Improving Preemptive Prioritization via Statistical Characterization of OLTP Locking

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

Abstract

OLTP and transactional workloads are increasingly common in computer systems, ranging from e-commerce to warehousing to inventory management. It is valuable to provide priority scheduling *in these systems, to reduce the response time for the most important clients, e.g. the "big spenders". Two-phase locking, commonly used in DBMS, makes prioritization difficult, as transactions wait for locks held by others regardless of priority. Common lock scheduling solutions, including non-preemptive priority inheritance and preemptive abort, have performance drawbacks for TPC-C type workloads.*

The contributions of this paper are two-fold: (i) We provide a detailed statistical analysis of locking in TPC-C workloads with priorities under several common preemptive and non-preemptive lock prioritization policies. We determine why non-preemptive policies fail to sufficiently help high-priority transactions, and why preemptive policies excessively hurt low-priority transactions. (ii) We propose and implement a policy, POW, *that provides all the benefits of preemptive prioritization without its penalties.*

1 Introduction

Long delays and the accompanying unpredictably large response times [1] are a source of frustration in online transaction processing (OLTP) database systems. In many applications, consistently low response times are essential for users. Consider, for example, an online stock market with significant price volatility. A trader issues trade orders based on constantly varying market prices, and any delay creates potential for huge profit loss.

Minimizing delay and its unpredictability is much more valuable for some users than for others. A trader making thousands of large-volume trades a day may be willing to pay more for reduced delays on trades. On the other hand, a trader making only one trade a month may accept much more variable response times. Thus, we divide transactions into two classes: high- and low-priority, based on whether the transaction is issued by a high- or low-paying customer. Our primary goal is to *prioritize high-priority transactions* to execute as if *in isolation* of low-priority transactions, and ensure low-priority transactions do not delay high-priority transactions. Second, low-priority transactions must not be excessively penalized.

Transaction prioritization can be important in countless contexts. In commercial OLTP, for instance, customers who experience many excessive delays may become frustrated, and take their business elsewhere. Giving high-priority service to customers who routinely buy expensive merchandise will maximize the company's profits. As a testament to the importance of transaction prioritization, it is provided in most major commercial DBMS: DB2 offers db2gov and QueryPatroller[13, 4] and Oracle offers DRM [15]. We have previously shown that *CPU scheduling is ineffective* for prioritization in OLTP applications (such as TPC-C), while *lock scheduling is highly effective* [14]. Unfortunately, all the above commercial systems focus on CPU, not lock prioritization. Additionally, there is little research on lock scheduling in fully implemented general-purpose DBMS, as most are analytical or simulation studies, or focus on RTDBMS.

Many open questions remain for lock scheduling in general-purpose DBMS. Of these, we focus on whether the DBMS should use a *preemptive* or a *non-preemptive* scheduling policy. Each type of policy has advantages and disadvantages, and there is no consensus as to which is best. While preemptive policies allow high-priority transactions to reduce lock waiting time by

*Supported by NSF grants CCR-0113660, IIS-0133686, CCR-0205544, and BES-0329549

†Supported by NSF grants CCR-0133077, CCR-0311383, 0313148, and by IBM via 2003 Pittsburgh Digital Greenhouse Grant.

[1] Response time is defined as the time from when a transaction is submitted until it completes, including restarts.

killing other lock holders, rollbacks and re-executions may be too costly. Non-preemptive policies avoid these preemptive overheads, but high-priority transactions may wait for low-priority transactions to complete before making progress.

The first contribution of our paper is a performance evaluation and in-depth statistical analysis of lock activity in TPC-C, for common scheduling policies. For non-preemptive policies, such as queue reordering (*NPrio*) and priority inheritance (*NPrioinher*), high-priority transactions are poorly isolated from low-priority transactions, resulting in variable and high response times. By contrast, preemptive policies (*PAbort*) yield good high-priority performance, but excessively penalize low-priority transactions.

To determine why non-preemptive policies fail to isolate high-priority transactions, we address four questions: (i) How many lock requests do transactions wait for? (ii) How long are lock waits? (iii) How long do transactions wait for current lock holders versus for other waiting transactions? (iv) How do lock waits affect response time? We show that the common policies primarily fail to eliminate *wait excess*: the time spent waiting for current lock holders to release locks. To determine why preemptive policies devastate low-priority transactions, we investigate potential reasons: (i) rollback costs (ii) preemption frequency, and (iii) wasted work per preemption. Surprisingly, we find that most of these issues are largely irrelevant, and wasted work per preemption dominates exclusively.

The second contribution of our paper is a demonstration that a little-known and unevaluated lock scheduling policy from the field of distributed databases, *Preempt-On-Wait* [16] (*POW*), excels over all the above policies. It combines the excellent high-priority performance of preemptive policies with the small penalty to low-priority transactions typical with non-preemptive policies.

The intuition behind *POW* is that if a high-priority transaction H needs a lock held by a low-priority transaction L, H should only preempt L if L will hold the lock a long time. We find that whether or not L waits in another lock queue is a highly accurate indicator of L's remaining holding time. Thus, *POW* only preempts low-priority transactions that both wait for a lock and block a high-priority transaction.

Our evaluation focuses on the TPC-C OLTP workload with Shore [3] (a modern prototype with transaction management, 2PL, and Aries-style recovery), and concentrates on improving high-priority transaction response times. Basic theory dictates that in all closed systems, like TPC-C, throughput is directly related to response-time.

This paper presents the first major statistical analysis of locking with priority-scheduling in a fully implemented general-purpose DBMS, and thus incorporates complex system interactions, such as I/O. While Shore is noncommercial, it is important to note that (i) this evaluation could not be conducted using a commercial DBMS due to the lack of source code, and (ii) resource utilizations for Shore have been repeatedly shown to be remarkably similar to that of IBM DB2 [14, 2].

