSQL and Shore. The primary contribution is a demonstration that transaction prioritization can provide 3x improvement for high-priority transactions in general-purpose DBMS. Furthermore, despite evaluating a wide-range of scheduling algorithms, we find that particularly simple scheduling policies are most effective in improving high-priority without significantly penalizing low-priority transactions.

1 Introduction

Online transaction processing (OLTP) is a mainstay in modern commerce, banking, and Internet applications, and is found increasingly in other applications such as software fault tracking [7] and discussion boards [20]. Access to dynamic web content can often be orders of magnitude slower than static content due to the dominating cost of database activity. Even worse, given the limited resources of the DBMS, transactions are even more costly during periods of high load, because requests spend more time waiting for critical resources. Since the competitor is "only a click away," the user must be kept satisfied with fast access times.

To alleviate the problem of costly database accesses, it can be extremely valuable to assign priorities to users and provide differing levels of performance. When both high- and low-priority clients share the database system, high-priority clients should complete more quickly on average than their low-priority counterparts. For example, an online merchant may make use of priorities to provide better performance to new prospective clients, or to big spenders expected to generate large profits. Alternatively, a web journal may provide improved responsiveness to "gold-customers" who pay higher subscription costs. Finally, point-of-sales systems could also run long-running maintenance queries "in the background" at low-priority while customer purchases execute quickly at high-priority.

The goal of this research is to provide prioritization and differentiated performance classes within a conventional (general-purpose) relational database system running OLTP and transactional web workloads, including read/write transactions. This paper provides a detailed resource utilization breakdown for OLTP workloads executing on a range of database platforms including IBM DB2[14], Shore[16], and PostgreSQL[17]. IBM DB2 and PostgreSQL are both widely used (commercial and noncommercial) database systems. Shore is chosen as a research prototype open-source system which represents the major concurrency control approach used in DB2. DB2 and Shore use traditional hierarchical locking and PostgreSQL (as Oracle) uses multiversion concurrency control (MVCC) [6]. The paper also implements several transaction prioritization algorithms within the Shore and PostgreSQL DBMS. The prioritization schemes studied include non-preemptive priorities, non-preemptive priorities with priority inheritance, and various schemes for preemptive priorities. Given the focus on web and complex transactional applications, we use the benchmark OLTP workloads TPC-C and TPC-W.

The contributions of this research are:

[*]This work was supported by NSF grants IIS-0133686 and CCR-0205544 and by IBM.

[†]This work was supported by NSF Career Grant CCR-0133077, NSF ITR Grant 99-167, and by IBM, Cisco Systems, and Seagate Technologies via a grant from the Pittsburgh Digital Greenhouse.

1. Identification of the bottleneck resource under various workloads and databases. This bottleneck varies across different databases, workloads, and loads (Section 4).

2. A demonstration that transaction prioritization can provide 3x improvement for high-priority requests in general-purpose DBMS (Section 5).

2 Prior Work

There is a wide range of well-known database research, including that of Abbott, Garcia-Molina, Stankovic, and others, which studies different transaction scheduling policies, evaluating the effectiveness of each. Most existing implementation work is in the domain of real-time database systems (RTDBMS), where the goal is not improvement of mean execution times for classes of transactions, but rather meeting deadlines associated with each transaction. These RTDBMS are sufficiently different from the general-purpose DBMS studied in this paper to warrant investigation as to whether results for RTDBMS apply to general-purpose DBMS as well. In addition to the existing implementation work in RTDBMS, there has also been work on simulation and analytical modeling of prioritization in DBMS and RTDBMS. Unfortunately, the simulation and analytical approaches have difficulty in capturing the complex interactions of CPU, I/O, and other resources in the database system.

In Section 2.1 we summarize the most relevant existing research on implementation of RTDBMS. In Section 2.2 we summarize the existing and ongoing work on prioritization in general-purpose DBMS.

2.1 Real-Time Databases

Real-time database systems (RTDBMS) have taken center stage in the field of database transaction scheduling for the past decade. These systems are useful for numerous important applications with intrinsic timing constraints, such as multimedia (*e.g.*, video-streaming), and industrial control systems. RTDBMS manage transactional timing constraints and time-based semantics in ways that general-purpose DBMS cannot [21].

RTDBMS differ from conventional DBMS with priorities. In RTDBMS, each transaction is associated with time-dependent constraints, *e.g.*, a deadline, which must be honored to maintain transactional semantics. The goal, minimizing the number of missed constraints (deadlines), requires maintaining time-cognizant protocols and various specialized data structures [21], unlike general-purpose DBMS. Scheduling issues such as priority inversion may have different costs for RTDBMS as compared with conventional DBMS: *i.e.*, a single priority inversion may cause a missed deadline while hardly affecting overall mean execution time. Lastly RTDBMS workloads can differ substantially from conventional DBMS workloads.

Abbott and Garcia-Molina [2, 1, 3, 4, 5] extensively study scheduling RTDBMS transactions in simulation. They concentrate on scheduling the critical resources (CPU, locks and I/O) to meet real-time deadlines. Within these critical resources they investigate several preemptive and non-preemptive scheduling policies. On the question of which resource needs to be scheduled, Abbot and Garcia-Molina conclude that CPU scheduling is most important, as transactions only acquire resources when they have the CPU [2]. They find scheduling concurrency control resources can also improve performance.

With respect to scheduling algorithms, both Abbott and Garcia-Molina[2] and Huang et al. [12] address priority inheritance and preemptive prioritization in RTDBMS. Both papers study similar systems, using two-phase locking protocols, and propose to solve the priority-inversion problem. Abbott and Garcia-Molina find that priority inheritance is important when ensuring that deadlines are met, in particular when the database is small. In contrast, Huang et. al. find that standard priority inheritance is not very effective in RTDBMS.

Kang et al. [13] differentiate between classes of real-time transactions, providing different classes with QOS guarantees on the rate of missed deadlines and data freshness. They focus on main memory databases.

The results in this paper will differ from those above as follows: (i) We do not find that CPU is the uniquely important resource to schedule. In fact for DBMS with traditional locking and TPC-C workload we see that scheduling the lock queues is far more effective than CPU scheduling. (ii) We do not find that priority inheritance is necessarily helpful. For certain workloads and DBMS it appears ineffective. We attribute these differences in results to the many differences between real-time and conventional database systems and workloads.

2.2 Priority Classes

When the goal is to establish priority classes for mean performance (rather than meeting specific deadlines), existing work can be divided into techniques which schedule transactions (i) outside the DBMS and (ii) inside the DBMS. External scheduling is typically implemented using admission control to prevent transactions from entering the DBMS. Internal scheduling schedule transactions as they execute within the database.

