Charm: An I/O-Driven Execution Strategy for High-Performance Transaction Processing

Lan Huang
Department of Computer Science
State University of New York
Stony Brook, NY 11790
lanhuang@cs.sunysb.edu

Tzi-cker Chiueh
Department of Computer Science
State University of New York
Stony Brook, NY 11790
chiueh@cs.sunysb.edu

Abstract

The performance of a transaction processing system whose database is not completely memory-resident critically depends on the amount of physical disk I/O required. This paper describes a high-performance transaction processing system called *Charm*, which aims to reduce the concurrency control overhead by minimizing the performance impacts of disk I/O on lock contention delay. In existing transaction processing systems, a transaction blocked by lock contention is forced to wait while the transaction currently holding the contended lock performs physical disk I/O. A substantial portion of a transaction's lock contention delay is thus attributed to disk I/Os performed by other transactions. *Charm* implements a two-stage transaction execution (TSTE) strategy, which makes sure that all the data pages that a transaction needs are memory-resident before it is allowed to lock database pages. Moreover, *Charm* supports an optimistic version of the TSTE strategy (OTSTE), which further eliminates unnecessary performance overhead associated with TSTE. Another TSTE variant (HTSTE) attempts to achieve the best of both TSTE and OTSTE by executing only selective transactions using TSTE and others using OTSTE. *Charm* has been implemented on the Berkeley DB package and requires only a trivial modification to existing applications. Performance measurements from a fully operational *Charm* prototype based on the TPC-C workload demonstrate that *Charm* out-performs conventional transaction processing systems by up to 164% in transaction throughput, when the application's performance is limited by lock contention.

1 Introduction

High-performance transaction processing systems have seen a resurgent interest within the explosively growing E-commerce community, especially after the publicly reported "melt-down" of the on-line electronic trading system of several well-established stock brokerage houses. The user-perceived response time of a transaction consists of three components: CPU processing time, disk I/O time, and waiting time due to lock contention. For high-throughput transaction processing systems used in banking and stock trading applications, CPU time is typically insignificant compared to disk I/O time. Unless the underlying database is fully memory-resident, read disk I/O cannot be completely eliminated. Moreover, database logging and data persistence require write disk I/O even with main-memory database management systems. Most on-line transaction processing systems, although equipped with a large amount of physical memory to reduce the number of disk I/Os, do not necessarily have the luxury to keep the entire database memory-resident. Lock contention delay is itself often dependent on the amount of disk I/O, because existing transactions systems allow transactions that are holding locks to perform disk I/O and thus lengthen the waiting time of those transactions that are blocked by the held locks.

While extensive research has been done in the concurrency control area to improve the throughput of transaction processing systems, this paper presents the design, implementation and evaluation of a transaction execution strategy that is *orthogonal* and thus *complementary* to existing concurrency control algorithms. By re-arranging the order of execution of I/O operations within each individual transaction, the Two-Stage Transaction Execution

(TSTE) strategy described in this paper greatly reduces the average response time of individual transactions and improves the overall throughput of the transaction processing system. The authors wish to emphasize that the TSTE strategy is *not* yet another concurrency control algorithm, because its goal is not to reduce the number of lock conflicts, but to reduce the contention delay associated with each lock conflict.

Specifically, the goal of TSTE is to reduce the lock contention delay in disk-resident transaction processing systems to the same level as that experienced by memory-resident transaction processing systems. TSTE runs each transaction in two stages, a *fetch* stage and an *operate* stage. In the *fetch* stage, TSTE brings all the data pages required by a transaction to main memory and pins them down. *No locks are acquired in this stage.* In the *operate* stage, TSTE actually executes the transaction in exactly the same way as in traditional transaction processing systems, i.e., acquiring/releasing and waiting for locks, etc. Because the data pages required by a transaction are guaranteed to be memory-resident in the *operate* stage, the contention among transactions running in this stage is identical to the case when these transactions are running against a memory-resident database. In other words, as far as lock contention is concerned, TSTE turns a disk-based transaction processing system into a memory-resident transaction processing system by decoupling disk I/O from lock acquisition/release: in the *fetch* stage, a TSTE transaction performs disk I/O but not locking, and vice versa in the *operate* stage. Consequently the following invariant always holds true for an ideal decoupled transaction processing system: *The lock contention delay that a transaction experiences never includes the disk I/O time of another transaction.* Due to practical implementation issues and inter-transaction data sharing behavior, the above invariant may not always hold in TSTE. Therefore, despite its ability to reduce the lock contention delay, TSTE also incurs additional performance overhead. We have developed several optimization techniques to effectively reduce this extra performance cost.

We have implemented the first *Charm* prototype, including several performance optimizations, based on the Linux-based Berkeley DB package, version 2.4.14 [1, 2, 17] and carried out a detailed performance study on this prototype using a standard online transaction processing benchmark, TPC-C. The rest of this paper is organized as follows. Section 2 reviews previous related research efforts on the reduction of concurrency control cost. Section 3 describes in more detail the TSTE strategy, as well as its optimized variants. Section 4 presents the results and analysis of a detailed performance study on the *Charm* prototype. Section 5 concludes this paper with a summary of main results from this research and an outline of on-going work.

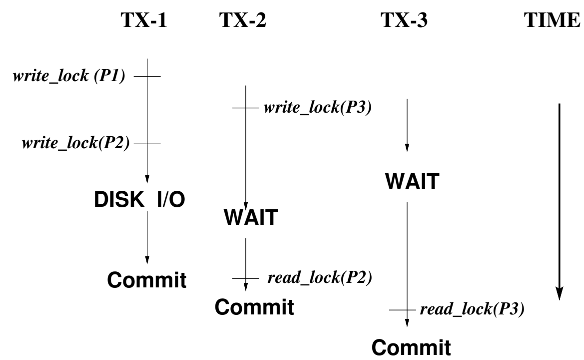


Figure 1: An example scenario in which transactions that are blocked due to lock contention (TX-2 and TX-3) actually experience the delay of the physical disk I/O performed by another transaction (TX-1).