For prioritization to be most effective, the fraction of high-priority transactions should be low. Throughout the paper, we randomly assign high-priority to 10% of the transactions, and low-priority to the remaining 90%. This is a pessimistically-realistic scenario, and results are similar when the ratio is varied.

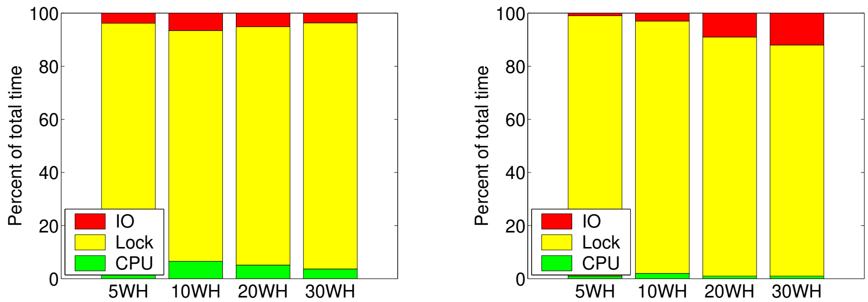
The paper is organized as follows: In Section 2 we describe the prior work on priority scheduling. In Section 3 we review existing results showing that lock queues are the appropriate resource to schedule given general-purpose DBMS with lock-based concurrency control. In Section 4, we describe our evaluation of the common lock scheduling policies. In Section 5, we present the bulk of this work, a statistical profile of locking in Shore TPC-C with priorities. In Section 6, we present the *POW* algorithm and its performance analysis. Finally, we conclude in Section 7.

2 Prior Work

DBMS lock scheduling has been studied for decades, covering countless policies and systems. Most work concerning preemptive and non-preemptive lock scheduling focus on RTDBMS, and are primarily simulation or analytical studies. This is in stark contrast to our focus on general purpose DBMS, OLTP workloads, and full prototype evaluation.

NPrio [1], a non-preemptive policy that reorders lock queues according to priority, is one of the earliest policies considered. Without preemption, however, improvement to high-priority transactions is limited, since they must sometimes wait for low-priority transactions. This problem is known as *priority inversion*, and most other policies' goals are to address it.

NPrioinher, uses *priority-inheritance* [18, 17, 9] to reduce the cost of priority inversions. Low-priority transactions that block high-priority transactions become high-priority themselves. The idea is to reduce high-priority transaction wait times by speeding up the transactions they wait for. If those transactions do not wait on locks, or if too many transactions' priorities increase, the effectiveness becomes unclear. In simulation and prototypes, some research finds that *NPrioinher* is not as effective as *PAbort* in RTDBMS [11, 10]. In con-



(a) **Shore**

(b) **IBM DB2**

Figure 1. TPC-C Shore and DB2 average I/O, Lock, and CPU resource utilization, relative to total average transaction response time.

trast, other simulation studies find that *NPrioinher* is, in fact, effective in RTDBMS as long as transaction arrivals are non-bursty [1].

PAbort (Preemptive Abort, or Wound-Wait) [9, 11, 16], preempts low-priority transactions that block a high-priority transaction. Since preempted transactions must be restarted, there may be significant extra work into the system, slowing transactions down. In simulation and RTDBMS testbeds, many researchers [11, 10, 9, 21] find that *NPinherit* is not as effective as *PAbort*. As indicated above, this contradicts the conclusions of others [1]. None of these studies consider general-purpose DBMS and workloads.

Much work has been done to improve preemptive policies by reducing the number of preemptions and extra work. In distributed databases, Rosenkrantz et. al. [16] mention *POW* (see Section 6) as a possible variation of *PAbort*, in which running transactions are not preempted, but do not implement the algorithm, nor analyze its performance. Conditional Restart (*CR*) and Conditional Priority Inheritance (*CPI*) [11, 10] in RTDBMS estimate the time until low-priority lock holders complete, and preempt if it take too long. We find common estimates, such as the number of locks held, do not work well for TPC-C type workloads.

Wait-depth-limited (*WDL*) [19, 20, 7, 6, 8] policies preempt transactions to keep chains of waiting transactions shorter than a given depth. Running Priority (*RP*) [6, 20] is a common *WDL* policy in which transactions wait only for transactions that are currently running. Though *RP* does not consider priorities, and *POW* is not *WDL*, the preemption conditions are similar (see Section 6).

Our work addresses three limitations in the literature:

- Neither preemptive nor non-preemptive policies are strictly superior, and it is difficult to predict which is best for OLTP workloads.
- Most work focuses on RTDBMS, rather than OLTP workloads and general-purpose DBMS. RTDBMS rely on specialized operating systems and workloads that result in different performance tradeoffs than in general DBMS.
- Only a few RTDBMS studies [10] examine locking in fully implemented systems, where complicated interactions can greatly affect performance.

Our prior work [14] is primarily a bottleneck analysis of TPC-C (summarized in Section 3), although we also observe the limitations of common non-preemptive and preemptive lock scheduling policies. This paper supercedes that work, focusing on DBMS using 2PL, with an in-depth analysis identifying the reasons for these limitations. Further, we introduce the *POW* policy which does not suffer these limitations.

3 Bottleneck: Locks

Here, we review prior work [14], demonstrating that for TPC-C OLTP workloads on DBMS using 2PL, locks are almost always the bottleneck resource. Specifically, from the perspective of an individual transaction, its response time is dominated by time waiting to acquire locks. As a consequence, I/O and CPU scheduling will be *ineffective* for prioritization, so we focus exclusively on lock scheduling. It is important to note that overall system CPU and I/O utilization are high, as some transactions are always making progress.

We consider TPC-C type workloads on both commercial and non-commercial DBMS, namely IBM DB2 [5], PostgreSQL [12], and Shore [3]. Each of these systems is profiled, counting time transactions spend waiting for locks and I/O and both consuming and waiting for CPU. (Lock time only accumulates when waiting for locks. After a lock is acquired, the time is spent in CPU, I/O, or waiting for other locks). For both IBM DB2 and Shore, which use traditional 2PL, our results show that, on average, transactions spend more than 80% of their lifetime waiting for locks. We find that this trend is present *over a wide range of configurations*. Only in the most unrealistic configurations are other resources be relevant.