The admission control approach is based on the principle that DBMS performance often declines as the number of executing transactions and the demand on system resources and data contention (locking) increase. Admission control deals with these problems by limiting the number of transactions executing inside the DBMS by holding transactions “outside of the database” until the load subsides.

Recent work at IBM implements priority classes in admission control [15]. The approach makes admission control decisions based not only on the number of transactions in the DBMS, but also on transaction priorities, by limiting the number of low-priority transactions that are able to interfere with high-priority transactions. Such admission control also limits inefficiencies introduced when the system is under overload, such as virtual memory paging and thrashing. Consequently, high-priority transactions under overload can benefit significantly.

We believe that prioritization based on admission control cannot fully exploit transaction scheduling to improve system performance. The fundamental problem is that admission control has limited information about resource utilization at the database. Scheduling transactions within the DBMS, however, has numerous benefits: The database system can easily parse and generate query plans for transactions, allowing more intelligent decisions to be made, including analysis to determine which resources transactions are most likely to contend for, if any. Only the DBMS itself can actively make decisions using the detailed state of the system and resource allocations (*i.e.* pending I/O requests and granted locks).

There is much room for further research in transaction scheduling internal to the DBMS. The most pertinent work in this area is by Carey et al. [9], who study our same fundamental problem — priority scheduling within a DBMS to improve the performance of high-priority transactions. They argue that transaction prioritization can be implemented by scheduling bottleneck ("critical") DBMS resources, and proceed to study the performance of several priority algorithms in simulation. They assume a read-only workload, but recommend that mixed read/write workloads be examined in the future. In this paper our workloads are mixed read/write and our work is an implementation rather than a simulation study.

There is little research in implementations of prioritization in conventional DBMS. As a testament to the importance of the problem, both IBM DB2 [14] and Oracle [18] offer CPU scheduling tools for prioritizing transactions. Experimentally, we find that, for our workloads, these tools provide nowhere near the improvement offered by the prioritization schemes implemented in this paper.

3 Experimental Setup

This section describes experimental setup details including the workloads, hardware, and software used.

3.1 Workloads

As representative workloads for OLTP and Transactional web applications, we choose the TPC-C [10] and TPC-W [11] benchmarks. TPC-C and TPC-W are designed as *closed loop systems*, with a fixed number of clients connected to the database that alternatively wait and execute transactions against the database. The time spent waiting between transactions is known as the *think time*, and models interactive clients who take time to interpret results. Varying the think time changes the amount of contention and load on the system by changing per-client transaction inter-arrival times; shorter think times result in a greater number of active (non-thinking) clients, and longer think times result in fewer active clients.

A goal of this work is to evaluate prioritization under a range of levels of concurrency (number of active clients). The level of concurrency in the database can be adjusted by either adjusting the think time, or by holding think time fixed and increasing the number of clients connected to the database. We find that both methods yield very similar results. For all our resource breakdown graphs (see Section 4), we will express our results as a function of varying the number of active clients connected to the database (and holding the think times fixed at zero). For the prioritization experiments (see Section 5), we will instead sometimes prefer to look at a range of think times, because varying the number of clients (active or non-active) affects other system overheads such as context-switching and swapping which sometimes leads to undesirable experimental variability.

For both TPC-C and TPC-W, a priority is assigned to each transaction, chosen according to a Bernoulli trial with probability 10% of being assigned high-priority and 90% of being assigned low-priority. In practice the ratio of high-priority to low-priority transactions may vary.

The TPC-C workload in this study is generated using software developed at IBM. We modify TPC-C slightly to remove the restriction that individual clients must always access the same warehouse and district. This change produces a more uniform access pattern to the database when the number of clients is larger than that dictated by the TPC-C specification. The TPC-W workload is generated using the publicly available TPC-W Kit from PHARM [8]. Minor modifications to TPC-W are made to improve performance, including rewriting the connection pooling algorithm to minimize overhead.

3.2 Hardware and DBMS

All of the TPC-C experiments for DB2 and Shore are performed on a Pentium 4, 2.2-GHz Why equipped with 1 GB of main memory and two hard drives, one 120 GB IDE drive and a 73 GB SCSI drive. The TPC-C PostgreSQL experiments are conducted on a comparable machine with 2 1-GHz processors and 2 GB of RAM, allowing us to handle the larger memory requirements of PostreSQL. The results for PostgreSQL on the dual-processor machine are similar to those when performed on the single-processor 2.2-GHz machine used by DB2 and Shore. The TPC-W experiments are all conducted with the database running on the 2.2-GHz machine; the web server and Java servlet engine run on a Pentium III, 736Hz processor with 512 MB of main memory; and the client applications run on two other machines. The operating system on all machines is Linux 2.4.17.

The DBMS we experiment with are IBM DB2 [14] version 7.1, PostgreSQL [17] version 7.3, and Shore [16] interim release 2. Several modifications are made to Shore, to improve its support for SIX locking modes, and to fix minor bugs experienced in transaction rollbacks.

4 The Bottleneck Resource

Given the complexity of modern database systems, it is extremely difficult to implement prioritization for every resource. Identification of the *bottleneck resource* (which dominates transaction execution times) is essential for maximizing the benefits of prioritization.

We start with a description of the model used for breaking down transaction resource utilization and then describe breakdown results on a variety of databases and workloads under a range of configurations.

4.1 DBMS Resources: CPU, I/O, Locks

Given that our goal is improvement of individual transaction execution times, and not overall system throughput, it is important to break down the execution times from the point of view of a transaction. We focus on three fundamental DBMS resources: CPU, I/O, and Locks, chosen since they are controlled directly by the database, and are believed to be important in performance [2].

We denote by T_{Trans} the total execution time ("wall clock time") of a transaction, *i.e.*, the difference between its arrival time, T_{BEGIN}, and its completion time (COMMIT or ROLLBACK), T_{END}. T_{Trans} is broken into components representing execution time spent using each resource respectively: T_{CPU}, T_{IO}, and T_{Lock}.

The time spent using CPU resources, T_{CPU}, consists of the time transactions spend executing (T_{CPURun}) and waiting to execute on a CPU ($T_{CPUWait}$). The DBMS in this study use generalized processor-sharing CPU scheduling, resulting in T_{CPURun} and $T_{CPUWait}$ being interleaved, as the system context switches between processes and threads. Time spent using the CPU for tasks not directly on behalf of a specific transaction (*e.g.*, cache management, networking), denoted by T_{CPUSys}, is not considered in this study.

