BP-Wrapper: A System Framework Making Any Replacement Algorithms (Almost) Lock Contention Free

Xiaoning Ding[1], Song Jiang[2], Xiaodong Zhang[1]

[1]Computer Science and Engineering Department
The Ohio State University
Columbus, OH 43210, USA
{dingxn,zhang}@cse.ohio-state.edu

[2]Electrical and Computer Engineering Deptartment
Wayne State University
Detroit, MI 48202, USA
sjiang@eng.wayne.edu

Abstract— **In a high-end database system, the execution concurrency level rises continuously in a multiprocessor environment due to the increase in number of concurrent transactions and the introduction of multi-core processors. A new challenge for buffer management to address is to retain its scalability in responding to the highly concurrent data processing demands and environment. The page replacement algorithm, a major component in the buffer management, can seriously degrade the system's performance if the algorithm is not implemented in a scalable way. A lock-protected data structure is used in most replacement algorithms, where high contention is caused by concurrent accesses. A common practice is to modify a replacement algorithm to reduce the contention, such as to approximate the LRU replacement with the clock algorithm. Unfortunately, this type of modification usually hurts hit ratios of original algorithms. This problem may not exist or can be tolerated in an environment of low concurrency, thus has not been given enough attention for a long time.**

In this paper, instead of making a trade-off between the high hit ratio of a replacement algorithm and the low lock contention of its approximation, we propose a system framework, called *BP-Wrapper*, that (almost) eliminates lock contention for any replacement algorithm without requiring any changes to the algorithm. In BP-Wrapper, we use batching and prefetching techniques to reduce lock contention and to retain high hit ratio. The implementation of BP-Wrapper in PostgreSQL version 8.2 adds only about 300 lines of C code. It can increase the throughput up to two folds compared with the replacement algorithms with lock contention when running TPC-C-like and TPC-W-like workloads.

I. Introduction

A modern database management system normally manages terabytes of data and processes hundreds of thousands of transactions concurrently on a powerful multiprocessor system. Such a database system demands buffer management be highly effective to minimize costly disk I/O operations, and be scalable with the growing number of concurrent transactions and the increase of the number of processors in the underlying system. The core of buffer management is the data replacement algorithm, which makes decisions on which data pages should be cached in memory to absorb effectively requests for on-disk data from upper level transaction-processing threads. The replacement algorithm usually maintains a complex data structure to track the data-access history of the threads so that replacement decisions can be made based on the organized information in the data structure. In a high-end production system, a large number of threads access data pages frequently and concurrently. It is desirable that the replacement algorithm makes replacement decisions both in an *effective* and *scalable* way.

Disk I/O operations are becoming increasingly expensive. In order to minimize costly disk accesses, a large number of replacement algorithms and their implementations have been proposed for databases, virtual memory, and I/O buffers, focusing on the improvement of hit ratios by organizing and managing deep page access history. Representative algorithms include those that address the weaknesses of LRU, such as LIRS [1], 2Q [2], and ARC [3]. These algorithms usually take actions upon each I/O access, either a hit or a miss in the buffer, which include a sequence of updates in their data structures recording the data access history. Usually the operations are designed to be simple and efficient to avoid excess overhead, as they are carried out very frequently.

To guarantee the integrity of their data structures, replacement algorithms carry out their operations in response to page accesses in a serialized fashion. Thus, to implement them in multi-tasking systems, lock synchronization is usually required. In other words, an exclusive lock (a.k.a. latch) must be secured before the operations are carried out. When a lock is held by one transaction-processing thread, other threads requesting the lock have to wait in the form of busy-waiting and/or context switches for their mutual exclusive operations.

Performance degradation due to lock contention can be significant in large-scale systems. This subject has been a major research issue for years (e.g. [4], [5], [6]). Our experiments show that contention on the lock associated with replacement algorithms may reduce database throughput by nearly two folds in a 16-processor system. There are two factors contributing to the performance degradation. One factor is the frequency of lock requests. Currently, the lock is globally shared by all transaction-processing threads to coordinate their page accesses. It is requested every time a thread accesses a page. Another factor is the cost to acquire a lock, including changing lock state, busy-waiting, and/or context switches. Compared to the time spent on the operations protected by the lock, the cost can be very high. As the number of multi-core processors in a multi-processor system increases to support DBMS systems, the performance degradation is expected to continue to grow since both of the factors can become more severe in a larger system.

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.96

While the replacement algorithm designers have not paid particular attention to the lock contention issue, the advantages of the algorithms, including high hit ratio, could be compromised in real systems. For example, some widely used DBMS systems, such as postgreSQL, have not adopted advanced replacement algorithms. Instead, they resorted to the clock-based approximations of the LRU replacement. Some of the other DBMS systems, such as Oracle Universal Server [7] and ADABAS [8], choose to use the distributed lock method to address the lock contention issue.

The clock-based approximations, such as CLOCK [9], CLOCK-PRO [10], and CAR [11], usually cannot achieve the high hit ratio compared to their corresponding original algorithms (LRU, LIRS, or ARC, respectively). They organize buffer pages into circular list(s), and use a reference bit or a reference counter to record access information for each buffer page. When a page is hit in the buffer, the clock-based approximations set the reference bit or increment the counter, instead of modifying the circular list(s) themselves. As a lock is not required for these operations, their caching performance is scalable. However, the clock-based approximations can record only limited history access information, i.e. whether a page has been accessed or how many times it has been accessed but not in what order the accesses occur. The lack of richer history information can hurt their hit ratios. Moreover, many sophisticated replacement algorithms do not have clock-based approximations since the access information they need cannot be approximated by the clock structure. Examples include the SEQ [12] algorithm and the buffer replacement policy used in DB2 [13], as they need to know in which order the buffer pages are accessed for the detection of sequences and sequential/random access patterns.

In general, lock contention can be reduced by using distributed locks to reduce lock granularity [4], [5], [6], [14]. However, the approach is not effective in addressing the issue of lock contention in the replacement algorithms. In the distributed lock approach, the buffer is divided into multiple partitions, each of which is protected by a local lock. Data pages are evenly distributed into the partitions either in a round-robin manner or through hashing. As only accesses to the same partition will compete for the same lock, lock contention can be ameliorated. However, as the recorded history information is localized to each partition, the lack of global history information can be harmful to the performance of the replacement algorithms. For example, the algorithms that need to detect sequence of accesses cannot retain their performance advantages when pages in the same sequence have been distributed into multiple partitions and cannot be identified as a sequence.

In summary, existing efforts on the research and development of replacement algorithms in DBMS systems have been focused on addressing the trade-offs between high hit ratio of advanced algorithms and low-lock-contention implementation in systems. Instead of making a compromise between these two metrics, our objective is to retain the performance advantages of advanced replacement algorithms and provide an efficient framework that makes any replacement algorithms (almost) lock contention free. With a small FIFO queue maintained for each DBMS transaction-processing thread, our framework provides two key scalability supports, which can be universally applied to any replacement algorithms. One is *batch execution*, which amortizes lock contention overhead among a batch of page accesses. The other is *prefetching*, which reduces the average lock-holding time by pre-loading necessary data for the replacement algorithm into the processor cache. We name the framework employing **B**atching and **P**refetching as *BP-Wrapper*. Our implementation of BP-wrapper in PostgreSQL version 8.2.3 has delivered a nearly two-fold throughput increase by removing almost all lock contention associated with buffer page replacement for TPC-W-like and TPC-C-like workloads.