2 Related Work

The idea of TSTE was originally proposed in earlier work [14] and has been analyzed by Franaszek et. al. [12, 13], which exploited the concept of *access invariance* to perform speculative disk I/O, and compared TSTE with various combinations of optimistic concurrency control and two-phase locking (2PL) through a simulation-based study. None of these works included the optimistic TSTE and hybrid TSTE schemes discussed in this paper. Moreover, *Charm* is the first known implementation of TSTE in industrial-strength transaction processing software, and this paper is the first to report empirical performance measurements of TSTE from a system researcher's view.

TSTE is *not* a new concurrency control algorithm. While newer concurrency control algorithms exploit semantics of transactions [3] or objects of abstract data types [4, 5] to release locks as early as possible, TSTE takes a completely different approach by removing disk I/O time from lock contention delay. There is a superficial similarity between optimistic concurrency control algorithms [6] and TSTE, especially its optimistic variant, because they both adopt rollback as a mechanism to undo the effects of writes when the speculated assumptions do not hold. However, the underlying conditions of speculation, as well as when to rollback, are very different for these two algorithms. Optimistic concurrency control assumes that lock contention is rare, and therefore proceeds with data accesses without acquiring locks. It is required to roll back the transaction when lock conflicts are detected at the *end* of the transaction. TSTE, on the other hand, assumes that disk I/O is rare, and undoes all the intermediate writes when it detects that a transaction is going to perform the *first* physical disk I/O. Whereas optimistic concurrency control attempts to reduce the lock acquisition/release overhead, TSTE aims to reduce the portion of lock contention delay attributed to disk I/O.

The idea of *decoupled architecture* [7] in computer architecture literature partially motivates the TSTE strategy. It was originally proposed to bridge the speed gap between CPU execution and memory access, but has been extended to address the performance difference between memory and disk [8]. The main idea is to statically split a program into a computation part and a data access part, and to run the data access part ahead of the computation part so that the data required by the computation part has been brought into cache (memory) from memory (disk) by the time it is actually needed. TSTE's *fetch* stage is essentially a (close to) perfect prefetching mechanism. However, the performance advantage of disk data prefetching, lies not in decreased data access delays, but in the reduction of lock contention delays as seen by other transactions that contend for locks that are being held. Recently Chang [15] developed an automatic binary modification tool that modifies existing binaries to perform speculative execution in order to generate prefetching hints.

TSTE is different from other file system or database prefetching research [9, 10] because its goal is not really about cutting down the number of physical disk I/Os, but to decouple disk I/Os from lock acquisitions. As the database size increases and application access patterns become more complicated, the effectiveness of file system/database prefetching methods decreases, but the usefulness of TSTE increases because it is more likely for a transaction to encounter disk I/O while holding locks.

3 Two-Stage Transaction Execution

3.1 Basic Algorithm

The fundamental observation motivating the Two-Stage Transaction Execution (TSTE) strategy is that the average lock contention delay in main memory database systems is much smaller than disk-resident database systems because of the absence of disk I/O. For example, in Figure 1 Transaction 2 is blocked because of the read access to P2, which is write-locked by Transaction 1, and Transaction 3 is blocked because of the read access to P3, which is write-locked by Transaction 2. As a result, both Transaction 2 and 3 experience the delay associated with the disk I/O performed by Transaction 1. In general, the delay of a physical disk I/O could appear as part of the response time of one or multiple transactions. The ultimate goal of TSTE is to ensure that each disk I/O's access delay contributes to the response time of exactly one transaction, the one that initiates the disk I/O.

Table 1 shows the average percentage of a transaction's lock contention delay that is due to disk I/Os performed by other transactions, measured on the Berkeley DB package running the TPC-C workload with the *Warehouse*[1] parameter set to five, which represents a database larger than 1 GByte. The testing machine has an 800 MHz PIII CPU, 512 Mbytes of memory is used for the user level buffer cache. There are a total of 640 Mbytes physical memory in this machine. The database tables reside on three 5400-RPM disks: one disk holds transaction logs; the other two disks hold all other data. This table is meant to illustrate the extent of the potential performance improvement if disk I/O is completely decoupled from lock contention. For example, up to 93.3% of the lock contention delay could be eliminated when the number of concurrent transactions is two. As the number of concurrent transactions increases, this percentage decreases because delay due to true data contention starts to dominate.

The basic idea of TSTE is to split the execution of each transaction into two stages. In the *fetch* stage, TSTE performs all the necessary disk I/Os to bring the data pages that a transaction needs into main memory and pins them down, by executing the transaction once *without updating the database.*

[1] Warehouse is the database size scaling factor.

One possible implementation of the *fetch* stage is to keep a local copy of the updates without committing them to the database at the end of the transaction. Another implementation, which is used in our system, is to skip all update operations and only bring into main memory necessary pages. In the second *operate* stage, the transaction is executed again, in the same way as in conventional transaction processing architectures. Because the *fetch* stage brings all required data pages into memory, transactions in the *operate* stage should never need to access the disks as long as access invariance holds. In those cases that the access invariance property does not hold, the transactions need to perform disk I/Os in the *operate* stage.

No. of Concurrent Transactions	**Percentage of Lock Contention Delay due to Disk I/O**
2	93.3%
4	80.9%
6	54.1%
8	34.8%
10	18.4%

Table 1: Average percentage of lock contention delay that a transaction experiences that is due to disk I/Os performed by other contending transactions, versus the number of concurrent transactions. The measurements are collected on the Berkeley DB package running the TPC-C workload with the *Warehouse* parameter set to five. The user-level database cache size is set to 512 Mbytes and CPU is an 800 MHz Pentium III.

The execution of a transaction in the *fetch* stage is special in two aspects. First, transactions executing in this stage do not lock database items before accessing them. Because transactions cannot hold locks on database items, it is impossible for one transaction in the *fetch* stage to wait for a lock held by another transaction. On the other hand, transactions in the *operate* stage never need to access disks, because the pages they need are brought into memory in the *fetch* stage. As a result, a transaction blocked on a lock should never experience, during the waiting period, any delay associated with disk I/O performed by the transaction currently holding the lock. With the TSTE strategy, the lock contention delay contains only CPU execution and queuing times and is thus much smaller than that in conventional transaction processing architectures.