Time spent using I/O resources, T_{IO}, is accumulated when transactions initiate and block until I/O activity completes (*i.e.* reading/writing a database page). This time is almost exclusively the time spent waiting for I/O requests to complete, denoted by T_{IOWait}. Time spent performing asynchronous I/O, denoted by $T_{IOASync}$, is not charged to T_{Trans}, even though it may be initiated (directly or indirectly) by a transaction. Additionally, I/O activity performed by the operating system, such as virtual memory paging, is not included in the I/O time.

Lock resources, as considered in this study, are more unconventional, warranting further detail. Database locks are broken into "heavyweight" and "lightweight' locks. Heavyweight locks are used for logical database objects, to ensure ACID properties for the database. Lightweight locks, include spinlocks and mutexes used to protect data internal to the database engine. This study assumes that lightweight locking is never a bottleneck, and considers only heavyweight locking. We use the term "locks" to refer to heavyweight locks throughout.

T_{Lock} denotes the time transactions spend on locks. This is broken into the time spent holding the lock $T_{LockHold}$ and the time spent waiting for the lock $T_{LockWait}$. When a transaction holds a lock, it uses or waits for other resources in the system. As a result, we consider T_{Lock} to be made up exclusively by $T_{LockWait}$.

In summary, we make the following assumptions:

$$\begin{aligned} T_{Trans} &\equiv T_{CPU} + T_{IO} + T_{Lock} \\ T_{CPU} &\equiv T_{CPUWait} + T_{CPURun} \\ T_{IO} &\equiv T_{IOWait} \\ T_{Lock} &\equiv T_{LockWait} \end{aligned}$$

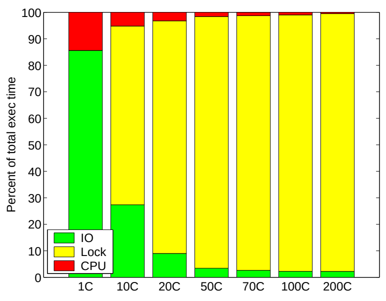
4.2 Breakdown procedure

To identify the bottleneck resource, for each experiment we divide the execution time of transactions into three components: CPU, I/O, and time waiting on locks (*LockWait*). Real world databases such IBM DB2 and PostgreSQL provide limited built-in facilities for measuring resource usage. In the case of DB2, since source code is unavailable, we rely on the DB2 snapshot and event monitoring [14] functionality. For PostreSQL and Shore DBMS we implement custom measurement functionality by instrumenting the DBMS itself. We compute the total CPU, I/O and LockWait over all transactions in a given experiment and then determine the fraction each component makes up of the sum of all execution times.

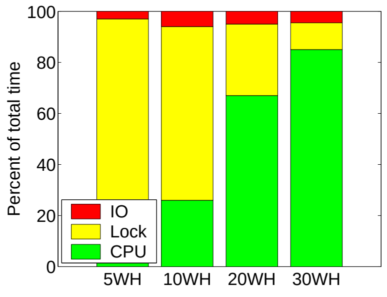
For IBM DB2 and PostgreSQL, which follow a process-based architecture, we also verify the resulting breakdown at the operating system level via the Linux `vmstat` command. Specifically we record the fraction of time that database processes spend in the CPU run queue (TASK_RUNNING state), blocked on I/O (TASK_INTERRUPTIBLE state), or waiting for locks (TASK_UNINTERRUPTIBLE state).

4.3 Breakdown results

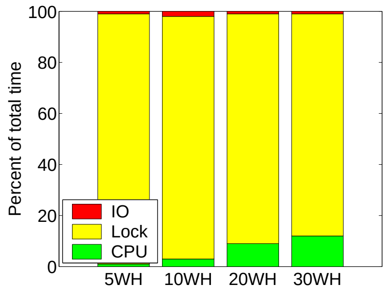
Central to the thesis of this work is the idea that understanding a workload's resource utilization is essential for implementing priorities. Towards this end, we construct resource utilization profiles for TPC-C under IBM DB2,



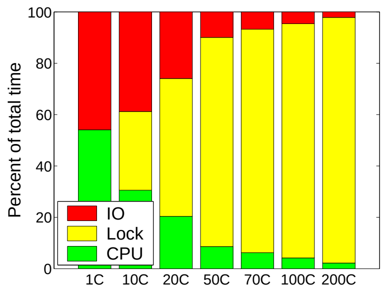
(a) **DB2**: Varying Clients, 10 Warehouses



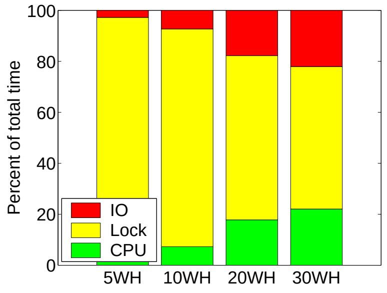
(b) **DB2**: 10 Clients, Varying Warehouses



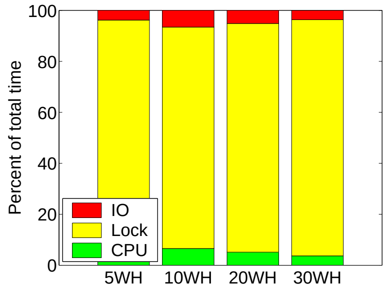
(c) **DB2**: Standard Scaling (10 clients per WH)



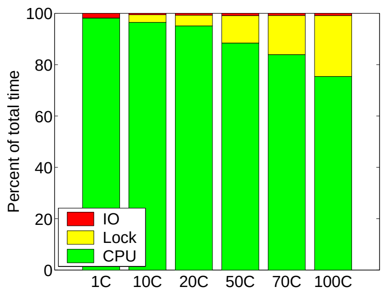
(d) **Shore**: Varying Clients, 10 Warehouses



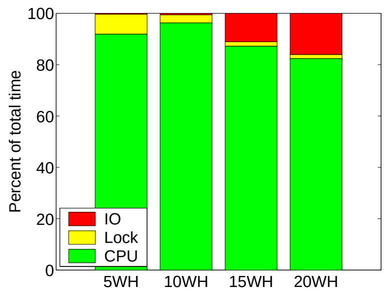
(e) **Shore**: 10 Clients, Varying Warehouses



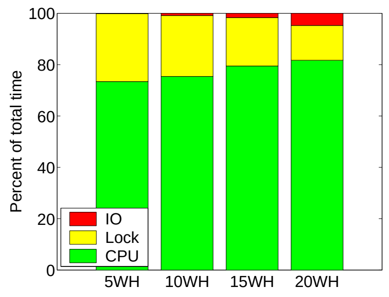
(f) **Shore**: Standard Scaling (10 clients per WH)



(g) **PostgreSQL**: Varying Clients, 10 Warehouses



(h) **PostgreSQL**: 10 Clients, Varying Warehouses



(i) **PostgreSQL**: Standard Scaling (10 clients per WH)

Figure 1: Resource breakdowns for TPC-C transactions under varying databases and configurations. The first row shows DB2; the second row shows Shore; and the third row shows PostgreSQL. Figures 1(a), 1(d), 1(g) show the impact of varying the level of concurrency in the database by increasing the number of active clients. Figures 1(b), 1(e), 1(h) show the impact of varying the size of the database (number of warehouses) while holding the number of clients fixed. Figures 1(c), 1(f), 1(i) reflect configurations that scale the concurrency and database size simultaneously so that the number of clients is always 10 times the number of warehouses, in accordance with TPC-C specifications.