The rest of the paper is organized as follows. In Section II we briefly describe the role that a replacement algorithm and its lock play in a typical database system. In Section III we describe the design of BP-Wrapper. Section IV provides a comprehensive evaluation of BP-wrapper. Related work is in Section V, and Section VI concludes the paper.

II. Background on Buffer Management in DBMSs

In a DBMS system, its buffer stores a collection of buffer pages of fixed sizes in the memory space shared by all the transaction-processing threads in a DBMS. Data pages read from the hard disk are cached in the buffer to avoid costly I/O operations if they are expected to be reused in the near future. The metadata of the buffer pages are organized by the buffer manager using data structures such as linked lists and hash tables. The metadata include identifiers of the cached data pages, statuses of the pages, and pointers to form linked lists and hash tables.

Figure 1 shows the diagram of a typical buffer manager. When a data page is requested by a thread, the thread searches for the buffer page containing the requested data page. Usually a hash table is used to speed up the searching. If the buffer page is found (a hit), an operation described by its replacement algorithm is carried out to update the data structures to reflect the page access. For example, when a buffer page is requested, the LRU replacement algorithm removes the buffer page from the LRU list and inserts it back to the MRU end of the list. Then the buffer page is returned to complete the request. In another scenario where the requested data page is not in the buffer (a miss), the replacement algorithm selects a victim page and evicts the data in the page to make room for caching the data to be loaded. The LRU algorithm always selects the buffer page at the tail of an LRU list as the victim page. When the data page is read into memory, the buffer page is moved to the head of the LRU list and returned to satisfy the request.

Because the buffer manager is a central component frequently used by all the transaction-processing threads upon each page request, simultaneous updates on its data structures have to be carried out in a controlled fashion to maintain integrity. DBMSs use exclusive locks for this purpose.

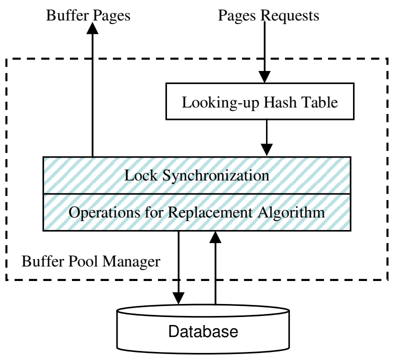

Fig. 1. The diagram of a buffer manager in a typical DBMS. In the buffer manager, which is enclosed in dashed-line rectangle, looking up the hash table can be executed concurrently, while operations for replacement algorithm represented by the shadowed block have to be serialized.

The use of locks does not limit system scalability for hash table searching because (1) In the hash table, metadata of buffer pages are evenly distributed into hash buckets. One lock for each bucket, instead of a global lock, is used to control the accesses to the bucket. To keep the searching time short, usually a large number of buckets are used. Therefore, the possibility for multiple threads to compete for the same bucket is low. (2) Multiple threads can search the same bucket simultaneously if none of them changes the bucket. Hash buckets are rarely changed as they change only upon misses and only when two hash buckets are changed for one buffer miss (one bucket for the victim page, and the other for the new page). In the DBMSs with a high memory capacity, only a very small portion of accesses are buffer misses. Therefore, we will not consider the lock contention on the hash table searching.

In contrast, the replacement algorithm can severely suffer from lock contention, which further limits the scalability of database systems because (1) A replacement algorithm uses a single lock for its data structure, which is a centralized hot spot. (2) Most replacement algorithms require an update of their data structures upon every page access. Therefore, a thread has to acquire the lock for every page request and execute the replacement algorithm operations exclusively. The highly contented lock may dramatically degrade system performance in a DBMS system running on a multiprocessor system.

III. Minimizing Lock Contention with BP-Wrapper

In a replacement algorithm, the cost associated with the use of a lock consists of two parts, namely lock acquisition cost and lock warm-up cost. Lock acquisition cost is the time overhead due to the fact that a thread has to block itself when its requested lock is held by another thread. Depending on the implementation, the cost can be the CPU cycles for busy waiting in the spinning lock and/or CPU cycles for context switches. This cost is determined by the severity of lock contention among the threads. If there are many processors in a server that processes a large number of concurrent transactions, the time spent on the acquisition of a lock can be significantly higher than a system of smaller scale. In other words, this cost is directly linked to the scalability of the replacement algorithm because each page access accompanies a lock acquisition for the access history to be recorded in a lock-protected shared data structure.

The lock warm-up cost refers to the cost paid to prepare the processor cache with the required data to run critical section code, or the penalty of processor-cache misses incurred by the transition of computation from non-critical section (code for transaction processing) into critical section (code for updating the shared data structure for replacement algorithms). When a lock is obtained, the data describing the lock as well as the data set to be accessed by the critical section code may not yet reside in the processor caches, thus a series of misses may be experienced to warm up the cache. The concern is that this miss penalty occurs when a thread holds the lock and there are probably other threads waiting for the lock. Thus, the impact of this cost could be amplified. Following the principle of minimizing the critical section, we aim to eliminate the lock warm-up cost.

To reduce these two potential lock costs, we design and integrate two methods in BP-Wrapper. Both methods are independent of the replacement algorithm itself, so that BP-Wrapper can be used directly with *any* replacement algorithms and to make them highly scalable.

A. *Reducing Lock Acquisition Cost Using Batching Technique*

As we mentioned, a lock must be acquired before a thread enters the critical section to carry out its operations on behalf of the replacement algorithm. Requesting a lock upon a page miss usually is not a concern because the lock acquisition cost is negligible compared with the cost of I/O operations. In addition, there are much fewer misses than hits in a database system. The challenge is that most replacement algorithms, especially those recently proposed algorithms that produce very low miss ratios [1], [2], [3], need to access lock-protected data for each page access, even if the access is a hit in the buffer. As we know, lock acquisition cost increases as lock contention intensifies, and becomes a serious performance bottleneck when the system scales up.