Figure 1 shows the average resource breakdown for both Shore and IBM DB2 as a function of database size, measured in TPC-C warehouses. Each warehouse adds 100MB, and the bufferpool is 800MB. The number of concurrent clients is 10 times the number of warehouses, as specified by TPC-C. On average, waiting for locks accounts for most of the response times, and dominates even as the number of clients (load) or the size of the database are varied. While IBM DB2 I/O time increases as the database grows, it is not realistic to run more than 30 warehouses on our limited testbed hardware. In real applications, growing I/O cost is hidden by additional memory and disks.

4 Evaluating Lock Scheduling Policies

As seen in Section 2, the effectiveness of preemptive and non-preemptive lock scheduling policies cannot be easily predicted. In this section, we experimentally evaluate the behavior of the common policies, and seek to understand their performance trade-offs.

4.1 Experimental Setup and Methodology

We focus on the following lock scheduling policies, which are commonly used and referenced in the literature:

Standard: This is the baseline for comparison: transactions are not prioritized.

NPrio: Non-preemptive lock queue reordering. When locks are released, waiting compatible transactions are granted the lock in priority-order (from high- to low-priority).

NPrioinher: Non-preemptive lock queue reordering with priority inheritance. Locks are granted as in *NPrio*. Additionally, low-priority transactions that block high-priority transactions become high-priority (for the remainder of their lifetime) to release locks more quickly.

PAbort: Preemptive Abort. A low-priority transaction that blocks a high-priority transaction is always immediately preempted (aborted, rolled back, and restarted).

We implement the above lock scheduling policies in Shore and measure their effects on average high- and low-priority transaction response times in a TPC-C workload. The tests are run on a 2.2GHz Pentium 4 with two disks (one for data, one for log), 1GB of RAM and an 800MB bufferpool. Transactions run in serializable isolation level, given the critical nature of many OLTP applications (while weaker isolation levels will result in less locking, this issue is orthogonal to this work).

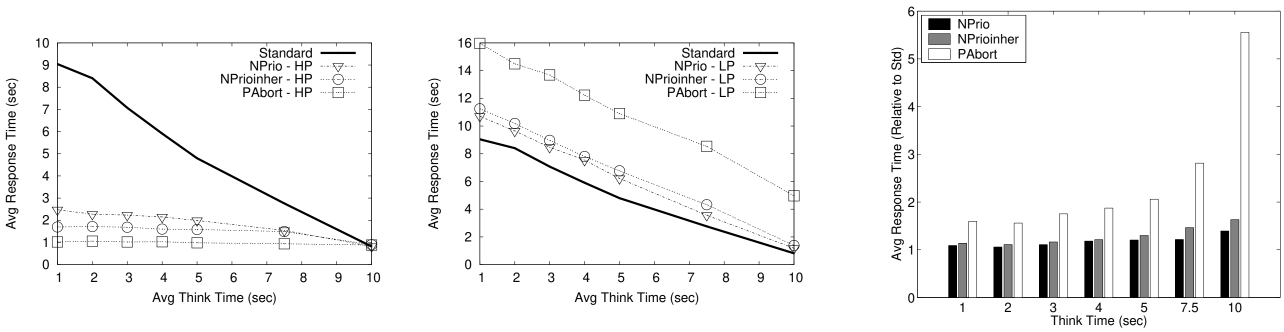
Implementation of these policies in Shore involves sorting lock queues and minor modifications to lock acquire and wakeup functions. The biggest difficulty is forcing the deadlock detector to handle dynamically reordering lock queues. This is an artifact of the original Shore deadlock detection algorithm and should be less of an issue with an independent deadlock detection process, such as in DB2.

10% of the TPC-C transactions are independently and randomly assigned high-priority and the remaining 90% low-priority. The TPC-C code tells Shore the transaction priority, and retries deadlocked and preempted transactions. The database size is 10 Warehouses (1GB on disk), and is appropriate for our hardware limitations (the database size does not greatly affect the lock bottleneck [14]).

To vary concurrency (load), we change the arrival process to have 300 clients (rather than the TPC-C-specified 100 clients) and consider a range of client "think times" between submitting transactions. We vary the think time from 10 seconds ("low load") to 1 second ("high load"). This range of think time results in an average number of active clients in the database from about 25 to 250, allowing us to investigate concurrency levels both well below and above the 100 clients specified by TPC-C.

4.2 Performance Evaluation

Figure 2 depicts transaction response times under the common lock scheduling policies. Figure 2(a) shows mean response time for high-priority transactions and Figure 2(b) shows mean response time for low-priority



(a) **High-Priority**

(b) **Low-Priority**

(c) **Overhead**

Figure 2. Average TPC-C Shore response times for high- and low-priority transactions as a function of load for *NPrio*, *NPrioinher*, *PAbort*, and *Standard* policies (2(a) and 2(b)). Aggregate high- and low-priority response time relative to *Standard* (2(c)).

transactions. Throughout the paper, lower think time (left end) indicates higher load.

NPrio improves response times of high-priority transactions relative to *Standard* by a factor of 4 at high load. By comparison, *NPrioinher* improves response times of high-priority transactions by a factor of 5.3 at high load, and *PAbort* improves high-priority response times over *Standard* by a factor of 9. This significant improvement in high-priority response times further confirms that locks are the bottleneck resource. Under low loads, lock waiting time becomes less significant, and all the policies perform similarly.

The story is very different for low-priority transactions. *NPrio* and *NPrioinher* only slightly harm low-priority transactions as compared to *Standard*, increasing response time by a factor of 1.2 at high load. By comparison, *PAbort* drastically hurts low-priority performance, increasing response time by a factor of 1.8 at high load when compared with *Standard* and by much more at low load.