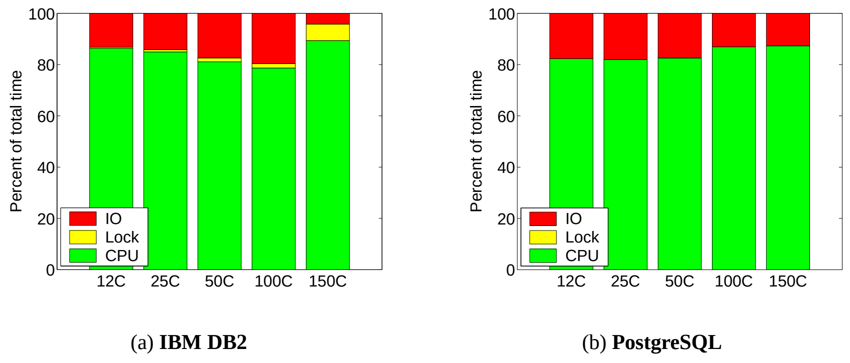


Figure 2: Resource breakdowns for TPC-W transactions under IBM DB2 and PostgreSQL. Since there does not exist a TPC-W implementation for Shore, there is no breakdown graph for Shore. Observe that DB2 sees very little locking and PostgreSQL sees virtually none. Observe that for TPC-W, CPU is the dominant resource.

PostgreSQL, and Shore, and TPC-W under IBM DB2 and PostgreSQL. We examine how these profiles change under different levels of concurrency and database sizes.

Figure 1 shows the resource breakdowns measured for TPC-C under IBM DB2, PostgreSQL, and Shore, depicting the average portions (indicated as percentages) of transaction execution time is due to CPU, I/O, and Lock resource usage. Although these show breakdowns normalized to 100%, there is a small measurement error, less than 10%. We attribute this error to (i) external system load and (ii) high granularity I/O and CPU measurements.

For each DBMS, Figure 1 presents three sets of results to demonstrate the most significant trends. In the first column, the database size is held constant, and the number of clients connected to the database is varied to vary concurrency. In the second column, the number of clients is held constant at 10 (with zero think times), and the size of the database is varied by increasing the number of warehouses. For large databases (large number of WH), this column differs significantly from TPC-specifications, as the number of clients is far smaller than intended. In the third column, the number of clients and the database size are scaled together as specified by the TPC-C specifications, to demonstrate breakdowns for standard, *"realistic"* configurations.

The first thing to observe in Figure 1 is that the first and second rows, corresponding to DB2 and Shore respectively, are quite similar. This is expected due to the fact that both databases use traditional locking. In both DB2 and Shore under TPC-C it is clear that locks are the bottleneck resource, for a wide range of configurations. Generally as the level of concurrency increases (see 1st column and look in the direction of more active clients) locks become increasingly important. As the size of the databases increases (see 2nd column), locks become less important, since clients are less-likely to contend for data. While both I/O and CPU grow at similar rates in Shore as the size of the database increases, for DB2 the growth in the CPU component outpaces the growth in I/O, resulting in a CPU bottleneck in these outlying cases. From the 3rd column, which shows the standard case, it is clear that locks are the bottleneck resource for Shore and DB2.

By contrast, looking at PostgreSQL results (shown in the third row of Figure 1), we see that CPU is always the dominating resource. As the concurrency level is increased (see 1st column), LockWait increases, but CPU remains the bottleneck. As the size of the database grows (see 2nd column), I/O becomes increasingly important, but it's still not the bottleneck. The stark difference between the PostgreSQL resource breakdowns under TPC-C and those for Shore and DB2 may be attributed to the fact that PostgreSQL uses MVCC [6], which inherently requires much less locking. For the PostgreSQL experiments, Figures 1(h) and (i) the case of 30 WH is not presented due to insufficient RAM.

Each breakdown presented in Figure 1 is an average computed over all transactions in an experimental run, and as such, may not be representative of any particular, or even most transactions. The breakdowns can be sharply skewed by a small fraction of exceptional transactions with extremely long execution times. Thus, the breakdowns should be taken as an indicator of the relative importance of the resources while minimizing average transaction execution times.

Figure 2 shows resource breakdowns for TPC-W transactions under IBM DB2 and PostgreSQL as a function of the number of active clients connected to the database. The size of the database is held constant and is representative of a database used by ten clients according to the TPC-W specification, examining extremely high contention. PostgreSQL and DB2 see little, if no locking, as TPC-W intrinsically has very little data contention. The MVCC used in PostgreSQL can usually obviate the need for lock waits. Even DB2's traditional locking waits in-

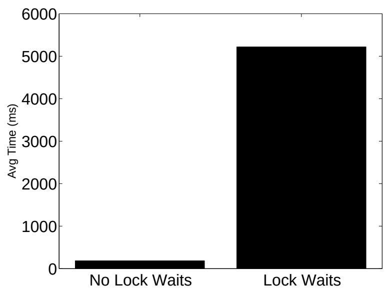


Figure 3: Comparison of execution times for transactions that never waited for locks versus those that waited for locks under Shore under TPC-C. Given that a transaction waits on a lock, its expected execution time is 28.5 times longer than if it did not.

frequently. I/O costs are also low, as the databases are small relative to main memory. Thus, CPU is the dominant (bottleneck) resource.

In another experiment, under TPC-W, we purposely reduce the size of the database well below that prescribed by the TPC-W specification, and varied the concurrency. Since clients are forced to access a smaller database, data contention increases dramatically. For this extreme experiment, we find that locking represents a significant portion of resource usage (between 40% and 60% of total execution time). The point is that even TPC-W workloads under sufficiently high load will experience behavior similar to that under TPC-C, despite the simplicity of the TPC-W transactions.

5 Scheduling the bottleneck

As seen in Section 4, the bottleneck resource for TPC-C under Shore and DB2 is LockWait, suggesting prioritization by scheduling the lock queues. Figure 3 illuminates this point, showing that waiting on a lock can increase execution time by a factor of almost thirty. For the TPC-W workload under any DBMS and for PostgreSQL, CPU is the bottleneck resource and lock time is almost nonexistent, suggesting prioritization of the CPU.