Second, because transactions do not acquire locks before accessing data, they should not modify data in the *fetch* stage, either. That is, transactions perform only read disk I/Os in the *fetch* stage, including those data pages that are to be written as well as the address generation computation for data page accesses, but skip all write operations. This guarantees that the result of executing transactions using TSTE is identical to that in conventional transaction processing architectures.

By separating disk I/O and lock acquisition into two mutually exclusive stages, TSTE eliminates the possibility of long lock contention delay due to disk I/O. However, TSTE itself incurs additional performance overhead. The fact that TSTE executes each transaction twice means that TSTE is doing redundant work compared to conventional transaction processing systems. For transactions whose CPU time is large, this redundant work may overshadow the performance gain from TSTE. Fortunately, the CPU processing time is typically small compared to disk I/O delay for applications that require high transaction throughput such as stock trading applications. However, for long running transactions, TSTE may require too much memory to hold fetched pages and may exceed memory resources.

Because TSTE does not acquire locks, it is possible that the data pages chosen to be brought into memory in the *fetch* stage are not the same as those needed in the *operate* stage. The reason is that between the *fetch* and *operate* stages, the pages that a transaction needs may change. In this case, some of the read disk I/Os performed in the *fetch* stage are useless and thus redundant. These disk I/Os correspond to *mis-prefetching*. For example, each `delivery` transaction in TPC-C is supposed to fulfill the oldest order in the database, and therefore should only start after the previous `delivery` transaction is finished. Prefetching the data record for the current "oldest order" before the previous `delivery` transaction is completed almost certainly lead to mis-prefetching. Furthermore, transactions that mis-prefetch actually need to perform read disk I/O in the *operate* stage to retrieve those pages that the *fetch* stage should have brought in. Therefore, the invariant that no transaction needs to wait for another transaction that is performing physical disk I/O does not hold for mis-prefetching transactions.

The scenario described in Figure 1 is not limited to lock contention incurred by false-sharing when only page-level or table-level locking is supported. Even for a system with fine-grain locking, lock contention blocked by disk I/O can be reduced to a minimum using TSTE.

There is an exception to this two-stage execution strategy: accesses to database pages that are typically memory resident and for which the lock holding period is usually short. An example is accesses to highly concurrent data structures such as B-trees [11]. Specifically, in the current *Charm* prototype, accesses to the *intermediate* but not *leaf* nodes of B-trees are preceded by lock acquisitions even in the *fetch* stage. The separate treatment of B-tree's intermediate and leaf nodes prevents de-referencing of outdated and potentially dangling pointers, and reduces the probability of *mis-prefetching*, at the expense of a relatively minor performance cost.

3.2 Optimistic TSTE

To reduce TSTE's redundant CPU computation overhead, we developed an optimistic version of the TSTE (OTSTE) algorithm. With database buffering, not all transactions need to perform physical disk I/O in the *fetch* stage. Therefore, what a TSTE transaction attempts to achieve in the *fetch* stage, i.e., making sure that data pages needed in the *operate* stage are in main memory, may be redundant in some cases. Instead of always running a transaction in two stages, OTSTE starts each transaction in the *operate* stage. If all of a TSTE transaction's data page accesses hit in the database cache, the transaction completes successfully in one stage. If, however, the transaction indeed needs to access the disk, it "converts" itself to the *fetch* stage when the first such instance arises and subsequently proceeds to the *operate* stage as in standard TSTE.

The transition from the *operate* stage to the *fetch* stage in OTSTE involves the following three steps. First, the effects of all the database writes before this data access are un-done. Second, all the locks acquired in the *operate* stage are released. Third, all the data pages that this transaction has touched are pinned down in memory, in preparation for the subsequent *operate* stage.

Compared to TSTE, OTSTE eliminates the *fetch* stage altogether when the data pages required by a transaction are already memory-resident. However, OTSTE also incurs an additional overhead due to the transition from the *operate* stage to the *fetch* stage when a transaction needs to perform physical disk I/O and acquire locks. The major component of this stage-transition overhead is attributed to the undo of earlier database writes. As we will show later, OTSTE's performance gain resulting from eliminating unnecessary *fetch* stages does not out-weigh the overhead associated with rolling back updates.

3.3 Hybrid TSTE

Transactions running under TSTE incur redundant computation overhead when the transactions do not require physical disk I/O. OTSTE is meant to address this problem. However, in OTSTE, mis-prefetching and rollback overhead still exist. To further reduce these overheads, we developed another variant of TSTE called Hybrid TSTE (HTSTE), which classifies transactions running under TSTE into three types:

1. Transactions that encounter significant mis-prefetches in the *operate* stage;
2. Transactions that do not encounter mis-prefetches in the *operate* stage, and
3. Transactions the do not require any disk I/O.

HTSTE runs the first type of transactions as in traditional transaction processing systems, i.e., start them in the *operate* stage and do not convert them into the *fetch* stage even when physical disk I/O is needed. For this type of transaction, HTSTE does not incur rollback overhead. HTSTE runs the second type of transaction directly in the *fetch* stage, i.e., as in generic TSTE. HTSTE runs the third type of transactions in the *operate* stage but converts them into the *fetch* stage when physical disk I/O is needed, as in OTSTE.

The unique feature of HTSTE is that it does not require a priori knowledge of the transactions' data access behaviors. Instead, the decision of whether a transaction should be executed using TSTE, OTSTE, or a conventional transaction-processing model is made dynamically. By default, HTSTE executes a given transaction type using TSTE, and collects two accumulative statistics for all its instances. The first statistic, called *mis-prefetch ratio*, is the percentage of data accesses that lead to physical disk I/O in the *operate* stage, and the other, called *conversion ratio*, is the percentage of execution instances of this transaction type that require two stages to complete. If a transaction type's *mis-prefetch ratio* is above a certain threshold, all its future instances will be executed as conventional transactions (Type 1). Otherwise if the transaction type's *conversion ratio* is above a certain threshold, all its future instances will continue to be executed under TSTE (Type 2). Otherwise, all the future instances of this transaction type will be executed using OTSTE (Type 3). We use a conservative value as threshold, e.g., only when more than 95% of one type of transaction requires no disk I/O, this transaction type will be started always in OTSTE. For the TPC-C benchmark[2], this prediction works out well: `neword` and `delivery` always require disk I/O during execution; `slev, ostat, payment` seldom do. If a certain transaction type does not display a clear trend in I/O behavior, it will be started as TSTE.

