# Priority Mechanisms for OLTP and Transactional Web Applications *

David T. McWherter Bianca Schroeder Anastassia Ailamaki Mor Harchol-Balter

## 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 traditional database systems (DBMS). This work first performs a detailed bottleneck analysis of resource usage by transactional workloads on commercial and noncommercial DBMS (IBM DB2, PostgreSQL, Shore) under a range of configurations. Second, this work implements and evaluates the performance of several preemptive and non-preemptive DBMS prioritization policies in PostgreSQL and Shore. The primary contributions of this work include (i) understanding the bottleneck resources in transactional DBMS workloads and (ii) a demonstration that prioritization in traditional DBMS can provide 2x–5x improvement for high-priority transactions using simple scheduling policies, without expense to low-priority transactions.*

## 1. Introduction

Online transaction processing (OLTP) is a mainstay in modern commerce, banking, and Internet applications. For many OLTP applications, particularly e-commerce applications, clients require fast access times. Unfortunately, serving requests which involve database activity for dynamic query processing and data generation can be very slow — orders of magnitude slower than delivering static content. This slowness is exacerbated under heavy load and overload.

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 may 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 traditional (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[16], Shore[17], and PostgreSQL[18]. IBM DB2 and PostgreSQL are both widely used (commercial and noncommercial) database systems. Shore is an open source research prototype using traditional two-phase locking (2PL), the concurrency control used in DB2. PostgreSQL (like Oracle), on the other hand, uses multiversion concurrency control (MVCC) [6]. The paper also implements several transaction prioritization policies within Shore and PostgreSQL. The prioritization policies studied include non-preemptive priorities, non-preemptive priorities with priority inheritance, and preemptive abort scheduling. Given the focus on web and complex transactional applications, we use the benchmark OLTP workloads TPC-C and TPC-W.

The primary contributions of this research are twofold:

1. Identification of bottleneck resource(s) across DBMS, workloads and concurrency levels.

2. Demonstration that simple priority scheduling inside the DBMS significantly improves high-priority transaction execution times without penalizing low-priority

*This work was benefitted from advice and code provided by Mengzhi Wang, Josef Burger, Tom Lane, and IBM. This work was supported by NSF grants IIS-0133686, CCR-0205544, CCR-0133077, and by IBM, Cisco Systems, Seagate Technologies via a grant from the Pittsburgh Digital Greenhouse and an IBM faculty award.

transactions.

With respect to bottleneck identification, we show that the bottleneck resource for TPC-C on IBM DB2 and Shore, both of which use 2PL, is lock waiting. By contrast, for the same TPC-C workload, PostgreSQL, which uses MVCC, exhibits an I/O synchronization bottleneck. For TPC-W on DB2 and PostgreSQL, we find that the bottleneck is always the CPU.

On the issue of scheduling policies, we find that scheduling of bottleneck resources results in improving high-priority transaction execution times considerably. For systems with lock bottlenecks (TPC-C on DB2 and Shore), CPU scheduling is ineffective, but lock scheduling can improve high-priority performance by a factor of 5.3. For systems with CPU bottleneck (TPC-W), lock scheduling is ineffective, while CPU scheduling improves high-priority performance by a factor of 4.5. For PostgreSQL, which has an I/O synchronization bottleneck, CPU scheduling with priority inheritance yields a factor of 6 improvement of high-priority transactions. Provided that the fraction of high-priority transactions is small, the penalty to the low-priority transactions is negligible as long as preemption is not used.

## 2. Prior Work

There is a wide range of well-known database research, including that of Abbott, Garcia-Molina, Stankovic, and others, studying different transaction scheduling policies and 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 transaction prioritization within 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. Traditional DBMS with transaction priorities differ from RTDBMS. In RTDBMS, each transaction is associated with time-dependent constraints (usually deadlines), which must be honored to maintain transactional semantics. The goal of 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 to traditional 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 traditional DBMS workloads.

Abbott and Garcia-Molina [2, 1, 3, 4, 5] extensively study scheduling RTDBMS in simulation, preemptively and non-preemptively scheduling the critical resources (CPU, locks and I/O) to meet real-time deadlines. 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 [5]. Additionally, they find scheduling of concurrency control resources also improves performance.

With respect to scheduling policies, both Abbott and Garcia-Molina [5] and Huang et. al. [14] examine priority inheritance and preemptive prioritization in RTDBMS that use 2PL, to address 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. [15] 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.

Our results will differ from those above as follows: (i) CPU is not always the most important resource to schedule. For DBMS using 2PL and TPC-C workloads we see that scheduling locks is far more effective than CPU scheduling. (ii) Priority inheritance is not always necessary, and is ineffective for some workloads and DBMS.

We attribute these differences in results to the many differences between real-time and traditional DBMS and their workloads.

### 2.2. Priority Classes

Existing work to establish priority classes for mean performance (rather than meeting specific deadlines), 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, by contrast, prioritizes transactions as they execute within the database.

Recent work at IBM implements priority classes in admission control [13]. 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 reduces lock contention and 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.

Despite the simplicity of admission control for prioritization, we believe that internal DBMS scheduling is more effective. Internal scheduling allows direct control of DBMS resources, and can utilize knowledge of query plans, transaction resource needs, and system resource availability (*e.g.* I/O requests and granted locks).