Thus throughout we examine *both* lock and CPU scheduling for *both* TPC-C and TPC-W. However, we have reservations about the TPC-W workload, and believe it is too simplistic for two reasons: (i) TPC-W transactions are extremely simplistic, and (ii) TPC-W transactions need very little concurrency control. The TPC-C workload, with more complex transaction interactions, is in fact more representative of real-world applications. Even TPC-W like workloads, under sufficiently high loads, will experience resource breakdown utilizations more similar to that of TPC-C.

We begin by defining the specific scheduling policies which we will explore.

5.1 Definition of the Policies

Our scheduling policies are divided into lock scheduling and CPU scheduling policies:

Lock scheduling policies We consider non-preemptive scheduling policies where the transaction holding the lock cannot be preempted (forced to release the lock) until it releases the lock willingly. Subsequently, we consider preemptive policies in which transactions holding locks can be forced to release them. Preemption typically involves aborting, rolling back, and resubmitting the transaction, increasing work for the DBMS.

The simplest non-preemptive policy, `NP-LQ`, simply reorders the transactions waiting in the lock queue, granting locks to high priority transactions before those of low-priority. The `NP-LQ` policy has two problems in theory: (i) priority-inversions, where a high-priorty transaction waits for a low-priority transaction to release a lock, are ignored, and (ii) although a high-priority transaction moves ahead in the *queue*, it still waits for the transaction(s) *holding* the lock to complete (known as “excess time” in queueing theory).

The `NP-LQ-Inherit` policy is a non-preemptive policy to fix problem (i). It raises the priorities of transactions from low- to high-priority when a high-priority transaction is forced to wait on one of low-priority. This is often known as “priority-inheritance” [19].

The `P-LQ` policy is a preemptive policy to fix problem (ii). When a high-priority transaction needs a lock held by a low-priority transaction, the low-priority transaction is aborted. In practice, for systems with traditional locking, it is common for the high-priority transaction to have to wait for the low-priority transaction’s rollback phase to complete before it can acquire the lock. This can reduce the effectiveness of preemption. Furthermore, the extra work created by preemption also reduces its potential effectiveness, potentially slowing down other transactions.

CPU scheduling policies We do not distinguish between preemptive and non-preemptive policies since all of our systems use (preemptive) generalized processor sharing (GPS).

We start with the simplest algorithm: `CPU-Prio`. Here the CPU resource is scheduled via weighted GPS, where high-priority transactions are given higher priority at the CPU than low-priority transactions. Specifically, under PostgreSQL and DB2 our implementation of `CPU-Prio` assigns UNIX priority nice level -20 to high-priority and $+20$ to low-priority transactions. Under Shore, threads for high-priority transactions are assigned a higher Sthread priority (`t_time_critical`) and low-priority transactions are assigned regular Sthread priority.

Although the `CPU-Prio` policy schedules CPU, not locks, it is clear that this policy might create a problem of priority inversion due to locks. Consider the situation where a high-priority transaction is blocked waiting for a lock held by a low-priority transaction. It is desirable to increase the CPU priority of the low-priority transaction so that it can more quickly complete.

To remedy this problem, `CPU-Prio-Inherit` schedules CPU according to `CPU-Prio`, however allows a low-priority transaction which is holding a lock desired by a high-priority transaction to boost its CPU priority to that of the high-priority transaction.

Organization of remaining sections In Section 5.2 we discuss simple priority schemes which require no preemption and no inheritance: `NP-LQ` and `CPU-Prio` defined above. In Section 5.3 we discuss priority schemes involving priority inheritance: `NP-LQ-Inherit` and `CPU-Prio-Inherit`. In Section 5.4 we discuss priority schemes which preempt the lock priorities: `P-LQ`.

5.2 Simple scheduling

The simple scheduling policies with no priority inheritance and no lock preemption, `NP-LQ` and `CPU-Prio`, exhibit striking differences depending on the workload and the DBMS. Figures 4 and 5 highlight these differences, showing the performance of high- and low-priority transactions using the policies for TPC-C and TPC-W workloads respectively. In all results, the concurrency varies on the x-axis, from high levels of concurrency on the left to low concurrency on the right. Concurrency is controlled either by varying think time (more natural for TPC-C) or equivalently varying the number of clients (more natural for TPC-W).

The optimal simple scheduling policy for TPC-C depends on the DBMS. Under Shore with TPC-C (see Figure 4(a)) `NP-LQ` is much more effective than `CPU-Prio` at improving high-priority transactions, under high load (when think time is less than 4000ms). In fact, simple lock queue reordering (`NP-LQ`) improves execution time of the high-priority transactions by a factor of 3. Under low loads, CPU and Lock scheduling become approximately equivalent. The penalty to the low-priority jobs under both `NP-LQ` and `CPU-Prio` is negligible and tracks the "Default" no priority setting (see Figure 4(b)). Lock scheduling is extremely effective under Shore because locks dominate transaction execution times under Shore's traditional locking protocol.

By contrast, under PostgreSQL with TPC-C (see Figure 4(c)), lock scheduling is not as effective as CPU scheduling. As the think time ranges from 5 to 25 seconds in these experiments, the number of running clients drops from 50 where locks are the dominating resource, to 20, where locks and CPU both dominate. Under high loads, `NP-LQ` scheduling improves high-priority execution times by a factor of 1.3, whereas `CPU-Prio` achieves 1.6x improvement. The penalty to the low-priority jobs under both `NP-LQ` and `CPU-Prio` is again largely negligible (see Figure 4(d)).

CPU scheduling is more effective than lock scheduling in PostgreSQL due to the way PostgreSQL implements locks. Instead of waiting on locks, transactions wait for the holders of locks to complete (these protocols are nearly equivalent under standard Two-Phase Locking protocols). When granting locks, PostgreSQL wakes all of the waiting transactions and lets them "race" to acquire the lock. The first transaction(s) to try to acquire the lock is granted it. Since transactions can only obtain locks when given the CPU, and high-priority transactions are more likely to get the CPU under `CPU-Prio`, CPU prioritization effectively prioritizes locks as well.

For TPC-W, we saw that locks are never the bottleneck resource (see Figure 2). Thus it does not seem likely that lock scheduling will be effective. Figure 5 shows the results of prioritization under TPC-W as a function of the number of active clients. As expected, `NP-LQ` does not significantly improve performance of the high-priority transactions. However `CPU-Prio` dramatically improves performance of high-priority transactions, by a factor of up to 5 under high load (high number of active clients). As in the case of TPC-C, the low-priority transactions are not significantly penalized.