It is interesting to note that in Figure 2(a), the high-priority transaction response times increase as a function of load when no priorities are used (*Standard*), but remain relatively stable when using priority scheduling. This artifact is due to the TPC-C arrival process, which uses a fixed number of clients that submit transactions separated by exponential think times (i.e., a “closed system” in queueing theory). As each transaction has probability p of being high-priority, each client is expected to create one high-priority transaction for each $\lfloor 1/p \rfloor - 1$ low-priority transactions. Since low-priority transactions are an order of magnitude slower than high-priority transactions, the fraction of high-priority clients in the system is in fact much smaller than p. As a result, in Figure 2, the time-average fraction of high-priority transactions in the system ranges between 1.1% and 7.7% for *NPrio* (with absolute values ranging from 3 to 2.3 on the average), with similar numbers for the remaining priority policies. As the load increases, there are more and more low-priority transactions, but only a few high-priority transactions, resulting in relatively stable high-priority response times and degrading low-priority performance.

While *PAbort* appears to offer significant benefits to high-priority transactions (factor of 9 improvement) its penalty to low-priority transactions is too high, making it inappropriate for real DBMS. At the same time, while *NPrioinher* does well for both high- and low-priority transactions, its inability to do as well as *PAbort* for high priorities is discouraging. The primary disadvantage of preemptive scheduling in *PAbort* is the fact that it introduces extra work into the system (rollbacks and re-execution of preempted transactions). It is important to understand exactly how much extra work is created.

One might think that prioritizing transactions does not affect the overall average response time (aggregated over high- and low-priority transactions), but simply provides better response time for high-priority transactions in exchange for worse response time for low-priority transactions. This is not necessarily true however for policies like *PAbort* which introduce significant overhead. Figure 2(c) studies the overhead incurred by all the common prioritization policies. Here the response times of the policies are shown normalized by the response time for *Standard* (that is they have been divided by *Standard*'s response time). An overhead of 1 (on the y-axis) indicates that the policy's average transaction response time (aggregate over high- and low-priority transactions) is the same as *Standard*, and the policy has not slowed the overall system down. The non-preemptive policies *NPrio* and *NPrioinher* have low overhead. However *PAbort* has overall average response times 1.5 to 6 times greater than *Standard*, indicating a huge overhead introduced due to preemption. The reason that preemption performs worse under low loads is due to the fact that transactions complete and release locks faster under lower loads, while rollback costs remain about constant (discussed in Section 5.2).

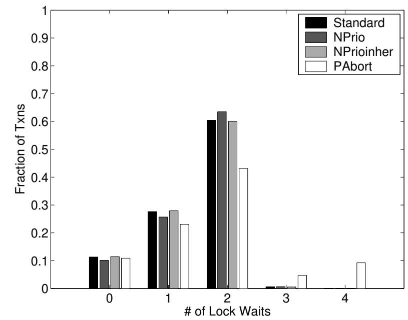


Figure 3. Distribution on the number of times that high-priority transactions wait for a lock under common lock scheduling policies (Similar for low-priority transactions).

5 Statistical Profile of TPC-C Locking

In this section, we examine several hypotheses to explain the behavior of *PAbort* and *NPrioinher*, and test these hypotheses using empirical statistical measurements of the system. We will determine first why non-preemptive high-priority performance is not as good as in *PAbort*, and second why low-priority performance under *PAbort* deteriorates.

5.1 High-Priority Performance under Non-Preemptive Policies

There are four questions that must be answered to understand why preemptive policies are superior to non-preemptive policies in improving high-priority response times. (i) How many lock requests do high-priority transactions wait for? (ii) How long are the lock waits, and how do they contribute to high-priority response times? (iii) How much lock waiting is attributed to current lock holders? (iv) How much do lock waits contribute to response times?

(i) How many lock requests do high-priority transactions wait for? Shore TPC-C transactions make between 3 and 550 lock requests, depending on the type of the transaction (e.g.: New Order, Payment, etc). Understanding the fraction of these lock requests that are forced to wait will determine the flexibility available to lock scheduling policies.

Figure 3 shows the probability distribution on the number of times transactions wait for locks under each of the common scheduling policies. While the distribution is shown for high-priority transactions, the distributions for low-priority transactions is similar. Over 99% of the transactions wait for fewer than 3 lock requests, while fewer than 1% wait for 3 or more lock requests (Figure 3 truncated at 4 for clarity).

Interestingly, the number of lock waits for non-preemptive policies does not change significantly as a function of the policy. Preemptive scheduling changes the distribution slightly, as preempting transactions reduces the expected number of locks held in the database, reducing contention. None of the policies try to directly reduce the number of times high-priority transactions wait on locks, which may involve knowledge of future lock requests. While reducing the number of lock waits may be an effective strategy to improve high-priority transactions, we only focus on reducing lock wait times once they occur.

(ii) How long are lock waits? The fact that high-priority transactions wait only for a few locks suggests an answer to our second question: individual lock waits are very long. For confirmation, we examine the average time that a transaction waits when it waits for a single lock request. We refer to this time as *QueueTime*, measured from when the transaction initiates the lock request until it is granted. Note that for preemptive policies, *QueueTime* includes preemptions, in which case it is made up of the time needed to preempt the transaction(s) holding the lock, until the preempted transaction(s) release the lock.

Figure 4 depicts the *QueueTime* experienced by high-priority transactions for *NPrio*, *NPrioinher*, and *PAbort*. On average, high-priority *QueueTime* makes up $40 - 50$ % of the the high-priority response time for all policies. Since 25% of transactions wait once and 40-60% wait twice, transactions are expected to include one or two *QueueTimes*, slowing the transactions considerably. Part of the reason that *PAbort* outperforms

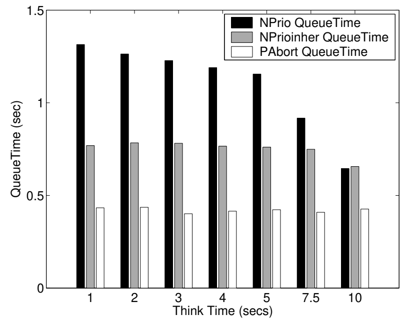


Figure 4. Average high-priority *QueueTime* **for** *NPrio*, *NPrioinher*, **and** *PAbort* **as a function of load (think time).**

NPrioinher is that its *QueueTime* is only half as long.