There is much room for further research in transaction scheduling internal to the DBMS. The most pertinent work, by Carey et. al. [9] is a simulation study of our same fundamental problem: evaluating priority scheduling policies within DBMS to improve high-priority transaction performance. They assume a read-only workload, but recommend mixed read/write workloads should also be examined in the future. In contrast, our work assumes mixed read/write workloads and our work uses fully implemented DBMS rather than a simulator.

Brown et. al. [7] address multiclass workloads with per-class response time goals. Again, this is a pure simulation study without experimental validation on a DBMS prototype. Moreover, it focuses on a single resource, memory, while in our work we analyze the resource breakdown for different DBMS and workloads and consider the different bottleneck resources.

Prioritization within traditional DBMS has not been a focus for academic research. As a testament to the importance of the problem, however, both IBM DB2 and Oracle provide prioritization tools (IBM DB2gov and QueryPatroller [16, 10] and Oracle DRM [19]), all of which focus on CPU scheduling. We have experimented extensively with IBM DB2gov, and find it does not provide nearly as large of a prioritization benefit for the lock-bound workloads discussed in this paper. This paper addresses a wider range of scheduling policies for both CPU and lock resources.

## 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 experiment with the TPC-C [11] and TPC-W [12] (TPC-W Shopping Mix) benchmarks.

The TPC-C workload implementation for DB2 and PostgreSQL is written and graciously donated by IBM. The TPC-C Shore implementation was written at CMU. TPC-C is modified to allow each client to access a different warehouse and district for each transaction, which produces more uniform access to the database. The TPC-W workload comes from the PHARM [8] project with minor improvements, such as an improved connection pooling algorithm.

### 3.2. Hardware and DBMS

All of the TPC-C experiments for DB2 and Shore are performed on a 2.2-GHz Pentium 4 with 1GB RAM, one 120GB IDE drive, and a 73GB SCSI drive. The TPC-C PostgreSQL experiments are conducted on a comparable machine with two 1-GHz processors and 2GB of RAM, allowing us to handle the larger memory requirements of PostgreSQL. The results for PostgreSQL on the dual-processor (two 1-GHz) 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 512MB of main memory; and the client applications run on two other machines. The operating system on all machines is Linux 2.4.

The DBMS we experiment with are IBM DB2 [16] version 7.1, PostgreSQL [18] version 7.3, and Shore [17] 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

Central to this work is the idea that understanding a workload's resource utilization is essential for effective prioritization. In order to improve high-priority transaction execution times, the *bottleneck resource*, where transactions spend the bulk of their execution time, must be scheduled, either directly or indirectly. Given the complexity of modern database systems, predicting the bottleneck resource is non-trivial.

In this section, we derive resource utilization breakdowns and determine the bottlenecks for TPC-C on Shore, DB2, and PostgreSQL and for TPC-W on DB2 and PostgreSQL. First, we describe the model used for breaking down transaction resource utilization. Next, we examine how these resource breakdowns change under varying concurrency levels and database sizes.

## 4.1. DBMS Resources: CPU, I/O, Locks

Since the goal of this paper is to improve individual transaction execution times, and not overall throughput, it is important to break down execution times from the point of view of a transaction. We focus on three core DBMS resources: CPU, I/O, and locks, chosen since they are under control of the database, and are believed to be important in performance [5].

We define the total execution time of a transaction, $T_{Trans}$, as the time from when the transaction is first submitted to when it completes. We break $T_{Trans}$ into three components, $T_{Trans} = T_{CPU} + T_{IO} + T_{Lock}$, corresponding to CPU, I/O, and locks, respectively. These components consist of just the synchronous time in which the transaction is completely dedicated to either waiting for or consuming the corresponding resource. $T_{CPU}$ consists of the time spent running on the processor and the time spent in the running state, waiting for the processor. $T_{IO}$ consists of the time spent issuing and waiting for synchronous I/O to complete (although the cost of issuing an I/O operation is negligible). $T_{Lock}$ is the time that a transaction spends waiting for database locks. Of course, time spent holding locks is accounted according to whether the transaction holding the lock is waiting for or consuming CPU or I/O or waiting for another lock.

Database locks are broken into "heavyweight" and "lightweight" locks. Heavyweight locks are used for logical database objects, to ensure the database ACID properties. Lightweight locks include spinlocks and mutexes used to protect data structures in the database engine (such as lock queues). We find that lightweight locking is not a significant component of transaction execution times in either Shore or IBM DB2. PostgreSQL, however, has significant lightweight lock waiting, due to an idiosyncrasy of the PostgreSQL implementation. We find almost all lightweight locking in PostgreSQL functions to serialize the I/O bufferpool and Write-Ahead-Logging activity (via the `WALInsert`, `WALWrite`, and `BufMgr` lightweight locks). As a result, we attribute all the lightweight lock waiting time for the above-listed locks to I/O. We use the term "locks" throughout the remainder of this paper to refer exclusively to heavyweight locks.

We use two different methods to obtain the desired resource breakdowns, depending on the DBMS used. For DB2, since its source code is unavailable, we rely on its built-in resource measurement facilities: snapshot and event monitoring [16]. For PostgreSQL and Shore, we implement custom measurement functionality by instrumenting the DBMS itself. We compute the total CPU, I/O and lock wait time over all transactions and then determine the fraction each component makes up of the sum of all execution times.