Observe that for both algorithms `NP-LQ` and `CPU-Prio`, under all DBMS and workloads studied, the low priority transactions on average are not significantly penalized. This result is important, and consistent with theoretical expected results: Performance of a small class of high-priority transactions can be improved without harming the overall performance of low-priority transactions.

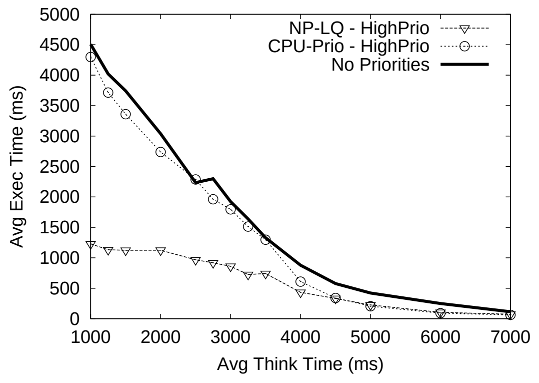
5.3 Priority inheritance

In this section we evaluate the two algorithms for priority inheritance: `NP-LQ-Inherit` and `CPU-Prio-Inherit`, which are extensions of the `NP-LQ` and `CPU-Prio` policies respectively.

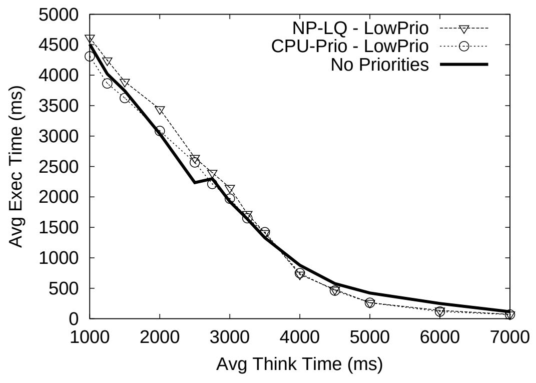
Figure 6 compares the policies `NP-LQ-Inherit` and `NP-LQ` for the Shore DBMS under TPC-C for a range of concurrency levels. We find that there is no significant benefit to adding priority inheritance to the `NP-LQ`.

For PostreSQL under TPC-C, adding priority inheritance to `NP-LQ` offers no appreciable gain in performance, however priority inheritance when combined with CPU scheduling is beneficial. Inheritance of CPU priority (Figure 7) improves high-priority transactions by a factor of 6, doubling the improvement of `CPU-Prio`. This significant leap in performance of `CPU-Prio-Inherit` over `CPU-Prio` suggests that priority-inversion is a common problem in PostgreSQL under `CPU-Prio`.

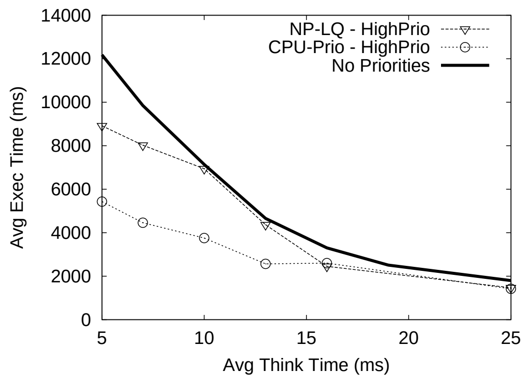
For TPC-W, recall from Section 5.2 that CPU scheduling (`CPU-Prio`) is more effective than `NP-LQ`. Thus Figure 8 compares the policies `CPU-Prio-Inherit` to `CPU-Prio` for the TPC-W workload on PostgreSQL.



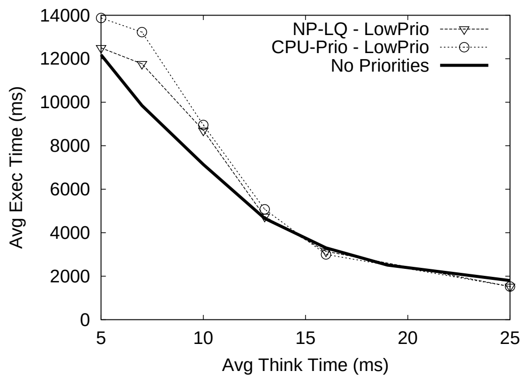
(a) **Shore** High-Priority



(b) **Shore** Low-Priority



(c) **PostgreSQL** High-Priority



(d) **PostgreSQL** Low-Priority

Figure 4: Mean execution times for high- and low-priority TPC-C transactions under non-preemptive scheduling strategies `NP-LQ` and `CPU-Prio`. We see that `NP-LQ` is most effective under Shore, whereas `CPU-Prio` is most effective under PostgreSQL. In the above graphs concurrency level (load) increases as think time goes down.

We find that there is no improvement for `CPU-Prio-Inherit` over `CPU-Prio`. This is to be expected given the low data contention found in the TPC-W workload; priority inversions can only occur during data contention.

Results for low-priority are not shown, but as in Figure 4, low-priority transactions are only negligibly penalized on average.

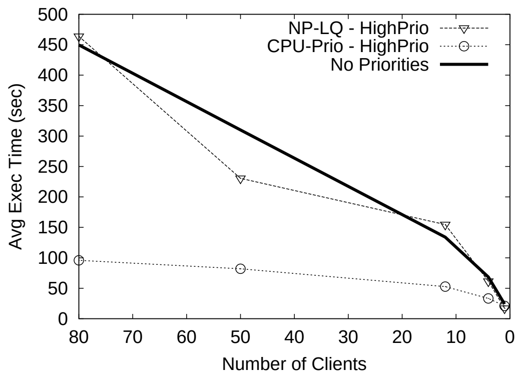
5.4 Preemptive scheduling

Under non-preemptive lock scheduling (`NP-LQ` and `NP-LQ-Inherit`), high-priority transactions can only move ahead of low-priority transactions waiting in the lock queue. A high-priority transaction cannot preempt a low-priority transaction (potentially) holding a lock.

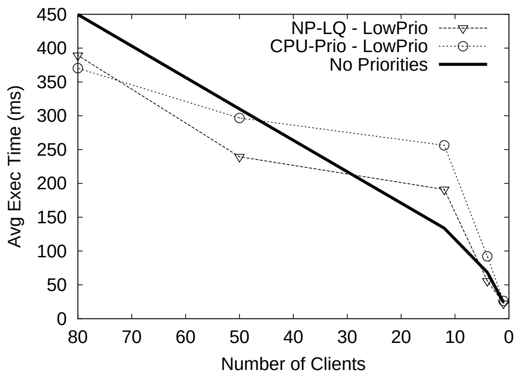
Figure 9 shows the average length of lock queues in Shore and PostgreSQL for TPC-C workloads under a range of concurrency. The queues themselves are fairly short (1.5–4 transactions) under all levels of concurrency examined. The `NP-LQ` and `NP-LQ-Inherit` policies improve performance by allowing a transaction to skip ahead of these few transactions in the queue.