Replacement algorithms in existing database systems require one lock-acquisition per page access. The total cost of lock-acquisitions increases as the page access frequency increases, which is caused by high concurrency in database systems. We use a technique called *batching* to amortize the cost over multiple page accesses and essentially reduce the cost. The basic idea is to periodically acquire a lock after accumulating a set of page accesses, and then to make corresponding replacement operations within one lock-holding period. To understand the performance potential of the technique, we have conducted an experiment in a 16-processor system (the detailed configuration is described in Section IV) to measure the duration in which the lock is requested and held by a thread (lock-holding time) for processing a certain

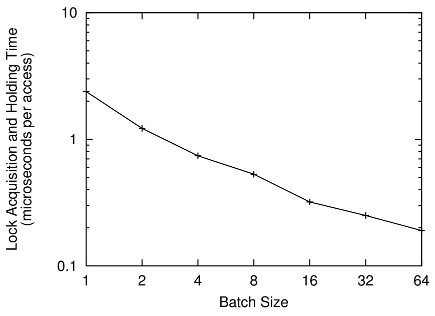

Fig. 2. Average lock acquisition and holding time per each page access with batch size varied from 1 to 64. Both axes in the figure are in the logarithmic scale. The workload is DBT-1, and the number of processors is 16.

number (batch size) of page accesses on the lock-protected data structure for the 2Q replacement algorithm, including the lock acquisition cost. We varied the batch size from 1 to 64 (i.e. the number of accumulated page accesses before acquiring a lock is varied from 1 to 64). Figure 2 shows the lock acquisition and holding time averaged over batch size. We found that the time is much larger with smaller batch sizes. While the total execution time in the critical section for multiple accesses is proportional to the number of accesses, the measurements convincingly show the effectiveness of our batching technique. The experiment also shows that a small number of batch size such as 64 is sufficient to significantly reduce the lock acquisition cost.

While each page access has to operate on the lock-protected data, as required by the replacement algorithms, it is usually unnecessary to carry out the operations immediately after the page access. There are two unique properties of the replacement algorithms that provide us with an opportunity to effectively apply a batching technique to significantly reduce the frequency of acquiring the lock. First, delaying the operations on the data structure for replacement algorithms, such as LRU stack or LIRS stacks [1], will not affect the threads getting correct data from the buffer, and thus will not affect the correctness of transaction processing. Second, in a system with millions of pages (e.g., a server used in our experiments has 64GB memory, or 8 million 8KB pages.), postponement of the operations of recording a couple of (e.g., 64) recent page accesses into the lock-protected data structure would cause little impact on the performance of the replacement algorithms. Furthermore, the order in which the batched operations are executed does not change with the adoption of the batching technique.

In the batching technique, we set up a FIFO queue for each transaction-processing thread. For each access hit associated with a thread, the access history information is recorded in the thread's queue. Specifically, when a thread requests a page and the page is found to be in the buffer, the pointer to the page is recorded in the FIFO queue of the thread. A buffer manager using the technique is shown in Figure 3. When the queue is full or the number of accesses recorded in the queue reaches a pre-determined threshold, called the *batch threshold*, we acquire a lock and then execute the operations defined by the replacement algorithm once for all the accesses in the queue in a batching fashion. This procedure is known as *committing* the recorded accesses. After the committing, the queue is emptied. With the FIFO queue, a thread is allowed to access many pages without requesting a lock for running the page replacement algorithm, or without paying the lock acquisition cost.

In the design of the batching technique, an alternative is to use one common FIFO queue shared by multiple threads. However, we choose to use a private FIFO queue for each thread for its following advantages:

- A private FIFO queue keeps the precise order of the page accesses that occur in the corresponding thread. Keeping the order is essential in some replacement algorithms like SEQ [12] because they need the ordering information for detection of access patterns.
- Recording access information into private FIFO queues incurs the least synchronization and coherence cost, which is required for the shared FIFO queue when multiple threads fill or clear the queue.

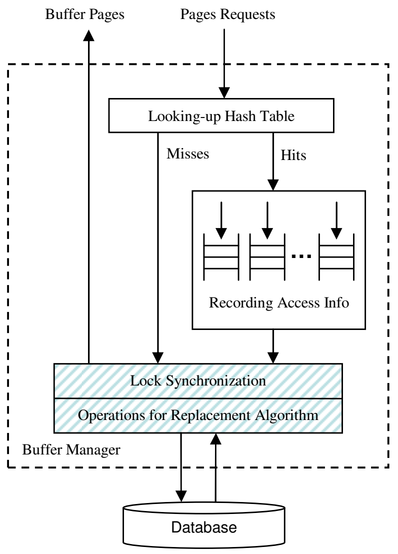

Fig. 3. The diagram of a buffer manager using the batching technique.

Figure 4 presents the pseudo-code describing the batching technique, including lock-related operations upon a page hit (replacement_for_page_hit()) and a page miss (replacement_for_page_miss()). As described in the pseudo-code, when there is a page hit, the access is first recorded in the queue (Queue[]). Then there are two conditions under which the committing procedure is activated and the actual replacement algorithm is executed. The first condition is that there are sufficient number of accesses in a queue (not fewer than *batch_threshold*) and the lock is available for free (the outcome of TryLock() is a success). While the lock acquisition cost is usually high, it is not desirable to pay the cost for a very

```
   /* a thread's FIFO queue whose maximum size is S*/
1  Page *Queue[S];
   /* the minimal number of pages in Queue[]
      to trigger a committing */
2  #define batch_threshold T
   /* current queue position to receive next page */
3  int Tail = 0;
   /* called upon a page hit */
4  void replacement_for_page_hit(Page *this_access) {
5    Queue[Tail] <-- this_access;
6    Tail = Tail + 1;
7    if (Tail >= batch_threshold)
        /* a non-blocking attempt to acquire lock */
8       trylock_outcome = TryLock();
9    if(trylock_outcome is a failure) {
10      if( Tail < S)
11        return;
12      else
13       Lock();
14   }
     /* A lock has been secured. Now commit the pages */
15   For each page P in Queue[]{
16     do what is specified by the replacement
         algorithm upon a hit on P;
17   }
18   UnLock();
19   Tail = 0;
20 }

   /* called upon a page miss */
21 void replacement_for_page_miss(Page *this_access) {
22   Lock();
23   for each page P in Queue[] {
24     do what is specified by the replacement
         algorithm upon a hit on P;
25   }
26   do what is specified by the replacement
       algorithm upon a miss on 'this_access';
27   UnLock();
28   Tail = 0;
29 }
```

Fig. 4. The pseudo-code for the replacement-algorithm-independent framework that uses the batching technique to provide the algorithm with efficient access to the data that needs lock protection.

small number of accesses. TryLock() makes an attempt to get the lock. If the lock is currently held by another thread, it fails without blocking its caller thread. Otherwise, the caller thread gets the lock with a very low cost. Though TryLock() is inexpensive, we do not use it for every page access to keep from producing too many lock acquisition attempts and reducing the chance for a TryLock() to succeed. The second condition is that the queue is full. In this case, a lock must be explicitly requested (Lock() in line 13). If either condition is met, the replacement algorithm starts to carry out its delayed bookkeeping work on its lock-protected data structure for each access recorded in the queue. If we take the LRU algorithm as an example, the work is to move pages involved in every access to the MRU end of the list. Note that the pseudo-code actually describes a framework that uses the batching technique to provide *any* algorithm with efficient access to the data that have to be lock-protected, because the description of the replacement algorithm itself is independent of the batching technique. No design of an existing replacement algorithm, which may have been given significant effort for the improvement of its hit ratio, has to be modified to accommodate the application of the technique for lock overhead reduction. This is in contrast to the transformation of an algorithm to its clock approximation for reduced lock contention but with a compromised replacement performance.