(iii) How much lock waiting is attributed to current lock holders? It is important to understand *QueueTime* in more detail, because, as we have seen, long *QueueTimes* prevent non-preemptive policies from sufficiently improving high-priority response times. Under non-preemptive policies, a transaction's *QueueTime* is comprised of two components: (i) *WaitExcess*, the time from when the lock request is made until the first transaction waiting for the lock is woken and acquires the lock, and (ii) *WaitRemainder*, the time from when the first waiter acquires the lock until the lock request is finally granted. Intuitively, *WaitExcess* is the time that a transaction waits for current holders to release the lock, and *WaitRemainder* is the time the transaction waits for other transactions in the queue with it. The question we want to address is which of *WaitExcess* or *WaitRemainder* is most responsible for high-priority *QueueTimes*.

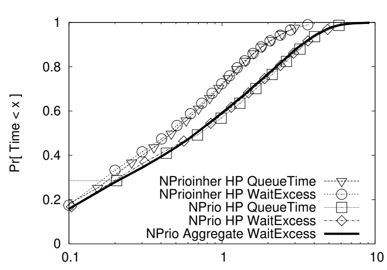


Figure 5. CDF of high-priority *QueueTime* **and** *WaitExcess* **for** *NPrio* **and** *NPrioinher* **for high load along with aggregate high- and low-priority** *WaitExcess* **for** *NPrio*.

Figure 5 compares the probability distributions for high-priority *QueueTime* and *WaitExcess* for *NPrio* and *NPrioinher* under high load. The two leftmost (upper) overlapping lines are *NPrioinher* high-priority *QueueTime* and high-priority *WaitExcess*. The three rightmost (lower) overlapping lines are *NPrio* high-priority *QueueTime*, high-priority *WaitExcess*, and overall average *NPrio WaitExcess*.

The fact that the high-priority *QueueTime* distribution is exactly the same as high-priority *WaitExcess* proves that high-priority transactions never wait behind other transactions in the queue, and only wait for the current lock holder. Figure 5 also demonstrates that priority inheritance (*NPrioinher*) reduces *WaitExcess* by boosting the priority of the current lock holders. Additionally, high-priority *NPrio WaitExcess* is identical to overall average *NPrio WaitExcess*, reflecting the fact that *NPrio* does not improve the response time of current lock holders.

Since high-priority *WaitRemainder* is effectively zero for the non-preemptive policies, the only remaining issue affecting high-priority performance is *WaitExcess*. While priority inheritance can help reduce *WaitExcess* by speeding up lock holders, there are limits to its effectiveness, and no clear way to extend the policy to improve its *QueueTimes* further.

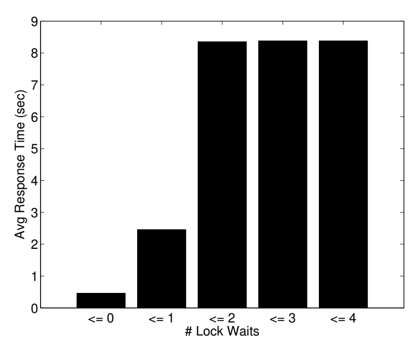


Figure 6. Average transaction response time as a function of the number of times a transaction waits under high load, when using the *Standard* **policy.**

(iv) How much do lock waits contribute to response times? Thus far we've seen that long response times can be attributed to waiting on a few locks with large *WaitExcess* times. We now ask how exactly the response time is correlated to the number of locks that a transaction waits on. Figure 6 depicts the average response time of a transaction as a function of the number of times the transaction waits for a lock request, for the *Standard* policy and high load (1 second think time). The average response time of transactions that never wait for locks is a factor of 18 smaller than the overall average response time. In addition, these transactions complete faster than the mean high-priority response time under both *NPrioinher* and *PAbort* (a factor of 3.7 and 2.2 improvement respectively). *This statistic shows that an accurate predictor for the length of a transaction's remaining execution time is whether the transaction is about to wait for locks or not.* A desirable feature of this predictor is that it has low overhead, as it requires no history or bookkeeping in order to provide an estimation.

In conclusion, in this section we show that for TPC-C workloads, while high-priority transactions acquire numerous locks during their lifetime, they are forced to wait on very few (almost always less than 3 waits). This suggests, and we confirm, that the time spent waiting for these blocking lock requests comprises a large portion of high-priority transaction response times. We also show that high-priority transactions almost never wait for other transactions in the queue with them, and just wait for the current holders of the lock to release them. Finally, we show that those transactions which wait for one or fewer locks (40% of all transactions) have extremely short response times.

5.2 Low-Priority Performance under Preemptive Policies

While high-priority response times under *PAbort* are very promising, the effect of *PAbort* on low-priority transactions is disastrous. Our goal is to examine the statistical evidence to determine exactly the cause and significance of this problem.

In this section, we examine the well-known penalties for preemption that lead to poor performance: (i) the cost of rolling back transactions, (ii) the number of times transactions are preempted, and (iii) the work lost executing transactions that are subsequently preempted. We show that (iii) is almost the exclusive reason for poor low-priority performance.

Individual rollback costs in *PAbort* have two primary consequences. First, rollbacks delay both the preempting high-priority transaction and the preempted low-priority transaction(s). To ensure ACID properties, a high-priority transaction cannot immediately acquire the lock of a transaction it preempts, but must wait for the preempted transaction to rollback and release the needed lock. The high-priority transaction need not wait for the entire rollback, but only until the needed lock is released. Typically, low-priority transactions are not resubmitted until the rollback is complete. Second, rollbacks require DBMS CPU and I/O resources to clean up the preempted transaction which could otherwise be used for other transactions, potentially slowing down transactions overall.