3.4 Cost Analyses

Let P ($P <= 1$) be the percentage of transactions that need to perform physical disk I/O during their lifetime, and let's assume an ideal HTSTE implementation (i.e., All single-phased transactions do not need to perform disk I/O and all two-phased transaction do need to perform disk I/O.), the time to finish one transaction for each TSTE variant is as follows:

TSTE: $Transaction\ Time = T_{fetch} + T_{operate}$ (1)

OTSTE: $Transaction\ Time = P * (T_{rollback} + T_{fetch} + T_{operate}) + (1 - P) * T_{operate}$ (2)

HTSTE: $Transaction\ Time = P * (T_{fetch} + T_{operatie}) + (1 - P) * T_{operate}$ (3)

T_{fetch} is the time spent in the *fetch* stage and $T_{operate}$ is that spent in the *operate* stage. $T_{rollback}$ is the time to undo operations of the previous transaction. Transactions running under TSTE experience both the *fetch* and the *operate* stage. Those running under OTSTE complete within a single stage with a probability of $(1 - P)$ and need to spend an additional rollback overhead with a probability of P. HTSTE categorizes transactions into I/O transactions and non-I/O transactions. Those transactions that need to perform disk I/O are executed in two stages, while those non-I/O transactions start with the *operate* stage directly without going through the *fetch* stage. HTSTE performs the best since it does not incur unnecessary rollback or fetch overhead. The performance difference among different TSTE variants depends on P, the rollback overhead and the relative costs of the *fetch* and *operate* stage. Note that here we are assuming an ideal implementation of HTSTE, which can correctly determine whether a transaction needs to perform disk I/O at run time. This may not be always the case in practice.

3.5 Prototype Implementation

We have implemented the TSTE prototype on the Berkeley DB package, version 2.4.14 [1], which is an industrial-strength transaction processing software library that has been used in several highly visible web-related companies such as Netscape. Berkeley DB supports page-level locking as well as error recovery through write-ahead logging. Berkeley DB supports an asynchronous database logging mode where commit records are not forced to disk, but are written lazily as the in-memory log buffer fills.

The fundamental difference between TSTE and traditional transaction processing systems lies in the *fetch* stage. What the TSTE prototype needs to do is to execute an input transaction twice, to *skip* all the lock acquisition requests and updates in the *fetch* stage and to pin down the database cache pages that are brought into memory in the first stage. When transactions are executed the second time, they are processed the same way as in Berkeley DB. HTSTE dynamically collects internal statistics and decides which type of transaction to start with the *fetch* stage and which to start with the *operate* stage.

To implement OTSTE, we modified Berkeley DB so that it starts each transaction in the *operate* stage, and pins down all the data pages accessed. When a TSTE transaction encounters a database cache miss for the first time, the system converts the transaction from the *operate* stage to the *fetch* stage. During this transition, all the effects of writes are un-

[2] TPC-C contains five types of transactions: `neword, payment, slev, ostat, and delivery`. `Neword` places a new order for a customer. `Payment` clears the balance of a customer. `Slev` checks the stock level. `Ostat` checks the order status. `Delivery` delivers an order.

done, i.e., the database state is rolled back to the beginning of the transaction, and all the locks acquired by this transaction so far are released. Finally, the transaction re-starts by re-executing the missed database cache access, but this time in the *fetch* stage.

Charm requires minor modifications to the programming interface that Berkeley DB provides to its applications to keep track of I/O behavior of each type of transaction online. Although TSTE executes each transaction twice, the operations in the transactions do not have to be idempotent because the execution in the first stage never results in persistent side effects.

4 Performance Evaluation

4.1 Experiment Setup

To evaluate the performance of TSTE, we ran the TSTE prototype on an 800-MHz Pentium-III machine with 640 Mbytes of physical memory running the Linux kernel version 2.2.5 and compared its performance measurements against those from Berkeley DB, which uses a traditional transaction processing architecture. We used the standard transaction processing benchmark TPC-C. This study focuses only on the throughput of `neword` transactions. In the workload mix, `neword` accounts for 43%, `payment` 43%, and `ostat, slev` and `delivery` each 4.7%. *Warehouse* is the parameter to scale the database size. The *Warehouse* parameter (w) in TPC-C was set to five, corresponding to a total database size of over 1.0 GByte, which grows during each test run as new records are inserted. The database cache size used in this study was 512-Mbytes unless otherwise stated.