For DB2 and PostgreSQL, which use a process-based architecture, we verify the breakdowns at the operating system via the `vmstat` command, recording the fraction of time DBMS processes spend in the CPU run queue (`TASK_RUNNING`), blocked on I/O (`TASK_INTERRUPTIBLE`), or waiting for locks (`TASK_UNINTERRUPTABLE`). We also use a patch to the Linux kernel to accurately measure CPU wait times (not measured in Linux by default).

## 4.2. Breakdown Results

**TPC-C.** Figure 1 shows the resource breakdowns measured for TPC-C running on IBM DB2, PostgreSQL, and Shore. The graphs depict the average portions (indicated as percentages) of transaction execution time due to CPU, I/O, and lock resource usage. Although breakdowns are normalized to 100%, there is a small measurement error of less than 10% for DB2. We attribute the error to the high-granularity of DB2's I/O and CPU measurements.

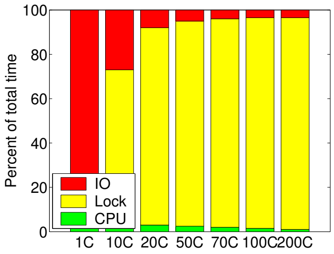
For each DBMS, Figure 1 presents three sets of results illustrating the most significant trends. In the first column, the database size is held constant at 10 warehouses (WH), and the number of clients connected to the database (concurrency) is varied. In the second column, the number of clients is held constant at 10, and the size of the database is varied by increasing the number of warehouses. In the third column, we vary the number of clients and warehouses together, always holding the number of clients at 10 times the number of warehouses, as specified by TPC-C, demonstrating breakdowns for standard *"realistic"* configurations. Throughout, the think times are fixed at zero.

The database sizes for TPC-C range from 500MB to 3GB, as the number of warehouses grows from 5 to 30 (100MB per WH). The bufferpool size is approximately 800MB for each DBMS, chosen to minimize transaction execution times.

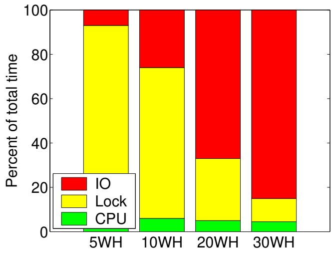
The main result shown in Figure 1 is that locks are the bottleneck resource for both Shore and DB2 (rows 1 and 2), while I/O tends to be the bottleneck resource for PostgreSQL (row 3). We now discuss these in more detail.

We start with some obvious trends. First observe that as concurrency is increased while fixing the database size (column 1), lock contention increases. Also, as the database size grows, while the concurrency level is held constant (column 2), the I/O component grows, and the lock component decreases. When the database and concurrency level are scaled according to TPC-C specifications, the relative resource breakdowns remain fairly stable.

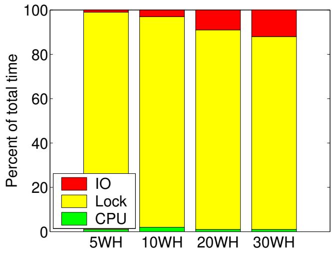
The resource breakdowns for Shore and DB2 (rows 1



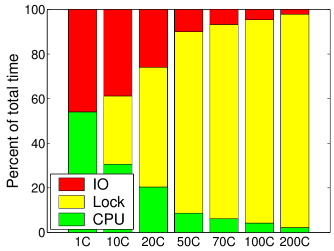

(a) **DB2**: Varying Clients, 10 Warehouses



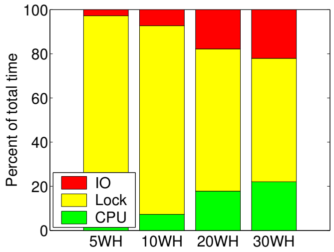

(b) **DB2**: 10 Clients, Varying Warehouses



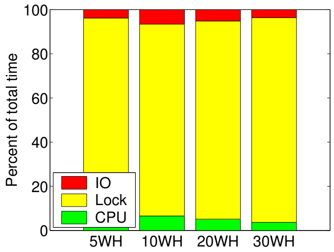

(c) **DB2**: Standard Scaling (10 clients per WH)



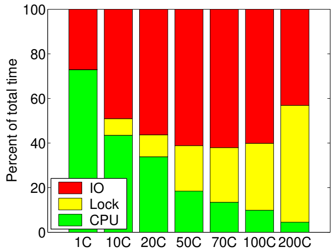

(d) **Shore**: Varying Clients, 10 Warehouses



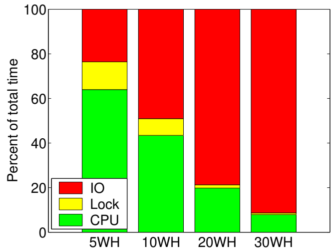

(e) **Shore**: 10 Clients, Varying Warehouses



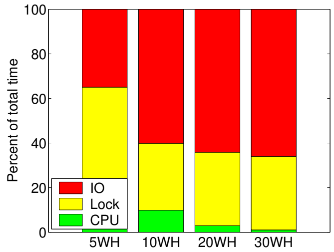

(f) **Shore**: Standard Scaling (10 clients per WH)