B. *Reducing Lock Warm-up Cost Using Prefetching Technique*

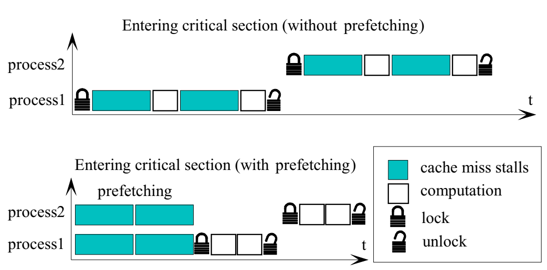
Entering critical section (without prefetching)
process2
process1
t
Entering critical section (with prefetching)
prefetching
process2
process1
t
cache miss stalls
computation
lock
unlock

Fig. 5. Using prefetching to move the cache miss penalty out of the lock holding period.

We use a prefetching technique to reduce the lock warm-up cost, which is part of the lock holding time. In the technique, we read the data that would be accessed in the critical section by the replacement algorithm immediately before a lock is requested. Taking the LRU algorithm as an example, we read the forward and/or backward pointers involved in the movement of accessed pages to the MRU end of the page list, as well as the fields of the lock data structure. A side-effect of the read is that the data are loaded into the processor cache and the cache misses otherwise experienced by the critical section code are removed. The potential benefit of the technique is illustrated in Figure 5.

The prefetching operation on the shared data without a lock does not compromise the integrity of the global data structure used in the replacement algorithm. The prefetching (read) operation only loads the data into processor cache. It does not modify any data. Meanwhile, if the prefetched data has been changed by other threads before they are used by the thread prefetching them, some hardware mechanism built in processors will automatically invalidate them from the cache or update them with their latest values to keep data coherent.

IV. Performance Evaluation

We have implemented the proposed PB-Wrapper framework, including the two lock contention reduction techniques, on the postgreSQL database system version 8.2.3. PostgreSQL used LRU and 2Q replacement algorithms in its previous versions, and gave them up due to the scalability issue. Since version 8.1, postgreSQL adopted the clock replacement algorithm in order to improve the scalability of its buffer management on multi-processor systems. The clock replacement algorithm does not need a lock upon hit access. In this sense, it eliminates lock contention and provides optimal scalability.

In the first part of our evaluation, we focus only on the scalability issue, and show that, using BP-Wrapper an advanced replacement algorithm like 2Q can be as scalable as the clock replacement algorithm, in spite of their more complex data structures and operations. In the experiments, we set the buffer large enough to hold the whole working sets of the benchmarks and pre-warm the buffer. Thus there are no misses incurred no matter which replacement algorithm is used. The performance differences among the postgreSQL systems with different buffer management implementations result completely from the differences in the scalability of their implementations, rather than hit ratios. Then the better performance we observe about an implementation, the more scalable it is. With its optimal scalability, the clock implementation should show the best performance when the system is scaled up. Thus we test the scalability of an implementation by measuring how close its performance is to that of the clock implementation. We show the results in Section IV-D.

Using the experiments in the first part, we aim to answer the following questions specifically. (1) Which technique is more effective in reducing lock contention of a replacement algorithm, batching or prefetching? (2) Compared with that of the clock algorithm, how much lock contention can be reduced for a replacement algorithm using the techniques? (3) Do the techniques used on a multi-processor platform and a multi-core platform have different performance impacts?

There exist a variety of advanced replacement algorithms that can provide excellent performance (much better than the clock algorithm) in terms of hit ratio. However, while LRU can be transformed into the clock algorithm, many of these algorithms are very hard, if not impossible, to be effectively transformed and thus are not appropriate choices in an environment of high concurrency. In the second part of our evaluation, we show that, if no actions are taken to reduce lock contention, the performance advantages of an advanced replacement algorithm due to its increased hit ratios can be compromised in a large scale system. Meanwhile, we show that our techniques help an advanced replacement algorithm retain its performance advantage by improving its scalability.

A. Tested Systems

We first modified postgreSQL 8.2.3 by replacing its clock algorithm with the 2Q algorithm, as a representative of the advanced replacement algorithms of high hit ratios. This modified system, which was not optimized for lower lock contention, is named as *postgreSQL-2Q*, or simply *pg2Q*, and serves as a baseline system in the comparison. Then we enhanced the baseline system with our BP-Wrapper framework. We enabled the batching technique and the prefetching technique separately, and have systems named as *pgBatching* and *pgPref*, respectively. We also enabled both batching and prefetching techniques, and name the system as *pgBat-Pre*. The stock postgreSQL 8.2.3 is denoted as *pgClock*. These tested systems are summarized in Table I. We also implemented systems by replacing the 2Q algorithm with the last four systems in the table with the LIRS [1] and MQ [15] replacement algorithms, respectively. We do not observe significant performance differences between the experiments with these algorithms and those with their 2Q counterparts, and we do not show their results.

TABLE I
THE SYSTEM NAMES, REPLACEMENT ALGORITHMS, AND SCALABILITY ENHANCEMENTS OF THE FIVE TESTED POSTGRESQL SYSTEMS.

Name	Replacement	Enhancement
pgClock	Clock	None
pg2Q	2Q	None
pgBatching	2Q	Batching
pgPref	2Q	Prefetching
pgBat-Pre	2Q	Batching and Prefetching

B. Implementation Issues

The implementation of batching and prefetching requires only limited modification of the baseline system. We add fewer than 300 lines of new code. Most modifications are made in a single file (src/backend/storage/buffer/freelist.c), which contains the source code of the replacement algorithm. In the implementation, each entry in the FIFO queues consists of two fields: one is a pointer to the meta-data of a buffer page (BufferDesc structure), and the other stores BufferTag, which is used to identify a data page. Before an entry is committed, we first compare the BufferTag in the entry against the BufferTag in the meta-data of the buffer page to ensure that the data page has not been invalidated or evicted. If the buffer still caches the valid data page, we run replacement-related operations to update the data structure to reflect the page access. In the 2Q algorithm, if the page is in Am list, it is moved to the MRU end of the list. In the LIRS algorithm, it is moved on the LIR or HIR lists. In the MQ algorithm, it is moved among multiple FIFO queues.

C. Experiment Setup and Workloads

We carried out our experiments on both a traditional unicore SMP platform and a multi-core platform. The unicore SMP platform is an SGI Altix 350 SMP server with 16 1.4GHz Intel Itanium 2 processors and 64GB memory. The storage is a 2TB LUN on an IBM FAStT600 turbo storage subsystem. The LUN consists of 9 250GB SATA disks in an 8+P RAID5 configuration. Operating system is Red Hat Enterprise Linux AS release 3 with SGI ProPack 3 SP6. The multi-core platform is a DELL PowerEdge 1900 server, which has two 2.66GHz quad-core Xeon X5355 processors. For the sake of convenience, we refer to each computing core as a processor as most operating systems do. So the PowerEdge 1900 server has 8 computing cores, or 8 processors. The memory size is 16GB. The storage is a RAID5 array with 5 15K RPM SCSI disks. Operating system is Red Hat Enterprise Linux AS release 5.