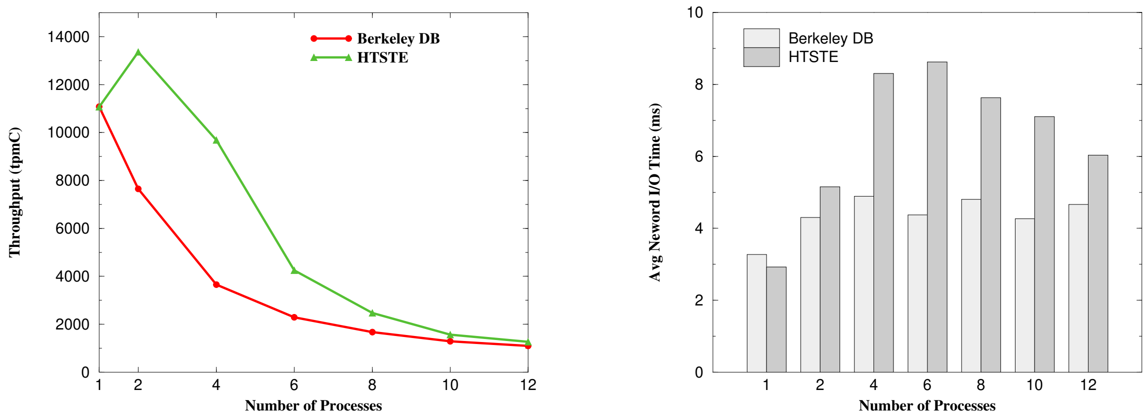
Each data point is the average result of ten runs, each of which consists of more than 10,000 transactions, which is sufficiently long for the system to reach a steady state. Before starting each run, we ran a number of transactions against the database to warm up the database cache. Unless otherwise stated, the number of warm-up transactions is 80,000. Each transaction is executed as a separate user process, and the number of concurrent transactions remains fixed throughout each run. All the following reported measurements are performed at the user level. All the runs assume page-level locking. We set the log buffer size to 500 Kbytes and turned on the asynchronous logging mechanism, which eliminated most disk I/Os associated with transaction commits in our tests. This set-up ensures that the performance bottleneck lies mainly in lock waiting and data accessing disk I/O rather than logging disk I/O. All tests were run on a single computer and thus the reported performance measurements did not include network access delays, as in standard client-server set-ups. In the experiments, we compared Berkeley DB, TSTE OTSTE and HTSTE by varying the following workload/system parameters: the number of concurrent transactions, the database cache size and thus database cache hit ratio, and the computation time.

Three 5400-RPM IDE disks are used in the experiments. The log resides on disk 1, the `orderline` table resides on disk 2 and all other tables reside on disk 3. The write cache of the log disk is turned off to protect data integrity. `Neword` throughput is reported as system throughput as the TPC-C benchmark requires, whose metric unit is tpmC (transaction per minute).

4.2 Overall Comparison

We use the *transaction throughput* as our performance metrics, which is the ratio between the total number of `neword` transactions that succeed and the elapsed time required to complete a run of test transactions. Note that some transactions in the run are aborted to resolve deadlocks. The un-modified Berkeley DB package serves as the baseline case that corresponds to standard transaction processing systems based on two-phase locking. In HTSTE, `neword` and `delivery` transaction start in the *fetch* stage; `payment, ostat` and `slev` start in the *operate* stage.

Figure 2 shows the throughput comparison and Figure 3 shows the lock contention comparison between HTSTE and Berkeley DB when a different number of concurrent transactions are running simultaneously. In this case, the database cache is 512 Mbytes, which gives a database cache hit ratio of 99.7%. HTSTE out-performs Berkeley DB by up to 164% in transaction throughput (when concurrency is four). HTSTE is better than Berkeley DB except in the degenerate case when the number of concurrent transactions is one, where there is no



(a) (b)

Figure 2: Overall system throughput (a) and average I/O time (b) per `neword` transaction comparison between HTSTE and Berkeley DB with different concurrency levels.

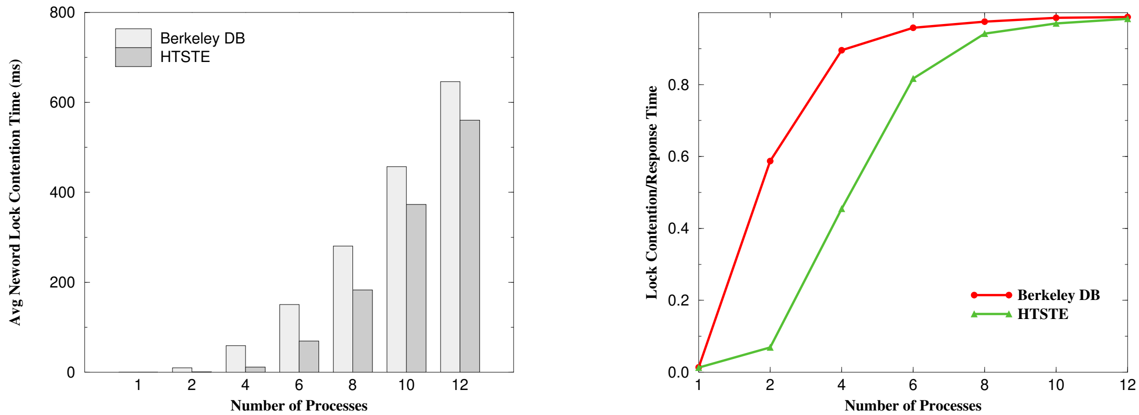
Concurrency	1	2	4	6	8	10	12
Berkeley DB	13.90	17.75	17.85	14.76	13.45	12.21	11.50
HTSTE	13.35	20.74	32.22	26.30	19.63	16.86	13.55

Table 2: Average file system read/write time comparison with different concurrency levels. The values are in terms of msec.

lock contention delay for HTSTE to reduce in the first place. And as concurrency level reaches 10-12, HTSTE and Berkeley DB have minor performance difference. The performance difference between HTSTE and Berkeley DB increases initially with the number of concurrent transactions until the concurrency reaches four. Before this point, lock contention delay dominates the transaction response time and therefore the reduction in lock contention delay that HTSTE affords plays an increasingly important role as lock contention delay increases with concurrency. After this point, the performance difference between HTSTE and Berkeley DB starts to decrease with the number of concurrent transactions because the reduction in lock contention delay becomes less significant percentage-wise with respect to the transaction response time.

To understand why HTSTE out-performs Berkeley DB, let's examine the three components of a transaction's response time: computation, disk I/O, and lock contention. The average computation time of a transaction under HTSTE is only slightly longer than that of Berkeley DB because of TSTE's two-pass execution. Figure 2(b) and Figure 3 show the comparison between HTSTE and Berkeley DB in the average I/O time and lock contention time per `neword` transaction. As expected, HTSTE significantly reduces the lock contention delay, by up to a factor of 10 when compared to Berkeley DB (Figure 3(a) 2 processes). However, HTSTE still experiences noticeable lock contention delays, because mis-prefetching in the *fetch* stage leads to physical disk I/O in the *operate* stage, which in turn lengthens the lock contention time. As transaction concurrency increases, the probability of mis-prefetching grows, and consequently the difference in lock contention delay between HTSTE and Berkeley DB decreases.