We find that transaction rollback costs average about .5 seconds over all loads. This cost is nontrivial relative to the cost of a high-priority transaction. By contrast it is insignificant for low-priority transactions, which take between 5 to 16 seconds on average. It should be noted that in optimized commercial systems, rollbacks should be even less significant.

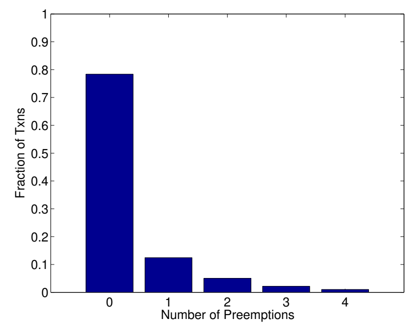


Figure 7. Probability distribution on the number of times a transaction is preempted by *PAbort* under high load (1 second think time).

The next question is whether there are simply too many rollbacks, in which case the total cost of several small rollbacks may be significant. We find, however, that this is not the case. Figure 7 shows the probability distribution on the number of times a transaction is preempted by *PAbort* under high load (1 second think time). About 80% of transactions are never preempted. For those that are preempted, the trend is approximately geometric with about 12% being preempted once, 5% twice, 2.5% three times, etc. On average, the number of preemptions (and rollbacks) per transaction is less than 0.4, and the expected cost of rollbacks overall is not large relative to the average low-priority response time. Something else must hurt the low-priority response times.

Finally, we examine the amount of work wasted in processing transactions that are eventually preempted. For all loads, the age of a transaction when it is preempted is between 75% and 90% of the length of an average transaction. Thus, a preempted transaction essentially doubles its expected execution cost (assuming independences). The conclusion is that the work lost due to preemption is the most significant flaw of *PAbort*.

6 Preempt-On-Wait Scheduling

In Section 4 we conclude that high-priority transaction performance is hindered under *NPrioinher* because transactions wait too long for current lock holders. Similarly, low-priority transactions are hurt under *PAbort* because too many transactions (20%) are preempted after completing a significant amount of work. In this section, we describe and evaluate the *Preempt-On-Wait* (*POW*) lock scheduling policy, which combines the best of both worlds: *PAbort*'s good high-priority performance and *NPrioinher*'s good low-priority performance.

Section 6.1 describes the *POW* algorithm and its implementation in Shore. Section 6.2 demonstrates that *POW* achieves the best of *PAbort* and *NPrioinher*. Section 6.3 compares *POW* to several state-of-the-art preemptive policies. Finally, Section 6.4 provides a statistical analysis explaining why *POW* meets its performance goals.

6.1 The POW Algorithm

POW is motivated by the following logic: consider a low-priority transaction L that blocks a high-priority transaction H. We have seen that if L is preempted, much work is lost, penalizing low-priority response times. If L is not preempted, its remaining time depends on whether it waits for a lock (in which case its remaining time is long) or doesn't wait for a lock (in which case its remaining time is very short).

In *POW*, when a high-priority transaction H waits for a lock X_1 held by a low-priority transaction L, L is preempted if and only if L currently, or in the future, waits for some other lock X_2. Additionally, lock queues are reordered as in *NPrio* to ensure that high-priority transactions are first to get the lock when it is released.

The implementation of *POW* for Shore builds on the implementations of *NPrio* and *PAbort* as described in Section 4.1. The only additional state needed is a boolean flag *fpow* for each transaction, which requires *almost no computational overhead.* If H must wait for L, and L is currently waiting for another lock, *fpow* is set. On all blocking lock acquisitions, if *fpow* is set, the transaction is aborted.

For example, consider that low-priority transactions L_1, L_2, and L_3 all hold a lock X in shared mode. L_1 is currently waiting to acquire another lock, L_2 will need to wait on another lock in the future before it completes, and L_3 will not wait for any more locks before it completes (though it may acquire several more). If high-priority transaction H requests an exclusive lock on X, it will immediately preempt L_1, and set the flag on L_2 and L_3. When L_2 makes its first lock request and is forced to wait, *POW* sees that L_2's flag is set, thus L_2 is aborted. Since L_3 does not block on any more locks, it completes. In this case, H acquires the lock X as soon as all of L_1, L_2, and L_3 have either completed or aborted.

	NPrioinher	*PAbort*	*POW*
HP improvement	3.41x	5.45x	5.60x
LP penalty	1.36x	2.27x	1.16x

Table 1. High- and Low-priority response time speedup relative to *Standard* policy.

6.2 POW Performance Evaluation

Figure 8 compares the performance of *POW* with that of the common lock-scheduling policies, as a function of load. *POW* high-priority response times are nearly identical to those for *PAbort* for all loads. Simultaneously, *POW* low-priority response times are nearly identical to those for *NPrioinher*. Therefore, *POW* outperforms both *PAbort* and *NPrioinher* (and also *NPrio*). As the probability of a transaction being high-priority varies from 1% to 10%, the same trend holds.