We tested the systems with the *DBT-1* test kit and the *DBT-2* test kit from OSDL database test suite [16], and a synthetic benchmark *TableScan*. *DBT-1* simulates the activities of web users who browse and order items from an on-line bookstore. It generates a database workload with the same characteristics as that in the TPC-W benchmark specification version 1.7 [17]. The database generated for the experiments includes information on 100,000 items and 2.9 million customers. *DBT-2* derives from the TPC-C specification version 5.0 [18] and provides an on-line transaction processing (OLTP) workload. In the experiments, we set the number of warehouses to 50. *TableScan* simulates sequential scan, one of most commonly used database operations. It makes concurrent queries, each of which scans an entire table. Each table consists of 800,000 rows, and each row is 128 bytes long.

In the experiments, we changed the numbers of processors used by postgreSQL by setting CPU affinity masks of its *back-end processes*, which are threads in charge of handling transactions in postgreSQL. Because a postgreSQL back-end process blocks itself and yields the processor when it fails to get a lock due to contention, we make the system over-committed and the processors always busy by keeping more active postgreSQL back-end processes than the number of processors used in each test. In the experiments, we set the FIFO queue size to 64, and batch threshold to 32 for *pgBatching* and *pgBat-Pre*.

D. Experiments on Scalability

In the experiments, we evaluate the scalability of the five different postgreSQL systems under the three workloads, respectively, when we increase the numbers of processors from 1 to 16 on the Altix 350 server and from 1 to 8 on the PowerEdge 1900 server. To eliminate page misses during the running of the workloads, we adjust the shared buffer size to ensure that the entire working sets are always held in the memory. We collect the throughputs (number of transactions per second) and the average response time of the transactions. For systems *pg2Q*, *pgBatching*, *pgPref*, and *pgBat-Pre*, we also calculate average lock contention. A lock contention happens when a lock request cannot be immediately satisfied and a process context switch occurs. During the running of a workload, the *average lock contention* is defined as the number of lock contentions per million page accesses. We show throughputs, average response times, and average lock contention for the three workloads on the different systems in Figure 6 and Figure 7 .

When the number of processors is increased, as we expect, the throughput of *pgClock* increases almost linearly with it, and its average response time increases moderately with workloads *DBT-1* and *TableScan*. However, with workload *DBT-2*, the throughput of *pgClock* increases sub-linearly and the average response time increases significantly. This is because the contention on other locks, such as the one to serialize Write-Ahead-Logging activities, becomes intensive with the growing number of processors.

System *pg2Q* can maintain its scalability only when the number of processors is less than 4. On the Altix 350, its throughput saturates when the number of processors is greater than 8 for *DBT-1* and *TableScan*, and 4 for *DBT-2*, respectively, and the average response time increases significantly when additional processors are added. For workload *TableScan*, its throughput even drops by 12.7% when the number of processors is increased from 8 to 16. When 16 processors are used, its throughputs are 61.1%, 56.5%, and 66.5% less than those of pgClock, and its average response times are 1.6, 1.5, and 1.8 times longer than those of *pgClock* for workloads *DBT-1*, *DBT-2*, and *TableScan*, respectively. By examining the plots of average lock contentions, we see that *pg2Q* has the highest number of contentions per million page accesses and the number increases rapidly with the number of processors (Note that the numbers are shown in logarithmic scale). Therefore, these experiments indicate that the lock contention is a major culprit of the system performance degradation.

We observe a similar trend on the PowerEdge 1900 server. The throughput of *TableScan* saturates even earlier, or when the number of processors reaches 4. The average lock contention numbers indicate that lock contention is more intensive on the multi-core PowerEdge 1900 than that on the Altix 350, especially with benchmark *TableScan*. When 8 processors are used, the average lock contentions on the PowerEdge 1900 are 74.4%, 18.5%, and 270.2% more than those on the Altix 350 for the three workloads, respectively. Thus, lock contention causes more performance degradation on the PowerEdge 1900 than it does on the Altix 350. With 8 processors, the throughputs of system *pg2Q* are 37.9%, 52.1%, and 57.2% less than those of pgClock on the PowerEdge 1900 system, while the throughputs of system *pg2Q* are 30.1%, 51.5%, and 32.6% less than those of *pgClock* on the Altix 350 system. Similarly, with system *pg2Q*, lock contention increases the average response times of the workloads by larger percentages on the PowerEdge 1900 than it does on the Altix 350.

The more intensive lock contention on the PowerEdge 1900 server is due to the processors used in the machines. The Xeon X5355 processors in the PowerEdge 1900 have data prefetching modules, which can speed up sequential memory accesses by fetching data speculatively to their last-level caches. However, Itanium 2 processors do not have such hardware support. Thus, on the PowerEdge 1900 server, computation outside of the critical section, which accesses memory sequentially in general, is accelerated by the prefetching modules, while the operations of the replacement algorithm protected by the lock can hardly be accelerated by the prefetching modules because they usually access memory randomly. Therefore, lock contention is intensified on the PowerEdge 1900 server as a larger proportion of time is spent on the critical section than that on the Altix 350.

Compared with *pg2Q*, *pgPref* reduces the average lock contention by 33.7% to 82.6% on the Altix 350 server and by 20.8% to 87.5% on the PowerEdge 1900. This is because prefetching reduces the lock holding time and accordingly increases the chance to get a free lock. As a result, the throughputs of *pgPref* are larger than those of *pg2Q* by up to 26.1% and the average response times of *pgPref* are smaller than those of *pg2Q* by up to 25.2%. We note that prefetching is more effective on the Altix 350 than on the PowerEdge 1900. This is because long pipelines and deep out-of-order execution capability of X5355 processors increase their ability to tolerate cache misses. Based on this observation, we expect that the prefetching technique would be more effective in reducing lock contention on systems with large-scale multi-