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

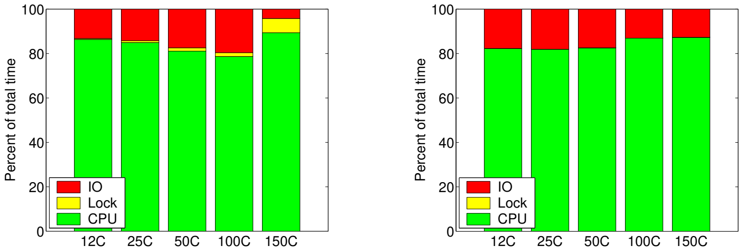



(h) **PostgreSQL**: 10 Clients, Varying Warehouses




(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. The first column (Figures 1(a), 1(d), 1(g)) shows the impact of varying concurrency level by varying the number of clients. The second column (Figures 1(b), 1(e), 1(h)) shows the impact of varying the database size (number of warehouses) while holding the number of clients fixed. The third column (Figures 1(c), 1(f), 1(i)) shows the impact of varying both the number of clients and the database size according to the TPC-C specification (10 clients for each warehouse).**




(a) **IBM DB2**

(b) **PostgreSQL**

**Figure 2. Resource breakdowns for TPC-W transactions running on IBM DB2 and PostgreSQL.**

and 2) are quite similar, and almost always depict lock bottlenecks. This may be surprising, since concurrency control was a very active area of research in the 1970's and 80's, and thus one might think that locking problems were all resolved at that time. Given our hardware limitations, we can only experiment with up to 30 WH. It is plausible that the bottleneck may shift to I/O as the database size increases. Alternatively, additional RAM and disks may hide the growing I/O for larger databases, leaving locks as the bottleneck resource.

The resource breakdowns for PostgreSQL (row 3) differ greatly from those for Shore and DB2: PostgreSQL almost always exhibits an I/O bottleneck. As indicated earlier, PostgreSQL I/O time includes the time for both the actual I/O operation and the lightweight lock I/O synchronization. Almost all (80–95%) of the I/O time is due to I/O synchronization in the standard case (Figure 1(i)). While this suggests that I/O scheduling will be necessary for PostgreSQL prioritization, in Section 5, CPU scheduling will be used to indirectly schedule I/O.

Although not the bottleneck, locks are sometimes a non-trivial component for PostgreSQL. In particular, locks reach 50% when concurrency is increased while fixing the database size (Figure 1(g)), and reach 30% when standard TPC-C scaling is used (Figure 1(i)).

The fact that PostgreSQL's resource breakdowns differ from those for Shore and DB2 is due to differences in concurrency control in these systems: Shore and DB2 employ 2PL, while PostgreSQL uses MVCC. With MVCC, PostgreSQL transactions only have to wait for write-on-write conflicts. The result is fewer lock waits in PostgreSQL than in Shore and DB2, shifting its bottleneck to I/O.

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 are primarily an indicator of the relative importance of the resources when minimizing average transaction execution times.

**TPC-W.** Figure 2 shows resource breakdowns for TPC-W transactions running on IBM DB2 and PostgreSQL as a function of the number of clients connected to the database. The size of the database is held constant (150MB), and is representative of a database used by 10 clients according to the TPC-W specification. Increasing the number of clients to 150 models extremely high data contention. PostgreSQL sees almost no locking and DB2 sees very little, as TPC-W intrinsically has very little data contention. I/O costs are also low since the database is so small relative to main memory. Thus, CPU is the bottleneck resource for TPC-W [1].

## 5. Scheduling the Bottleneck

As seen in Section 4, the bottleneck resource for TPC-C on Shore and DB2 is locks, suggesting that lock prioritization will be effective. Figure 3 motivates this point, showing that transactions that do not wait for locks are almost 20 times faster than those that do.

The bottleneck resource for PostgreSQL is usually I/O. While I/O scheduling is outside the scope of this paper, it is well-known that CPU scheduling may indirectly schedule other resources [5], such as I/O or locks. This is due to the fact that transactions need CPU resources to issue resource requests. Consequently, we investigate whether CPU scheduling is effective for PostgreSQL.

Throughout, we examine *both* lock and CPU scheduling for *both* TPC-C and TPC-W. We have reservations, however, about the TPC-W workload for two reasons: its trans-

[1] We find that under extreme configurations, lock waiting can be significant for TPC-W as well. Since these configurations depart so much from the TPC-W specifications, we do not consider them here.

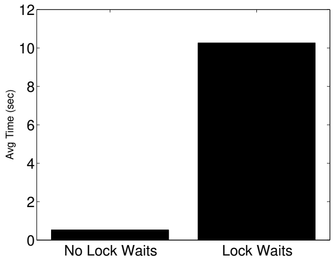



**Figure 3. Average execution time for TPC-C Shore transactions that never wait for locks compared to those that do, with no prioritization. Think time is 1 second.**

actions are (i) extremely simplistic, and (ii) need very little concurrency control. The TPC-C workload, with more complex transaction interactions, is in fact more representative of real-world applications. Note that we do not evaluate any of the scheduling policies on IBM DB2, since it does not support such policies and the source code is unavailable for experimentation.