Surprisingly, HTSTE fares worse than Berkeley DB in the average disk I/O time when more than two concurrent transactions are running simultaneously (Figure 2(b)). The total number of disk I/Os issued in both cases is roughly the same and the longer average I/O time in HTSTE is due to disk queuing. Table 2 shows that average file system read/write time for HTSTE is longer than Berkeley DB. We consider that the file system read/write time closely reflects the disk I/O latency without file system cache effects. The reason is that in our experiments, the file system cache is minimized



(a) (b)

Figure 3: Average lock contention time per `neword` transaction (a) and the ratio between average lock contention time and average response time (b).

Concurrency	1	2	4	6	8	10	12
Berkeley DB	0.0%	0.7%	1.8%	2.8%	3.7%	4.6%	5.4%
HTSTE	0.0%	0.1%	0.4%	1.6%	3.2%	4.1%	5.0%

Table 3: The percentage of aborted transactions under different concurrency levels.

to almost zero bytes and the Berkeley DB package maintains its own user-level buffer cache. Because HTSTE allows transactions to perform their disk I/Os as soon as possible, it is more likely that disk I/Os are clustered and thus experience longer disk queuing delay. In contrast, under Berkeley DB the disk I/Os are more spread out because transactions are interlocked by lock contention. Therefore the TSTE strategy presents an interesting design trade-off between decreasing lock contention delay and increasing disk I/O time. In general, it is easier to invest more hardware to address the problem of longer disk queuing delay, e.g., by adding more disks, than to lowering the lock contention delay. Therefore, TSTE represents an effective approach to build more scalable transaction processing systems by taking advantage of increased hardware resources.

A minor benefit of HTSTE's reduced lock contention delay is the decreasing number of deadlocks (Table 3). This allows HTSTE to complete more transactions successfully within a given period of time than Berkeley DB, although the contribution of this is less than 2% of the throughput difference between HTSTE and Berkeley DB.

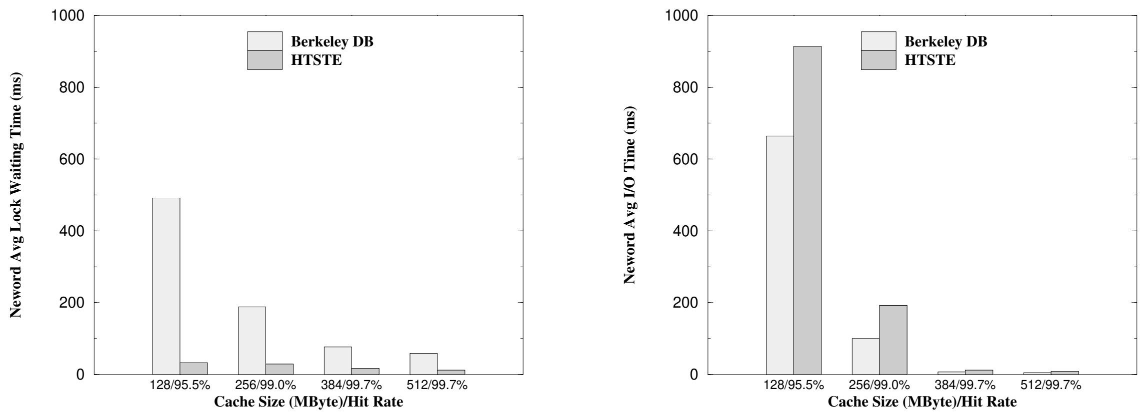
When $w = 1$, the initial database size is reduced to 150 Mbytes, which is even less I/O bound, the experiments show trends similar to those we described in this section. When $w = 10$ or larger, similar trends are observed when the system is running at the non-I/O bound status.

4.3 Sensitivity to Workload/System Parameters

The performance edge of HTSTE over Berkeley DB is most pronounced when there is a "medium" amount of physical disk I/O (i.e., when fewer than 25% transactions perform disk I/O). Figure 4 shows the performance difference between HTSTE and Berkeley DB under different cache sizes. To tune the effective database buffer cache size and thus disk I/O rate, we locked down a certain amount of main memory. Figure 5 gives the breakdown for lock waiting and average I/O time for `neword` transactions as a function of cache size. HTSTE experiences about the same amount of disk I/O as Berke-

Cache Size (Mbytes)	Neword	Payment	Slev	Ostat	Delivery
128	99.8%	71.4%	93.1%	82.3%	98.7%
256	75.5%	21.1%	87.5%	48.1%	97.3%
384	22.0%	3.7%	62.0%	18.1%	84.1%
512	13.3%	1.3%	51.0%	9.4%	71.7%

Table 4: Percentage of transactions that perform I/O during their lifetime with different cache size. The number of concurrent transactions is four. `neword, payment, slev, ostat, delivery` are five types of transactions in TPC-C. The percentages are relative to the total of each type of transaction.



(a) (b)

Figure 5: Lock waiting time (a) and average I/O time (b) per `neword` transaction under different cache size/hit rate.

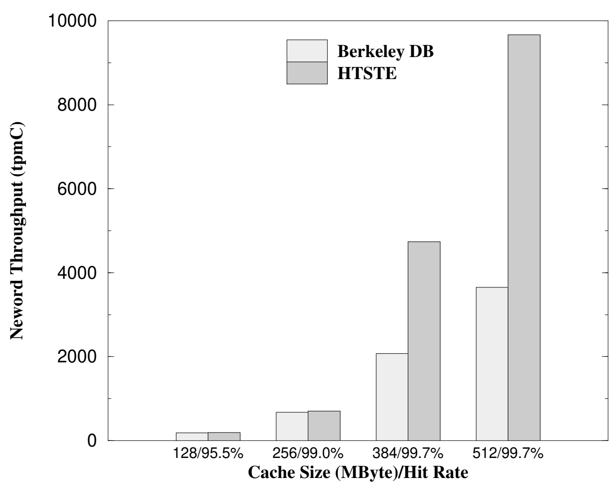


Figure 4: System throughput for `neword` transactions under different cache size/buffer hit rate. Concurrency level is 4.

ley DB. But it suffers more from disk queuing hence longer average I/O time (Figure 5(b)). When the cache hit rate is low, the storage subsystem is the performance bottleneck. The overall throughput is not improved much because the lock waiting time is now transferred to I/O queuing time. Table 5 shows that average file system read/write time for HTSTE is longer than Berkeley DB. The conditions under which TSTE algorithms excel are: some transaction that could proceed without I/O is blocked by some transaction that is holding a lock and doing I/O. If the buffer hit rate is low, the scenario becomes: most transactions need to perform I/O requests during execution, the storage system is stressed so much that I/O queuing is as bad as the cost of lock waiting. This is not the case TSTE can optimize. If the storage system can weather more burst I/O requests, then TSTE can start outperforming 2PL at a lower buffer hit rate.