The goal of preemptive lock prioritization (`P-LQ`) is to eliminate the time high-priority transactions spend waiting for *current low-priority lock holder(s)* to release the lock. Surprisingly, we find that `P-LQ` is a rather poor policy for our workloads and goals. Figure 10 compares the performance of `P-LQ` against the most effective of our non-preemptive algorithms for Shore and PostreSQL under the TPC-C workload. For Shore, we compare `P-LQ` to `NP-LQ` and find high-priority transactions benefit at most 30%, however the penalty to low-priority transactions is devastating. Low-priority transactions may be penalized by a factor of up to 5. For PostreSQL, we compare `P-LQ` to `CPU-Prio-Inherit` and find no additional benefit for high-priority transactions. Low-priority transactions suffer by about about 25% under `P-LQ`.

The reason why low priority transactions suffer so



(a) **PostgreSQL** High-Priority



(b) **PostgreSQL** Low-Priority

Figure 5: Mean execution times for high- and low-priority TPC-W transactions under non-preemptive scheduling strategies `NP-LQ` and `CPU-Prio`. We see that `CPU-Prio` improves mean execution time for high-priority requests by a factor of up to 5. In the above graphs, the concurrency level (load) increases as the number of clients goes up.

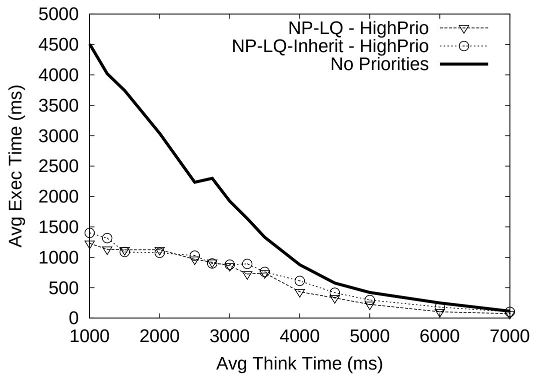


Figure 6: Evaluation of priority inheritance policy `NP-LQ-Inherit` for Shore under TPC-C. Adding priority inheritance to reordering of lock queues fails to improve performance significantly. PostgreSQL shows similar results. The concurrency level (load) increases as the average think time goes down.

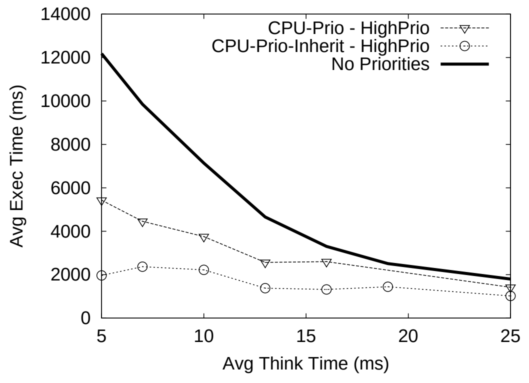


Figure 7: Evaluation of priority inheritance, combined with CPU scheduling, policy `CPU-Prio-Inherit` under PostgreSQL under the TPC-C workload. `CPU-Prio-Inherit` can be a very effective tool for PostgreSQL's MVCC-based system.

much under preemptive policies is that preemption has a cost. The preempted transaction must be aborted and rolled back and restarted, introducing extra work. This overhead is more severe under Shore than under PostreSQL, which uses MVCC [6], where abortion and rollback costs are low.

TPC-W results for `P-LQ` are omitted because lock contention is so low; thus lock scheduling is insignificant.

Future extensions to Preemptive Priorities Under both the TPC-C and TPC-W workloads in each DBMS, preemptive lock scheduling (`P-LQ`) is no better than the best non-preemptive policies. However, the preemptive policy `P-LQ` is rather simplistic: high-priority transactions preempt any low-priority transactions in their way. Alternatively, since the cost of waiting for the lock may be cheaper than the overhead of preempting the low-priority transaction, there is room for improvement over this "bulldozing" approach to preemption.

We explore a few other preemptive algorithms, that are more selective in the decision to preempt. These algorithms predict a victim transaction's remaining life expectancy and the cost of rolling back the victim to determine whether to preempt or wait. The predictors we examine are (i) the number of locks held of the victim and (ii) the "wall-clock" age of the victim. Although preliminary, we find these are both poor predictors of both

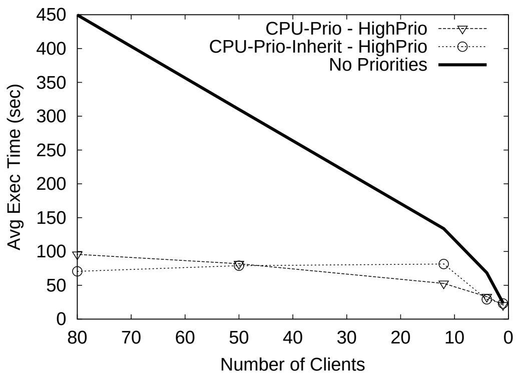
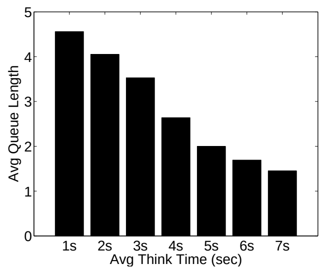
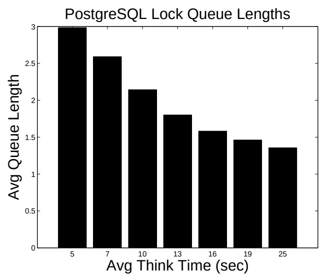


Figure 8: Comparison of `CPU-Prio-Inherit` to `CPU-Prio` under PostgreSQL for the TPC-W workload. The graph shows no additional benefit for priority inheritance over simple CPU scheduling.



(a) Shore



(b) PostgreSQL

Figure 9: Average lock queues for TPC-C are extremely short. Thus reordering lock queues may not suffice.

the rollback costs and life-expectancy of our transactions.

6 Conclusion

In this paper we provide an implementation of prioritization of differentiated performance classes within a conventional (general-purpose) relational database running TPC-C or `TPC-W` workloads.

To do this, we start by identifying the bottleneck resource at which the prioritization scheduling should take place. We study the lifetime of a transaction and divide it into three components: CPU time, I/O time, and time spent waiting for locks. The results are clearly broken down by workload and concurrency control mechanism. The bottleneck resource for DBMS using traditional locking (such as Shore and DB2) under TPC-C like workloads is *lock waiting time*, across a wide range of configurations. By contrast, the bottleneck resource for DBMS using either MVCC or running under `TPC-W` like workloads (which have less data contention) is *CPU*.