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

## 5.1 Prioritization Workload

Throughout this section, we use a representative 10 warehouse database for TPC-C (1GB) and a 10 client database for TPC-W (150MB). Priorities are assigned to each TPC-C and TPC-W transaction according to a Bernoulli trial with probability 10% of being a high-priority.

TPC-C and TPC-W are *closed loop systems*, where a fixed number of clients alternatingly wait and execute transactions against the database. The time spent waiting is known as *think time* and models interactive clients interpreting results. The concurrency level can be adjusted by either fixing the number of clients and varying think time, or fixing the think time and varying the number of clients. We find both methods yield similar results. Throughout our experiments we will fix think time at zero and vary the number of clients. The only exception will be for TPC-C experiments, where we will instead vary the think time and fix the number of clients at 300. We choose 300, because that allows us to use think time to vary the number of running clients both above and below the TPC-C-specified 100 clients. The reason that we vary think time for TPC-C prioritization is that the TPC-C clients can consume significant system resources, and thus using a constant number of clients helps reduce variability due to this overhead.

## 5.2. Definition of the Policies

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

**Lock scheduling policies.** We first consider non-preemptive lock scheduling policies, where lock holders are never forced to release their locks abnormally due to preemption. Subsequently, we consider preemptive policies, in which high-priority transactions can preempt low-priority lock holders to acquire their locks. Preemption involves aborting, rolling back, and resubmitting the transaction, adding more work for the DBMS.

The simplest non-preemptive policy, `NP-LQ`, just reorders transactions waiting in the lock queue, and grants locks to high-priority transactions before those of low-priority. This policy has a problem: high-priority transactions moved to the front of the queue must wait for low-priority transactions already holding the lock to complete (known as “excess time” in queueing theory). The case where a high-priority transaction waits for a low-priority transaction is commonly known as priority inversion. Two techniques are commonly used to address the problem, *priority inheritance* [20] and *preemption*.

`NP-LQ-Inherit` is a non-preemptive policy that uses priority inheritance to reduce excess times. The policy is identical to `NP-LQ`, but the priority of each transaction is raised to the highest priority of any transaction that waits for it. Thus, high-priority transactions never wait for transactions with priority lower than their own. The intended result is that high-priority excess times are reduced, improving high-priority execution times.

`P-LQ` aims to reduce high-priority excess times by preempting transactions currently holding locks needed by high-priority transactions. The policy is identical to `NP-LQ`, but when a high-priority transaction needs a lock held by a low-priority transaction, the low-priority transaction is aborted (known as *preemptive abort*). In practice, two factors reduce the effectiveness of preemption. First, the preempting high-priority transaction must still wait for (part of) the low-priority transaction rollback to complete before continuing. Second, extra work created by preemption potentially slows down other transactions.

**CPU scheduling policies.** Each of the DBMS considered relies on approximations of (preemptive) generalized processor sharing (GPS), and as a result we do not distinguish preemptive or non-preemptive scheduling of the CPU device itself. We do, however, consider preemption of transactions due to lock conflicts while using CPU prioritization. We call CPU scheduling policies that preempt lock holders preemptive, and those that do not non-preemptive.

The simplest policy, CPU−Prio, is a non-preemptive policy that schedules the CPU using weighted GPS. It simply gives more weight to processes working on high-priority transactions. Specifically, for PostgreSQL, we assign UNIX priority nice level $-20$ to high-priority processes and $+20$ to low-priority processes. For Shore, high-priority threads get "time critical" priority, while low-priority transactions get "regular" priority.

Although the CPU−Prio policy prioritizes CPU, the policy may suffer from priority inversions due to locks. A high-priority transaction with high CPU-priority cannot progress if it waits for a lock held by a low-priority transaction. CPU−Prio−Inherit is a non-preemptive policy that adds priority inheritance to the CPU−Prio policy. The priority of low-priority transactions that block high-priority transactions is raised, thus reducing high-priority excess times.

The P−CPU policy is a preemptive policy identical to CPU−Prio except that low-priority transactions that block high-priority transactions are preempted and rolled back.

**Organization of remaining sections.** In Section 5.3 we present results for simple scheduling policies without preemption or priority inheritance: NP−LQ and CPU−Prio, defined above. In Section 5.4, we examine policies with priority inheritance: NP−LQ−Inherit and CPU−Prio−Inherit. In Section 5.5, we discuss the preemptive policies P−LQ and P−CPU.

### 5.3. 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 (for TPC-C) or, equivalently, by varying the number of clients (for TPC-W).

The best simple scheduling policy for TPC-C depends on the DBMS. For TPC-C running on Shore (see Figure 4(a)), CPU−Prio does not appreciably improve high-priority transaction execution times. NP−LQ, on the other hand, improves high-priority performance by 3.7 times. The penalty to low-priority transactions under both NP−LQ and CPU−Prio is small (less than 17% for NP−LQ) and tracks the "Default" no-priority setting (see Figure 4(b)). Lock scheduling is extremely effective for Shore because locks dominate transaction execution times under 2PL.

By contrast, for PostgreSQL, lock scheduling is not as effective as CPU scheduling (see Figure 4(c)). Under high loads, NP−LQ improves high-priority execution times by a factor of 1.3, whereas CPU−Prio improves them by a factor of 2. With both policies, low-priority transactions are not significantly penalized (see Figure 4(d)). As the think time increases from 5 to 25 seconds, concurrency decreases from 200 to 20 running (non-thinking) clients on average, and the lock fraction of execution times becomes insignificant. As expected, the result is that lock scheduling (NP−LQ) is not very effective.