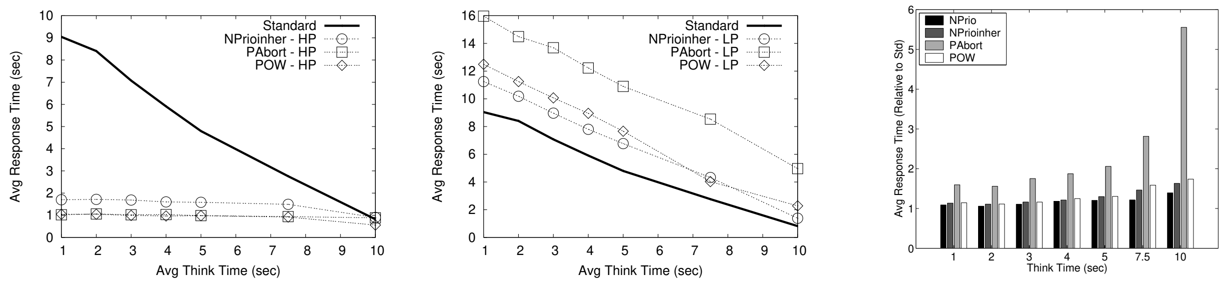
Table 1 shows the high-priority improvement and low-priority penalty under *POW* and the common policies, averaged over the range of think times. *POW*'s improvement to high-priority transactions (a factor of 5.6 improvement over *Standard*) exceeds even that of *PAbort*. *POW*'s penalty to low-priority transactions (a factor of 1.16 above *Standard*) is even lower than that of *NPrioinher*.

As explained in Section 4.2, response times of high-priority transactions remain relatively constant as the load increases, and the number of high-priority transactions in the system at any time is relatively constant (between 1.1% and 5.2%).

Figure 8(c) depicts the overhead (overall average transaction response time relative to *Standard*) for *POW*. The overhead of *POW* is always comparable with *NPrioinher* and (to a lesser extent) *NPrio*. By comparison, the overhead of *PAbort* is disastrous (See Figure 2(c)).

6.3 POW vs Other Preemptive Polices

In this section, we compare *POW* to other types of state of the art preemptive lock scheduling policies from the literature: *WDL* (wait-depth-limited) [7] and



(a) **High-Priority**

(b) **Low-Priority**

(c) **Overhead**

Figure 8. Average response time for high- and low-priority transactions for POW**,** $PAbort$**, and** $NPrioinher$ **as a function of load (8(a) and 8(b)). Aggregate high- and low-priority average response time relative to** $Standard$ **(8(c)).**

CR (conditional restart). Since POW allows long lock chains to form, it is not itself a *WDL* policy. Additionally, its preemption conditions differ from those employed in typical *CR* policies.

First, we consider *WDL1*, a simple wait-depth-limited policy without prioritization. This policy simply preempts transactions to ensure that a lock chain contains no more than 2 transactions. When three transactions wait in a chain $T_1 \rightarrow T_2 \rightarrow T_3$, T_2 is preempted. We find that *WDL1* performs poorly on our workload, particularly under high loads, since it has hot-spots and and high contention. Transactions are preempted too frequently (more than 50 times each) and make no forward progress. Our attempts to extend *WDL1* to respect priorities fail to resolve this problem.

We also consider a variation of the *CR* policy, *CR300*. *CR300* is identical to $PAbort$, except that transactions are given a reprieve time (300ms) to complete before being preempted. By contrast, *CR* preempts transactions immediately, but must make difficult predictions about transactions' remaining times. *CR300* safely avoids this issue, relying on the fact that high-priority transactions wait only for $WaitExcess$, and those $WaitExcesses$ are very short (we find 30% are less than 300ms and 50% are less than 500ms). Varying the reprieve time from 100ms to 1000ms does not change performance significantly.

Figure 9 illustrates high- and low-priority transaction response times for each of the policies $Standard$, *CR300*, and POW. Invariably, the best policy for both high- and low-priority transactions is POW. *CR300* performs similarly to $NPrioinher$ for both high- and low-priority transactions. POW manages to outperform *CR300* by preempting more transactions that greatly slow down high-priority transactions (POW preempts 4 times as many under high loads). It turns out that whether a transaction waits for a lock is a better predictor of its remaining time than whether it completes within its reprieve time.

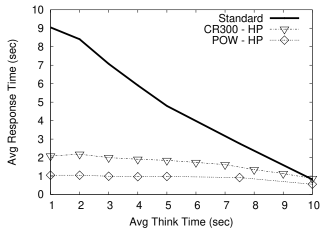
6.4 Explaining POW Performance

In order to understand why POW improves high-priority transaction response times as much as $PAbort$ without hurting low-priority response times, we conduct a statistical evaluation of TPC-C under POW.

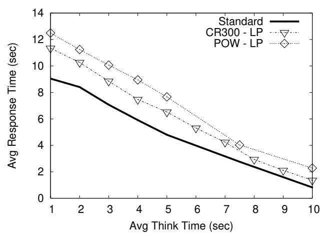
Low-Priority Penalty. As determined in Section 4.2, the primary reason low-priority transactions suffer with $PAbort$ is work lost to transactions later preempted. Two factors affect the amount of this wasted work: (i) the number of preempted transactions and (ii) the age of a transaction when preempted.

We find that POW preempts less than 1% of transactions ($\sim$ 265 preemptions), while $PAbort$ preempts 20% of the transactions ($\sim$ 5000 preemptions). These figures are fairly constant over all loads. Thus, POW allows almost all low-priority transactions that block high-priority transactions to complete during their "reprieve." Thus, only a handful (1%) of transactions are penalized more with POW than with $NPrioinher$. The result is that low-priority response times and overhead for POW are similar to that of $NPrioinher$.

High-Priority Improvement. POW improves high-priority transaction response times because it significantly reduces high-priority $QueueTime$. We consider $QueueTime$ in two cases: in the case where



(a) **High-Priority**



(b) **Low-Priority**

Figure 9. Average response time for high- and low-priority transactions with preemptive policies *CR300* **and** *POW***.**

the lock holder completes ($QueueTime|Wait$) and in the case where the lock holder is preempted ($QueueTime|Preempt$).

Figure 10 compares the average high-priority *QueueTime* for *POW* and *PAbort*. While *POW*'s ($QueueTime|Preempt$) can be large, ($QueueTime|Wait$) is similar to *PAbort*'s *QueueTime*. Since *POW* preempts so few transactions, the overall average *POW QueueTime* is similar to the average *PAbort QueueTime*. The variability in *POW*'s ($QueueTime|Preempt$) in Figure 10 is a result of so few preemptions.