Table 4 lists the percentage of transactions that need disk I/O under different cache sizes. In

Cache Size (Mbytes)	128	256	384	512
Berkeley DB	112.42	114.14	34.01	17.85
HTSTE	146.94	122.92	46.81	32.22

Table 5: Average file system read/write time comparison with different cache size. The values are in terms of msec. Concurrency level is four.

Concurrency	2	3	4	5	6
w = 1	0.9%	2.1%	2.9%	3.8%	4.4%
w = 5	1.7%	2.1%	2.4%	3.0%	3.3%

Table 6: Percentage of OTSTE transactions that need to perform physical disk I/O in the *operate* stage for varying numbers of concurrent transactions.

HTSTE, relatively less time is spent in lock waiting. As the cache size increases, the database buffer hit ratio improves, and the workload becomes less and less I/O-bound. When the workload is I/O-bound, the disk I/O time dominates the entire transaction response time, and the reduction in lock contention delay that HTSTE enables plays a less significant role in relative terms. On the other hand, when the workload is CPU-bound, there is not much physical disk I/O and HTSTE is not able to achieve a significant reduction in lock contention delays because they are relatively small to begin with. In practice, it is technically impossible for applications that require high transaction processing throughput, i.e., on the order of 1000 transactions per second, to run at an operating point that is I/O-bound. At the same time, it is economically infeasible to eliminate all disk I/Os by having enough memory to hold the entire database. Therefore, we believe that the operating point, at which high-performance transaction processing systems can achieve 1000+ transactions per second, is exactly where a TSTE-like strategy is most useful.

4.4 Detailed Analysis

One potential performance problem associated with TSTE and its variants is *mis-prefetching*, which causes transactions in the *operate* stage to perform physical disk I/Os, and thus prolong lock contention delay. Table 6 shows the average percentage of transactions that actually need to perform physical disk I/Os in their *operate* stage for both $w = 1$ and $w = 5$ cases. The probability of mis-prefetching increases with the degree of lock contention, which in turn grows with the number of concurrent transactions. As the size of the underlying database increases (from $w = 1$ to $w = 5$), the probability of true data contention among a fixed number of concurrent transactions decreases, and so does the probability of mis-prefetching. In general, the absolute percentage of mis-prefetching transactions is quite low for the TPC-C benchmark, under 4.5% for the $w = 1$ case and under 3.4% for the $w = 5$ case. These results conclusively demonstrate that TSTE's performance cost due to mis-prefetching is insignificant for the TPC-C benchmark, and explain why TSTE out-performs Berkeley DB.

Compared to Berkeley DB, TSTE incurs additional processing overhead because it executes a transaction twice. TPC-C benchmark's five transactions all perform much more disk I/O than computation. With a benchmark requiring more computation than TPC-C, TSTE will pay more processing overhead. OTSTE corrects this problem by avoiding this additional processing overhead when all the data pages that a transaction needs are already resident in memory. To evaluate the impact of CPU time on the performance comparison among TSTE, OTSTE and Berkeley DB, we added an idle loop to the end of each TPC-C transaction, and measured the total elapsed time of completing a sequence of 10,000 transactions. By adding an idle loop into each TPC-C transaction, we simulate transactions with larger computation versus disk I/O rate. Table 7 shows the performance comparison among TSTE, HTSTE, OTSTE and Berkeley DB in terms of transaction throughput and average lock waiting time for the following two cases: a 0-msec idle loop and a 30-msec idle loop When the transaction CPU time is small (0-msec case), TSTE's additional processing overhead does not cause serious perfor-

	System Throughput (tpmC)				Average Lock Waiting Time (msec)			
Idle Loop	**TSTE**	**HTSTE**	**OTSTE**	**Berkeley DB**	**TSTE**	**HTSTE**	**OTSTE**	**Berkeley DB**
0 msec	9662	9686	6869	3648	11.8	11.2	20.5	58.9
30 msec	7490	6507	5787	3650	14.7	16.3	26.2	56.6

Table 7: System throughput and average lock waiting time for `neword` transaction with different idle loop padding. $w = 5$. Buffer hit rate is 99.7%. Concurrency is four.

Transaction Type	*Neword*	*Payment*	*Slev*	*Ostat*	*Delivery*
Memory Usage (w = 1)	184	48	1340	64	344
Memory Usage (w = 5)	370	66	2305	104	534

Table 8: The average physical memory requirement for each transaction instance of the five types of transactions in the TCP-C benchmark. All numbers are in terms of Kbytes.

mance problem, and therefore TSTE always outperforms Berkeley DB. On the other hand, the optimistic optimization of OTSTE is not particularly useful when compared to TSTE or HTSTE in this case. Also, HTSTE and TSTE have similar performance. However, when the transaction CPU time is high (30-msec case), the performance difference between TSTE and Berkeley DB decreases because TSTE's additional processing overhead erodes a significant portion of its performance gain from lock contention delay reduction. In this case, OTSTE performs worse than TSTE and HTSTE, as the cost reduction in redundant execution cost does not compensate for the additional lock contention overhead it adds. OTSTE in general requires more lock acquisitions/releases than TSTE because those lock acquisitions/releases made optimistically need to be performed twice when physical disk I/O and thus rollback occurs. HTSTE exhibits a similar problem, but to a lesser degree, and thus shows worse throughput than TSTE. A similar trend is observed in the $w = 1$ configuration.