The effectiveness of CPU−Prio for TPC-C on PostgreSQL is surprising, given that I/O (I/O-related lightweight locks) is its bottleneck. Due to CPU prioritization, high-priority transactions are able to request I/O resources before low-priority transactions can. As a result, high-priority transactions wait fewer times (52% fewer) for I/O, and when they do wait, they wait behind fewer transactions (43% fewer). The fact that simple CPU prioritization is able to improve performance so significantly suggests that more complicated I/O scheduling is not always necessary.

For TPC-W, locks are never the bottleneck resource (see Figure 2), suggesting lock scheduling will be ineffective. As confirmation, Figure 5 shows average execution times with NP−LQ and CPU−Prio for TPC-W as a function of the number of clients. As expected, NP−LQ does not significantly improve high-priority transactions. CPU−Prio, however, dramatically improves high-priority transaction times by a factor of up to 4.5 under high load (high number of clients) relative to a system with no priorities.

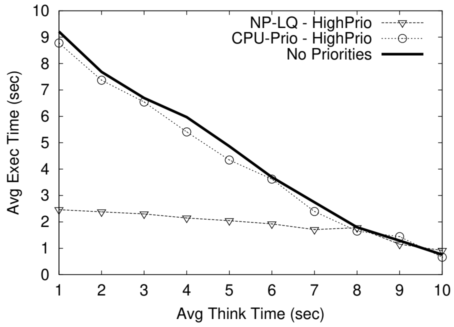
Low-priority transactions, on average, are not significantly penalized by either NP−LQ or CPU−Prio, for all DBMS and workloads studied. This result is important, and consistent with theoretical results: Performance of a small class of high-priority transactions can be improved without harming the overall low-priority performance.

### 5.4. Priority Inheritance

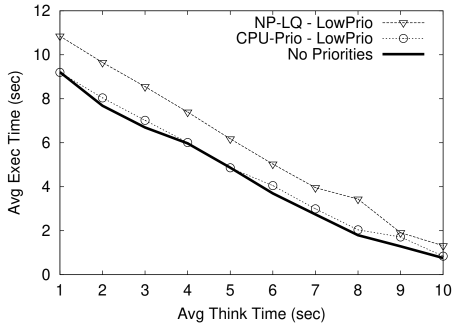
In this section we evaluate the two policies using priority inheritance: NP−LQ−Inherit and CPU−Prio−Inherit, which are extensions of the NP−LQ and CPU−Prio policies, respectively.

Figure 6 compares the policies NP−LQ−Inherit and NP−LQ for TPC-C running on Shore for a range of concurrency levels. We find that adding priority inheritance to simple lock queue reordering (NP−LQ) improves performance by 30%. NP−LQ improves high-priority transaction execution times by a factor of 3.7 relative to a system without priorities, and NP−LQ−Inherit improves execution times by a factor of 5.3.

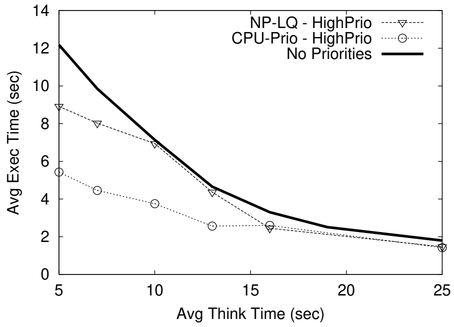
For TPC-C running on PostgreSQL, adding priority inheritance to NP−LQ offers no appreciable gain in perfor-



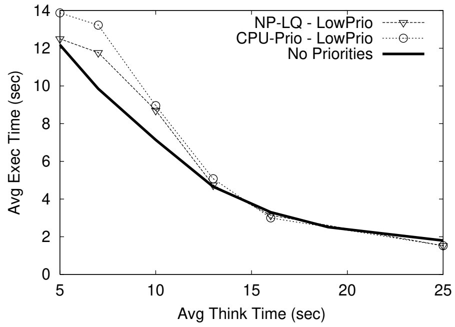

(a) **Shore** High-Priority




(b) **Shore** Low-Priority




(c) **PostgreSQL** High-Priority




(d) **PostgreSQL** Low-Priority

**Figure 4. Mean execution times for `NP-LQ` compared to `CPU-Prio` for Shore and PostgreSQL TPC-C with varying contention. Concurrency (load) increases to the left, as think time goes down.**

mance, however, priority inheritance with CPU scheduling is beneficial. Figure 7 shows CPU priority inheritance improves high-priority transactions by a factor of 6, whereas `CPU-Prio` only helps by a factor of 2. The significant improvement in performance is due to the fact that the lock holder(s) are sped up, resulting in significantly smaller wait excesses.