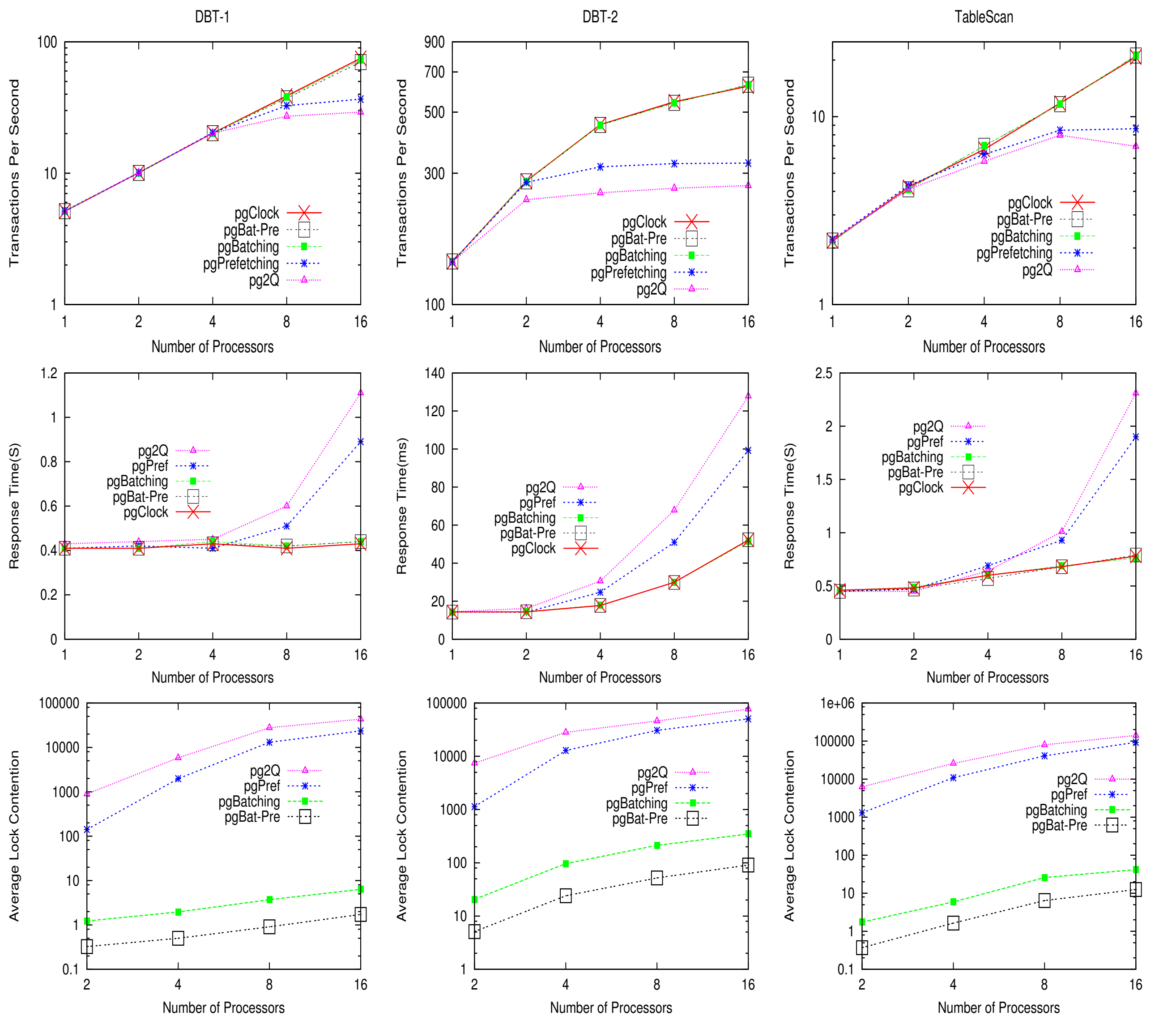

Fig. 6. Throughput, average response time, and average lock contention of five postgreSQL systems (*pgClock*, *pg2Q*, *pgPref*, *pgBatching*, and *pgBat-Pre*) with workloads *DBT-1*, *DBT-2*, and *TableScan* on SGI Altix 350 when the number of processors increases from 1 to 16. Note that the Y axes of the plots for throughput and average lock contention are in logarithmic scale. We do not show the average lock contention when only one processor is used because the values are too small to fit in the graphs.

core processors, such as Sun Niagara 2 processors and Azul vega processors, than it is on the PowerEdge 1900. These processors have many in-order computing cores to support more concurrent threads in a single chip. Thus cache misses usually have larger performance impact with these processors than they do with processors having a few out-of-order computing cores, such as Xeon 5355 processors.

The scalability of system *pgPref* is as poor as that of *pg2Q*, because prefetching cannot reduce lock contention sufficiently, especially when more than four processors are used, as shown in the plots of average lock contention. For example, when two processors are used on the Altix 350 server, it reduces average contention by 82.4% over *pg2Q* on average for the three workloads. When additional processors are used, it reduces the contention by smaller percentages, 60.2% for 4 processors, 46.3% for 8 processors, and 38.6% for 16 processors. As we know, the prefetching technique in *pgPref* reduces lock contention and thus improves system throughput. However, with an increased throughput, the lock is requested more frequently, which offsets the effect of reduced lock contention. This phenomenon becomes even more apparent with a larger number of processors.

In the experiments, system *pgBatching* demonstrates almost the same scalability as that of *pgClock*, the optimal algorithm in scalability. Its throughput curves and average response time curves overlap with those of *pgClock* very well when the number of processors is scaled up. As shown in the figures for average lock contention, system *pgBatching* improves