When data is brought into main memory in the *fetch* stage TSTE pins down those buffer pages to ensure that they remain available in the *operate* stage. Table 8 shows the average physical memory usage of each transaction type in TPC-C, and shows that the maximum physical memory requirement for a TSTE-based system running 100 transactions *simultaneously* is about 230 Mbytes, which is relatively modest for state-of-the-art server-class machines. Therefore the additional memory pressure that TSTE introduces is less of an issue in practice.

5 Conclusion

This paper presents the design, implementation, and evaluation of a novel transaction execution engine called *Charm* that attempts to reduce the lock contention delay due to disk I/O to the minimum. The basic idea is to separate disk I/O from lock acquisition/release so that a transaction is not allowed to compete for locks unless all its required pages are guaranteed to be in memory. With this execution strategy, the lock contention delay and thus the response time of individual transactions are significantly reduced. The total elapsed time for completing a given number of transactions also improves accordingly. Specifically, this paper makes the following contributions:

- We have developed a general two-phase transaction execution (TSTE) scheme to minimize the performance impact of disk I/O on lock contention, and several of its variants to address TSTE's redundant computation and unnecessary rollback problems.

- The TSTE prototype is the first known implementation of a two-phase transaction execution scheme that effectively decouples lock contention from physical disk I/O.

- Empirical performance measurements of the TCP-C benchmark on a working TSTE prototype demonstrate that the best variant of TSTE, HTSTE, can out-perform standard 2PL implementation by up to 164% in terms of transaction throughput, and the optimistic version of TSTE (OTSTE) actually performs worse than generic TSTE and Hybrid TSTE because extra lock contention overhead for OTSTE exceeds the saving in redundant transaction computation when the computation time is comparable or less than disk I/O time.

- TSTE best fits those applications with "medium" physical disk I/O. It cannot improve the performance a lot when storage system is the performance bottleneck.

One performance aspect of transaction processing systems that this research ignores is the I/O overhead associated with transaction commits. While it is a standard industry practice to reduce the commit I/O cost using group commits, the batch size chosen is typically much smaller than is used in this work. We have developed a track-based logging scheme [16] to minimize the performance impacts of the disk I/Os resulting from transaction commits, and plan to integrate TSTE with track-based logging to solve the performance problems of both read and write I/Os in high-performance transaction processing systems.

6 Acknowledgments

We would like to thank Wee Teck Ng, our shepherd Margo Seltzer, and our anonymous reviewers for their valuable feedbacks. We also appreciate the technical support team of Sleepycat Software for replying to our questions regarding the Berkeley DB package.

References

[1] Berkeley DB package, http://www.sleepycat.com.

[2] Seltzer, M.; Olson, M., "LIBTP: Portable, Modular Transactions for Unix," *Proceedings of the Winter 1992 USENIX Conference,* p. 9-25, San Francisco, CA, USA 20-24 Jan. 1992.

[3] Bernstein, A.J.; Wai-Hong Leung; Gerstl, D.S.; Lewis, P.M., "Design and performance of an assertional concurrency control system, *Proceedings 14th International Conference on Data Engineering,* p. 436-45, Orlando, FL, USA 23-27 Feb. 1998.

[4] Badrinath, B.R.; Ramamritham, K., "Synchronizing transactions on objects," *IEEE Transactions on Computers*, vol.37, no.5, p. 541-7, May 1988.

[5] Herlihy, M., "Apologizing versus asking permission: optimistic concurrency control for abstract data types," *ACM Transactions on Database Systems,* vol.15, no.1, p. 96-124, March 1990.

[6] Kung, H.T.; Robinson, J.T., "On optimistic methods for concurrency control," *ACM Transactions on Database Systems,* vol.6, no.2, p. 213-26, June 1981.

[7] Smith, J.E.; Weiss, S.; Pang, N.Y., "A simulation study of decoupled architecture computers," *IEEE Transactions on Computers,* vol.C-35, no.8, p. 692-702, Aug. 1986.

[8] Mitra, T.; Yang, C.K., "File system extensions for application-specific disk prefetching," ECSL-TR-64, Computer Science Department, SUNY at Stony Brook, January 1999.

[9] Jung-Ho Ahn; Hyoung-Joo Kim, "SEOF: an adaptable object prefetch policy for object-oriented database systems," *Proceedings 13th International Conference on Data Engineering,* p. 4-13, Birmingham, UK. April 7-11, 1997.

[10] Kimbrel, T.; Cao, P.; Felten, E.W.; Karlin, A.R.; Li, K., "Integrated parallel prefetching and caching," *Performance Evaluation Review,* vol.24, no.1, p. 262-3, ACM May 1996.

[11] Lehman, P.L.; Yao, S.B., "Efficient locking for concurrent operations on B-trees," *ACM Transactions on Database Systems,* vol.6, no.4, p. 650-70, Dec. 1981.

[12] Franaszek, P.A. et al. "Concurrency control for high contention environments," *ACM Transactions on Database Systems*, June 1992. vol.17, no.2, p. 304-45.

[13] Franaszek, P.A. et al. "Access invariance and its use in high contention environments," *Proceedings of Sixth International Conference on Data Engineering,* p. 47-55, Los Angeles, CA, USA 5-9 Feb. 1990.

[14] Reuter, A., "The transaction pipeline processor," *Proceedings of the International Workshop on High Performance Transaction Systems,* Pacific Grove, CA, Sep. 1985.

[15] Chang, F.; Gibson, G.A., "Automatic I/O hint generation through speculative execution," *Proceedings of Third Symposium on Operating Systems Design and Implementation,* p. 1-14, New Orleans, LA, USA 22-25 Feb. 1999.

[16] Chiueh, T.; Huang, L.; "Trail: A Fast Synchronous Write Disk Subsystem Using Track-based Logging," ECSL-TR-68, Computer Science Department, SUNY at Stony Brook, June 1999.

[17] Olson M.; Bostic K.; Seltzer M., "Berkeley DB," *Proceedings of the 1999 Summer Usenix Technical Conference,* Monterey, California, June 1999.