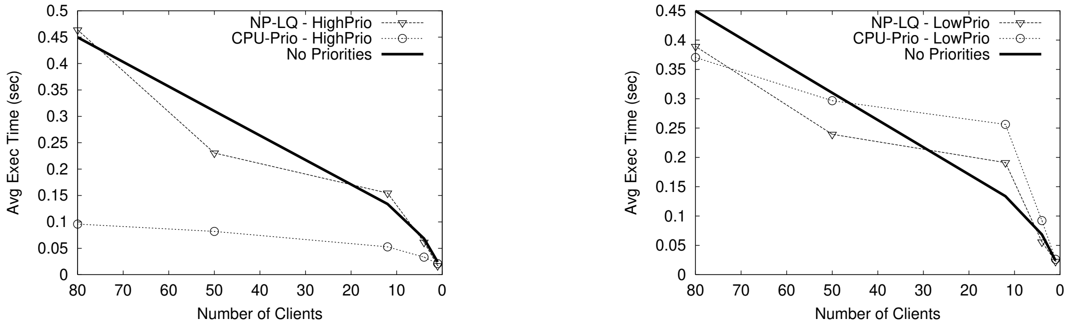
Recall from Section 5.3 that CPU scheduling (`CPU-Prio`) is more effective than `NP-LQ` for TPC-W. Thus Figure 8 compares the policies `CPU-Prio-Inherit` to `CPU-Prio` for the TPC-W workload on PostgreSQL. We find that there is no improvement for `CPU-Prio-Inherit` over `CPU-Prio`. This is to be expected given the low data contention found in the TPC-W workload; priority inversions can only occur during data contention. Results for low-priority transactions are not shown, but as in Figure 4, low-priority transactions are only negligibly penalized on average.

## 5.5. Preemptive Scheduling

Non-preemptive scheduling already provides substantial performance improvements for high-priority TPC-C transactions, using lock scheduling for Shore and CPU scheduling for PostgreSQL. We now focus on whether preemption can provide further benefits. In particular, we evaluate whether `P-LQ` improves on `NP-LQ` for Shore and whether `P-CPU` improves on `CPU-Prio` for PostgreSQL.

With non-preemptive scheduling, high-priority transactions sometimes must wait on lock requests for locks currently held by low-priority transactions (the wait excess). The wait excess time is reduced, but not eliminated, with priority inheritance, which speeds up the low-priority transactions blocking high-priority transactions. Preemptive scheduling (`P-LQ` and `P-CPU`) attempts to eliminate the wait excess for high-priority transactions by preempting low-priority lock holders in the way of high-priority transactions.

We find that preemptive policies provide little benefit




(a) **PostgreSQL** High-Priority

(b) **PostgreSQL** Low-Priority

**Figure 5. Mean execution times for `NP-LQ` compared to `CPU-Prio` for PostgreSQL TPC-W with varying loads.**

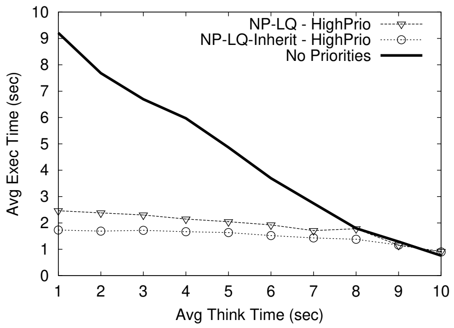



**Figure 6. `NP-LQ-Inherit` compared to `NP-LQ` for Shore TPC-C.**

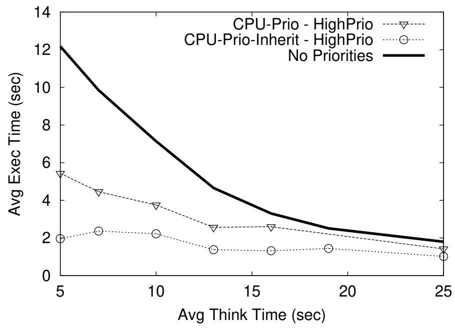



**Figure 7. `CPU-Prio-Inherit` compared to `CPU-Prio` on PostgreSQL TPC-C.**

over non-preemptive policies. Figures 9(a) and 9(b) compare the average high- and low-priority execution times for `P-LQ` against `NP-LQ-Inherit` for TPC-C on Shore as a function of think time. High-priority transactions with `P-LQ` improve by a factor of 9.3 whereas `NP-LQ-Inherit` helps only by a factor of 5.3. Low-priority transactions, however, are slowed by a factor of 1.7, which is excessive, making this policy impractical. Figures 9(c) and 9(d) compare the performance of `P-CPU` to `CPU-Prio-Inherit` for TPC-C on PostgreSQL. Preemption seems to offer no significant benefit or penalty beyond `CPU-Prio-Inherit`.

TPC-W results for `P-LQ` and `P-CPU` are omitted as lock scheduling is ineffective since lock contention is low.

**Future extensions to Preemptive Priorities.** There are two problems with preemption that limit its effectiveness in our experiments. First, the penalty to low-priority transactions may be excessive. Second, the cost of waiting for a transaction to complete may be cheaper than preemption. As a result, there may be room for improvement in both `P-LQ` and `P-CPU`.

We explore a few other preemptive policies that are more selective about which transactions to preempt. These policies predict a victim transaction's remaining life expectancy and the cost of rolling back the victim to determine whether to preempt or wait. The first idea is to use the number of locks held by the victim to predict its remaining age. If it holds many, it is almost finished, but if it holds few, it is just starting. Second, we use the "wall-clock" age of the victim as a predictor. Although preliminary, we find these are both poor predictors of transaction life-expectancy. It is possible that better predictors can be invented.