The above bottleneck analysis provides a roadmap for choosing prioritization algorithms. We experiment with lock scheduling algorithms and CPU scheduling algorithms. We consider non-preemptive policies, policies with priority inheritance, and preemptive policies. As above, the results are clearly broken down by workload and concurrency control mechanism.

For DBMS using traditional locking such as Shore and (we believe) by extension DB2, under TPC-C like workloads, we find that simple non-preemptive lock scheduling (`NP-LQ`) is most effective. High-priority transaction execution times drop by a factor of 3, while low-priority transactions are hardly penalized. Adding priority inheritance or preemption does not appreciably help, and preemption especially penalizes low-priority transactions.

For DBMS using MVCC (with TPC-C or TPC-W workloads) and for TPC-W workloads (with any concurrency control mechanism), we find that lock scheduling is largely ineffective (even preemptive lock scheduling) and CPU scheduling is highly effective. For example, we find that for PostgreSQL running under TPC-C, the simplest CPU scheduling algorithm `CPU-Prio` provides a factor of 2 improvement for the high-priority transactions, and adding priority inheritance (`CPU-Prio-Inherit`) brings this up to a factor of near 6 improvement under high loads, while hardly penalizing low-priority transactions. For PostgreSQL running under the TPC-W workload, we find that the best scheduling algorithm is the simplest CPU scheduling policy `CPU-Prio`, which improves performance for high-priority transactions by a factor of up to 5. The reason why inheritance is more effective for the TPC-C example above is that TPC-C has much more data contention than TPC-W, leading to more priority inversions.

In conclusion, our results suggest that (i) knowledge of the bottleneck resources is important for determining the best scheduling policies, and (ii) priority scheduling at the bottleneck resource using simple algorithms can yield significant performance improvements for both TPC-C and TPC-W workloads on real general-purpose DBMS.

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: A performance evaluation. In *Proceedings of Very Large Database Conference*, pages 1–12, 1988.

[3] 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.

[4] 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.

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

[6] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *TODS*, 8(4):465–483, 1983.

[7] Bugzilla. www.bugzilla.org.

[8] Trey Cain, Milo Martin, Tim Heil, Eric Weglarz, and Todd Bezenek. Java tpc-w implementation. http://www.ece.wisc.edu/ pharm/tpcw.shtml, 2000.

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

[10] Transaction Processing Performance Council. TPC benchmark c. Number Revision 5.1.0, December 2002.

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

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

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

[15] Erich Nahum. A method for transparent admission control and request scheduling in dynamic e-commerce web sites. Unpublished Manuscript, IBM Research Lab, May 2003.

[16] University of Wisconsin. Shore - a high-performance, scalable, persistent object repository. `http://www.cs.wisc.edu/shore/`.

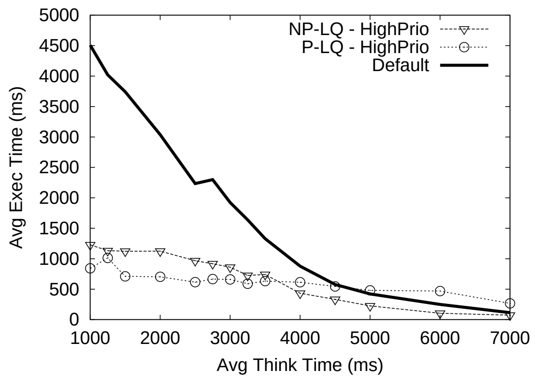
[17] PostgreSQL. `http://www.postgresql.org`.

[18] Ann Rhee, Sumanta Chatterjee, and Tirthankar Lahiri. The oracle database resource manager: Scheduling cpu resources at the application. High Performance Transaction Systems Workshop, 2001.

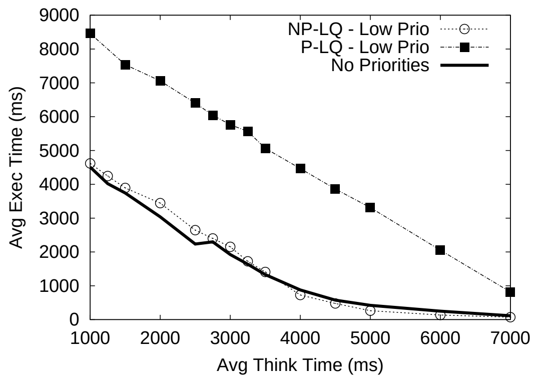
[19] L. Sha, R. Rajkumar, and J. Lehozky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[20] Slashdot. www.slashdot.org.

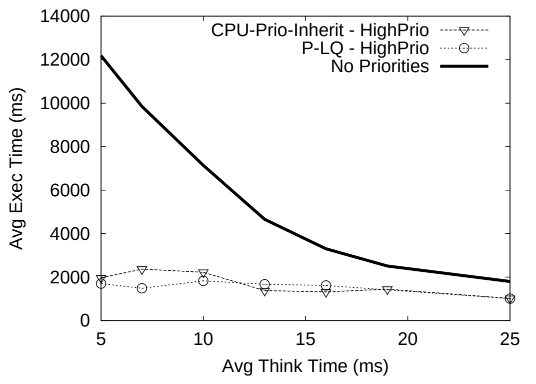
[21] John A. Stankovic, Sang Hyuk Son, and Jorgen Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32(6):29–36, 1999.



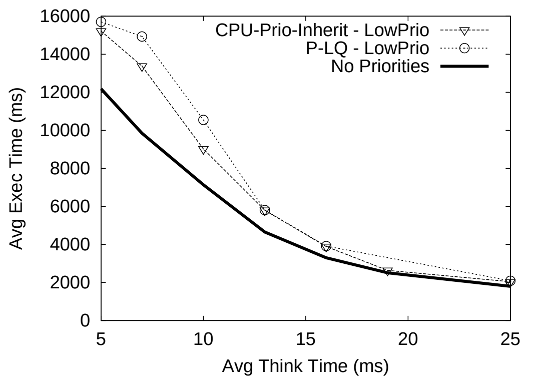
(a) **Shore** High-Priority



(b) **Shore** Low-Priority



(c) **PostgreSQL** High-Priority



(d) **PostgreSQL** Low-Priority

Figure 10: Comparison of preemptive policy `P-LQ` against the best non-preemptive policies for each Shore and PostgreSQL under TPC-C. The high-priority transactions (left column) show little improvement with `P-LQ` relative to the non-preemptive policies. The low-priority transactions (right column) suffer greatly under `P-LQ`. In the above figures, the concurrency level (load) increases as the average think time goes down.