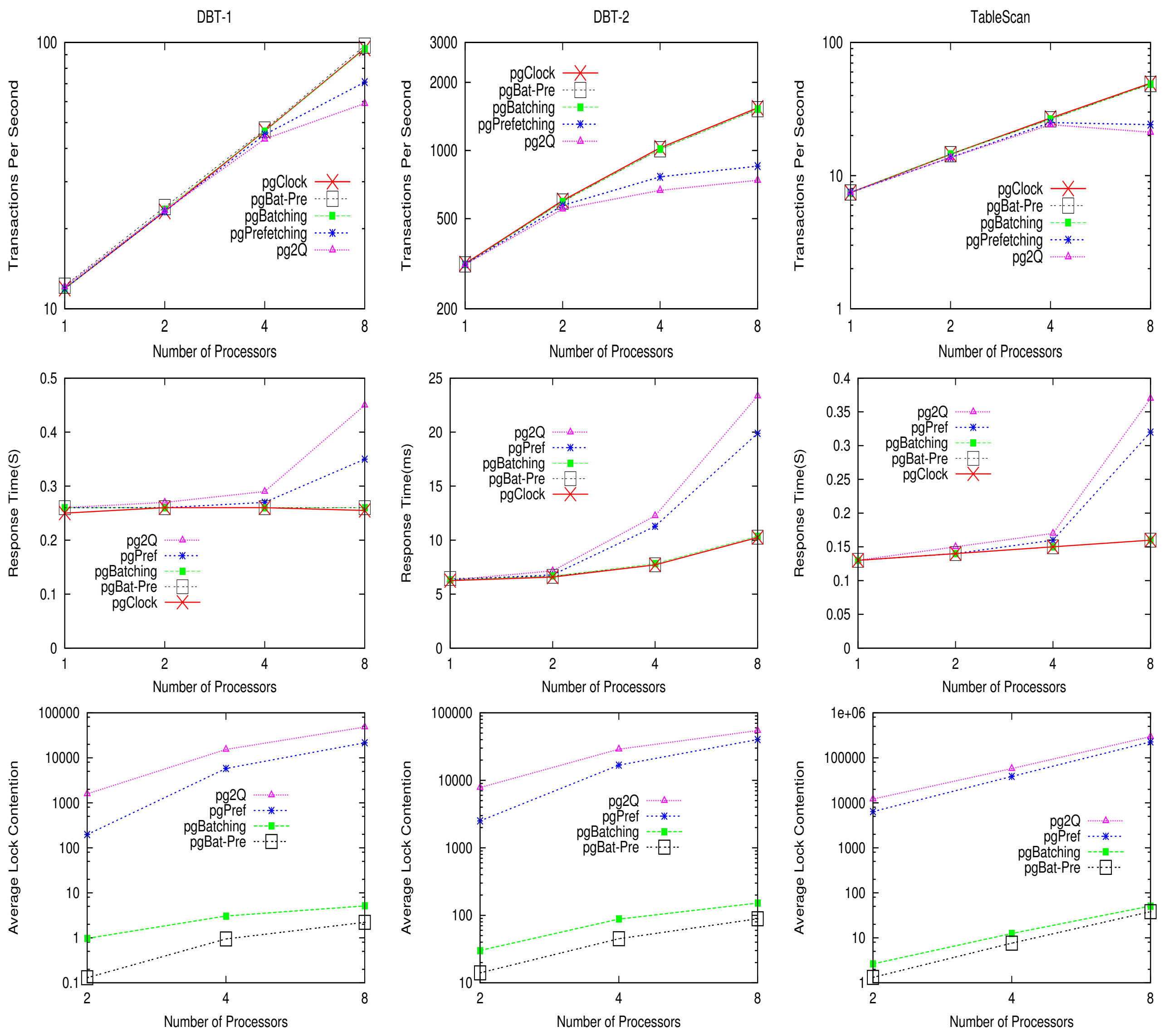

Fig. 7. Throughput, average response time, and average lock contention of five postgreSQL systems (*pgClock*, *pg2Q*, *pgPref*, *pgBatching*, and *pgBat-Pre*) with workloads *DBT-1*, *DBT-2*, and *TableScan* on DELL PowerEdge 1900 when number of processors increases from 1 to 8. Note that the Y axes of the plots for throughput and average lock contention are in logarithmic scale. We do not show the average lock contention when only one processor is used because the values are too small to fit in the graphs.

scalability through reducing lock contention by a factor from 197 to over 9000. We notice that average lock contention in *pgBatching* with 16 processors is even much lower than that of *pgPref* and *pg2Q* with 2 processors.

Using both batching and prefetching techniques, system *pgBat-Pre* can further reduce lock contention compared with *pgBatching*. However, the reduced lock contention is not translated into higher throughput or lower average response time, as shown in the figures, because the average lock contention of *pgBatching* is already very small, and the impact on performance would be diminished with further decrease. When 16 processors are used, both *pgBatching* and *pgBat-Pre* have around or fewer than 400 lock contentions in a million page accesses. When the number of processors keeps increasing, in particular with the prevalence of multi-core processors, the lock contention would be more serious and further reduction of lock contention by *pgBat-Pre* would help improve system performance.

E. Parameter Sensitivity Study

System *pgBatching* uses a small FIFO queue to record access history for each postgreSQL back-end process to amortize the lock acquisition cost. There are two key parameters in the batching technique. One is the size of the FIFO queue, the other is batch threshold, or the minimal number of entries in the queue to trigger a committing. In this section, we perform sensitivity study on the two parameters in the Altix 350 system with 16 processors.

We first gradually increase the queue size from 2 to 64 while keeping the batch threshold as half of the queue size, and run *pgBatching* with the three workloads. Their throughputs and average lock contentions are as shown in Table II. We then keep queue size fixed at 64, and gradually increase the batch threshold from 1 to 64 in the evaluation of *pgBatching* under the three workloads. The throughputs and the average lock contentions are shown in Table III.

TABLE II
THROUGHPUTS AND AVERAGE LOCK CONTENTIONS OF *pgBatching* WITH WORKLOADS DBT-1, DBT-2, AND TABLESCAN WHEN THE QUEUE SIZE INCREASES FROM 2 TO 64 AND THE BATCH THRESHOLD IS 1/2 OF THE QUEUE SIZE.

Queue Size	Throughput			Average Lock Contention		
	DBT-1	DBT-2	TableScan	DBT-1	DBT-2	TableScan
2	46.3	353.4	9.4	34328	44932	79962
4	70.1	609.6	17.4	2675	12539	15001
8	72.0	622.1	19.9	72	1392	616
16	72.7	625.3	20.6	21	616	287
32	72.9	628.6	21.1	11	441	151
64	72.9	629.2	21.2	6	351	126

TABLE III
THROUGPUTS AND AVERAGE LOCK CONTENTION OF *pgBatching* UNDER WORKLOADS DBT-1, DBT-2, AND TABLESCAN WHEN THE BATCH THRESHOLD INCREASES FROM 1 TO 64

Thre-shold	Throughput			Average Lock Contention		
	DBT-1	DBT-2	TableScan	DBT-1	DBT-2	TableScan
1	72.0	616.3	19.8	62	1916	561
2	72.4	622.5	20.3	29	970	370
4	72.6	627.7	20.5	22	526	313
8	72.7	627.9	20.9	16	513	216
16	72.8	628.0	21.1	10	472	165
32	72.9	629.2	21.2	6	351	126
48	72.2	624.2	20.3	31	677	361
64	70.2	611.9	19.4	1230	3569	956

Increasing queue size understandably reduces average lock contention and thus improves throughput because access history can be committed in larger batches. When we increase the queue size from 2 to 8, the average lock contention is decreased by a factor of 477 for DBT-1, 32 for DBT-2, and 130 for TableScan. Accordingly throughput is significantly increased. When the queue size increases beyond 8, increasing queue size can still reduce the average lock contention, but the improvement can hardly be translated into throughput improvement due to already highly reduced lock contention. We also notice that *pgBatching* outperforms *pgPref* even with a very small queue size (2).

Intuitively, choosing a low *batch threshold* would give a postgreSQL back-end process more chance to get the lock without being blocked by trying the lock via TryLock() more times before its FIFO queue becomes full, and thus would decrease the probability of contention. However, the data in Table III shows this trend only when the batch threshold is larger than 32. When we increase the batch threshold from 1 to 32, we find that its average contention decreases and throughput increases. This is because a low batch threshold can

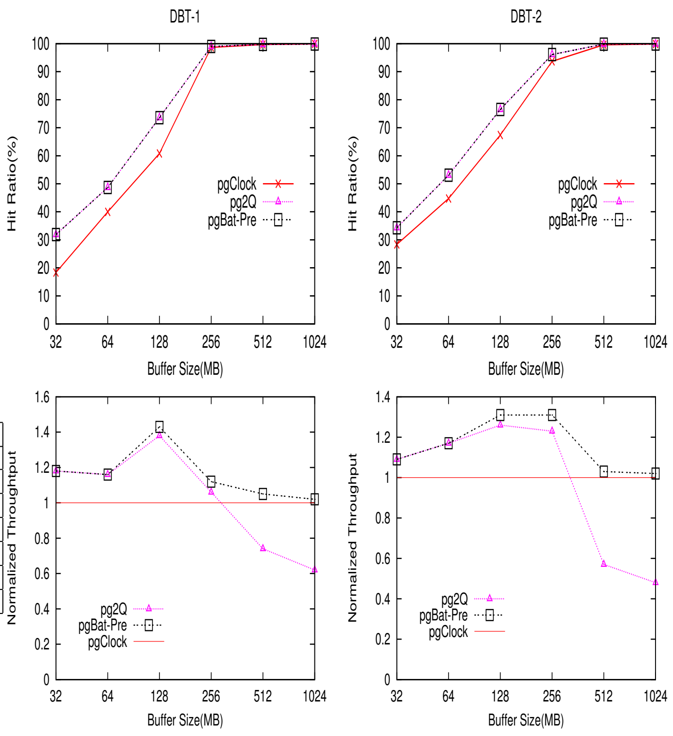

Fig. 8. Hit ratios and normalized throughputs of three postgreSQL systems (pgClock, pg2Q, and pgBat-Pre) with workloads *DBT-1* and *DBT-2* on the PowerEdge 1900 when the number of processors is 8. Throughputs are normalized against those of the pgClock system.

lead to a premature committing, or committing of its access history in small batches. This can decrease the probability that a thread gets the lock via *TryLock()*.