POW is as good for high-priority transactions as *PAbort* because whenever a high-priority transaction waits for a lock, it waits no longer than it would have if it preempted the current holder(s). In the few cases where *POW* preempts the lock holder(s), the waiting time will be extraordinarily large. Furthermore, the time lost waiting to determine whether to preempt the holders is not very long (2-3 times an average lock wait).

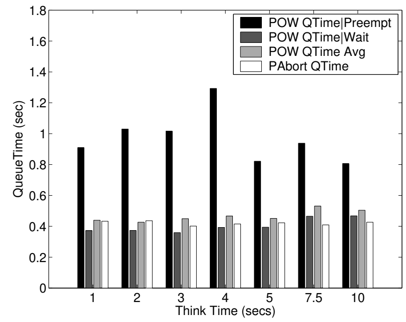


Figure 10. Average time for high-priority *QueueTime*, *QTime|Preempt*, **and** *QTime|Wait* **as a function of load.**

7 Conclusion

The goal of this work is to provide user priority classes for OLTP applications, such as TPC-C, using priority scheduling in the DBMS. As Shore (and similarly, IBM DB2) exhibits lock bottlenecks for these workloads, we consequently focus exclusively on lock scheduling. Experimental evaluation of common preemptive and non-preemptive lock scheduling policies in this environment reveals that no policy is clearly superior. The common policies have limited ability to improve high-priority response times without significantly hurting those for low-priority transactions.

Consequently, we formulate a novel and detailed statistical analysis of locking in TPC-C on Shore with the common lock scheduling policies. We draw two primary consequences from this analysis. *First,* with non-preemptive lock scheduling, *WaitExcess* dominates the delays experienced by high-priority transactions, and *WaitExcess* is not greatly reduced by techniques such as priority inheritance (Figures 4 and 5). Furthermore, if a transaction waits for a *WaitExcess*, its response time is 5-18 times longer than if it does not (Figure 6). As a result, while non-preemptive scheduling policies such as *NPrioinher* barely penalize low-priority transactions, they insufficiently improve high-priority transactions. *Second,* for preemptive lock scheduling, though preemption can introduce many overheads, the only relevant penalty is work wasted on transactions later preempted.

The above analysis suggests prioritization policies must exploit the tradeoff between wasted work and *WaitExcess*. To that end, we propose and implement the *POW* lock scheduling policy for TPC-C workloads on lock-based DBMS. *POW* exploits the statistical profile of locking in the workload, combining the excellent high-priority performance of *PAbort* with the good low-priority performance of *NPrioinher*. This is the first application and evaluation of *POW* for OLTP DBMS and workloads. Experimental results show that *POW* improves high-priority transaction response times by a factor of 5.6 on the average, while hurting low-priority transactions by only 16%. Thus, preemption can be effective with a low penalty in traditional DBMS and OLTP applications.

This work has several high-level impacts: First, *POW*-like policies can be used in online and commercial OLTP environments to increase profits, both by enabling service-level agreements and by ensuring good performance and satisfaction for high-profit customers. Second, analytical DBMS performance models may use our statistical profile of TPC-C locking to develop more accurate and tractable models for OLTP workloads. The dominating factors of *WaitExcess* and work lost in preempted transactions allow modelers to ignore irrelevant aspects of the system. Last, our analysis forms a basis for studying other workloads and building a taxonomy of workloads' statistical profiles, invaluable for DBMS algorithm development and tuning.

References

[1] Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *VLDB*, pages 1–12, 1988.

[2] A. Ailamaki, D.J. DeWitt, and M.D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB*, 11(3), 2002.

[3] M. Carey, D. J. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up persistent applications. In *Proc. of SIGMOD*, May 1994.

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

[5] DB2 Product Family. `http://www.ibm.com/software/data/db2`.

[6] Peter A. Franaszek and John T. Robinson. Limitations of concurrency in transaction processing. *ACM TODS*, 10(1):1–28, March 1985.

[7] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. Wait depth limited concurrency control. In *Proc. of ICDE*, pages 92–101. IEEE, April 1991.

[8] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. Concurrency control for high contention environments. *ACM TODS*, 17:304–345, June 1992.

[9] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Data access scheduling in firm real-time database systems. *JRTS*, 4(3):203–241, September 1992.

[10] Jiandong Huang, John A. Stankovic, Krithi Ramamritham, and Donald F. Towsley. On using priority inheritance in real-time databases. In *Proc. Real-Time Systems Symposium*, pages 210–221, 1991.

[11] Jiandong Huang, John A. Stankovic, Krithi Ramamritham, Donald F. Towsley, and Bhaskar Purimetla. Priority inheritance in soft real-time databases. *JRTS*, 4(3):243–268, 1992.

[12] Sailesh Krishnamurthy, Spiros Papadimitriou, Bianca Schroeder, and Anastassia Ailamaki. PostgreSQL, chapter in Database System Concepts, by H. Korth, A. Sibershatz, and S. Sudarshan, McGraw Hill, 5th Edition.

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

[14] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proc. of ICDE*. IEEE, 2003.

[15] Ann Rhee, Sumanta Chatterjee, and Tirthankar Lahiri. The Oracle database resource manager: Scheduling CPU resources at the application. HPTS, 2001.

[16] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis, II. System level concurrency control for distributed database systems. volume 3, pages 178–198. ACM Press, 1978.

[17] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical Report CMU-CS-98-181, Carnegie Mellon University, 1987.

[18] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[19] Alexander Thomasian. Performance analysis of locking policies with limited wait depth. *Performance Evaluation Review*, 20(1), June 1992.

[20] Alexander Thomasian. A performance comparison of locking methods with limited wait depth. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):421–434, 1997.

[21] Özgür Ulusoy and Geneva G. Belford. Concurrency control in real-time database systems. In *Proc. ACM Annual Conference on Communications*, pages 181–188, 1992.