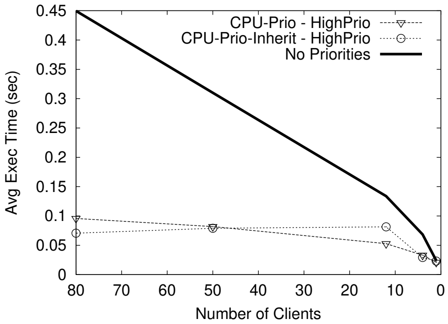



**Figure 8.** `CPU-Prio-Inherit` **compared to** `CPU-Prio` **for TPC-W running on PostgreSQL.**

## 6. Conclusion

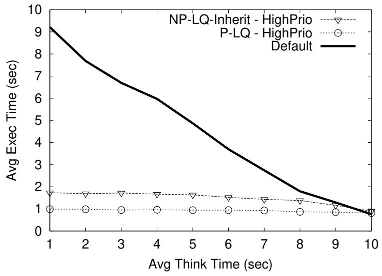
In this paper we develop and evaluate an implementation of transaction prioritization for differentiated performance classes for TPC-C or TPC-W workloads running on traditional relational DBMS.

We first identify the bottleneck resource at which priority scheduling is most effective. We divide the lifetime of a transaction into three components: CPU, I/O, and lock wait times. The results are clearly differentiated by workload and concurrency control mechanism. Across a wide range of configurations, the bottleneck for TPC-C running on DBMS using 2PL (Shore and DB2) is *lock waiting*. By contrast, the bottleneck for TPC-C running on MVCC DBMS is *I/O synchronization* for low loads, although locking can dominate at extremely high concurrency levels. For TPC-W workloads, *CPU* is always the bottleneck.

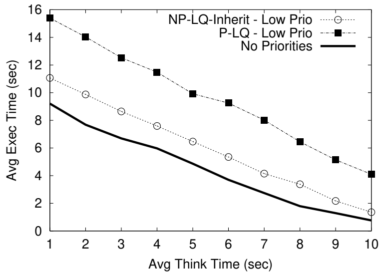
This bottleneck analysis provides a roadmap for which resources must be scheduled to improve performance. In this paper we focus on lock and CPU scheduling to directly or indirectly schedule the bottleneck resource. We evaluate the effectiveness of simple prioritization, priority inheritance, and preemptive abort scheduling, and the results are broken down by workload and concurrency control mechanism.

For TPC-C on 2PL DBMS (Shore), non-preemptive lock scheduling with priority inheritance (`NP-LQ-Inherit`) is most effective. For Shore, high-priority transaction execution times improve 5.3 times, while low-priority transactions are hardly penalized. Priority inheritance and preemption do not appreciably help, and preemption excessively penalizes low-priority transactions. By extension, we believe that these results will hold for IBM DB2 since it has a similar resource breakdown to Shore.

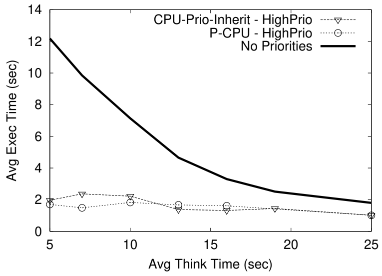
For TPC-C on MVCC DBMS, and in particular PostgreSQL, CPU scheduling is most effective, due to its ability to indirectly schedule the I/O bottleneck. For TPC-



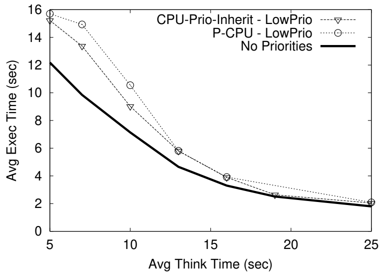

(a) **Shore** High-Priority




(b) **Shore** Low-Priority




(c) **PostgreSQL** High-Priority




(d) **PostgreSQL** Low-Priority

**Figure 9. Preemptive policies** `P-LQ` **and** `P-CPU` **for Shore and PostgreSQL respectively, compared to the best non-preemptive policies for TPC-C.**

C running on PostgreSQL, the simplest CPU scheduling policy (CPU-Prio) provides a factor of 2 improvement for high-priority transactions, while adding priority inheritance (CPU-Prio-Inherit) provides a factor of 6 improvement while hardly penalizing low-priority transactions. Preemption (P-CPU) provides no appreciable benefit over CPU-Prio-Inherit.

For TPC-W on all DBMS, we find that lock scheduling is largely ineffective since transactions rarely wait for locks. CPU scheduling, however, is extremely effective. For TPC-W running on PostgreSQL, we find that the simplest scheduling policy, CPU-Prio, is best, and improves performance for high-priority transactions by a factor of up to 4.5. Priority inheritance is not necessary since data contention for TPC-W is almost non-existent.

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 policies 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] 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.

[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] IBM Corporation. IBM DB2 query patroller administration guide version 7.0, 2000.

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

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

[13] Sameh Elnitky, Erich M. Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in dynamic e-commerce web sites. Unpublished Manuscript, May 2003.

[14] 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.

[15] 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.

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

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

[18] PostgreSQL. http://www.postgresql.org.

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

[20] 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.

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