When the batch threshold exceeds 32, each back-end process commits its access history in batches larger than 32 page accesses, which allows effective amortization of the lock acquisition cost. The advantage of *TryLock()*, i.e., acquiring a lock without being blocked, shows up. With relatively smaller threshold, *TryLock()* can be called more times before the FIFO queue becomes full and the chance of the process getting a lock without blocking itself is higher. Therefore, we see that the average contention increases and throughput decreases when the batch threshold increases from 32 to 64.

When the batch threshold is set equal to the queue size (64), the chance of using *TryLock()* is eliminated. Consequently, we see a significant increase in the average lock contention and decrease of throughput, even though a large batch is formed for amortization of the lock acquisition cost. This indicates that a batch threshold sufficiently smaller than the queue size is necessary to take advantage of *TryLock()*.

F. Overall Performance

In this section, we evaluate the overall performance of three systems *pgClock*, *pg2Q*, and *pgBat-Pre* on the PowerEdge 1900 when the number of processors is eight. We have evaluated the scalability of the various systems by setting the buffer size equal to the data sizes of the workloads. The experiments have shown that lock contention can be reduced significantly by combining the batching and prefetching techniques. However, buffer sizes are usually much smaller than data sizes in real systems. Thus, the ability of the systems to reduce costly I/O operations by improving hit ratios is also critical to the overall performance. In this experiment, we change the buffer size from 32MB to 1024MB, and let the systems issue direct I/O requests to bypass the operating system buffer cache. As the data set sizes of *DBT-1* and *DBT-2* are 6.8GB and 5.6GB respectively, not all the accesses can be satisfied from the buffer.

Figure 8 shows the changes of hit-ratio and throughput in the three systems. When memory size is small (less than 256MB), the systems are I/O bound. Systems *pg2Q* and *pgBat-Pre* produce higher throughputs than system *pgClock* by maintaining higher hit ratios. However, as the memory size becomes larger, the overall performance of a system is increasingly determined by its scalability and the advantage of system *pg2Q* in terms of hit ratio has less impact. When the buffer size reaches 1GB, its overall performance drops below that of system *pgClock*. Meanwhile, system *pgBat-Pref* retains its performance advantage with its improved scalability. We also notice that the hit ratio curves of *pg2Q* and *pgBat-Pref* overlap very well. This indicates that our techniques do not hurt hit ratios.

V. Related Work

Research efforts addressing performance degradation due to lock contention have been made actively in the system and database community. In general, lock contention can be reduced by applying the following different approaches.

A. Reducing Lock Granularity Using Distributed Locks

Reducing lock granularity is a commonly used method for decreasing lock contention. Replacing a globally shared lock with multiple fine-grained locks can remove the single point of contention. Oracle Universal Server [7], ADABAS [8], and Mr.LRU replacement algorithm [19] use multiple lists to manage their buffers. Each list is protected by a separate lock. When a new page enters the buffer, Oracle Universal Server inserts the page into the first unlocked list, and ADABAS chooses a list in a round-robin manner. They both allow a page to be evicted from one list and inserted into another list later. Thus many replacement algorithms, such as 2Q [2] and LIRS [1], would not work. To solve the problem, Mr.LRU chooses a list by hashing, which guarantees that a page enters the same list every time it is loaded from the disk.

The distributed lock approaches, including the one used in Mr.LRU, have the following serious drawbacks. (1) They cannot be used to implement replacement algorithms that need to detect access sequences, such as SEQ [12], because pages in the same sequence may be distributed into multiple lists. (2) Though pages can be evenly distributed into multiple lists, accesses to buffer pages may not. Lists with hot pages, such as top-level index pages or pages in a small table for a parallel join, may still suffer from lock contention. (3) To reduce contention, the buffer has to be partitioned into hundreds, even thousands, of lists. Thus each list has a much smaller size than the size of buffer. With the small lists, those pages that need a special protection from eviction, such as dirty pages and index pages, may be evicted prematurely from the buffer. In contrast, our framework is able to implement all replacement algorithms without partitioning the buffer.

B. Reducing Lock Holding Time

As we know, the larger the lock holding time, the more serious the contention would be. Reducing lock holding time is another effective approach to minimizing lock contention.

The TSTE (two stage transaction execution) strategy used in Charm [20] separates disk I/O and lock acquisition into two mutually exclusive stages and ensures that all the data pages that a transaction needs are already in local memory before they are locked. By this method, TSTE reduces lock contention delay in the disk-resident transaction processing system to the same level as that experienced by memory-resident transaction systems.

In Linux kernel 2.4, the scheduler traverses the task structures in a global queue protected by a spin-lock to select a task to run. Paper [21] shows that the contention on the spin-lock can be very serious when the traversal time is lengthened by unnecessary conflict misses in the hardware cache during the traverse. By carefully laying out task structures in memory, most reducing conflict misses can be avoided and lock contention can be greatly reduced because the traversal takes much less time. In contrast, our framework uses prefetch to reduce lock holding time by reducing hardware cache misses when executing the replacement algorithm and by executing a replacement algorithm in a batch mode for multiple accesses.

C. Wait-Free Synchronization and Transactional Memory

As lock synchronization can cause issues like performance degradation and priority inversion, wait-free synchronization [22] addresses these issues by guaranteeing that a transaction completes the operation of accessing the shared resource in a limited number of steps regardless of the execution progress of other transactions. However, programs adopting this technique are difficult to design, and it is possible that an algorithm does not have its wait-free implementation. Moreover, wait-free synchronization requires atomic primitives supported by hardware. For example, the wait-free implementation of a double-ended queue requires Double Compare-And-Swap (DCAS) primitive, which is not available on most processors.

Transactional memory is another approach to address the issue of lock contention. In a transactional memory, a transaction is considered as a series of operations on the shared resources. The atomicity of its execution is guaranteed by either hardware [23] or software [24] that implements transactional memory. It improves system scalability through enabling optimistic concurrency control [25], [26]. While hardware transactional memory has not been available, there are various software transactional memory (STM) implementations. Performance comparisons between STM and lock synchronization show that STM outperforms locks when the shared resources are infrequently changed in the transaction processing [27]. Because data structures in replacement algorithms can be changed frequently (upon each page access), transactional memory can hardly improve the scalability of replacement algorithms. In contrast, both of the batching and the prefetching techniques in BP-Wrapper can be easily implemented in software and do not require special hardware support.

VI. Conclusion and Future Work

In this paper, we address the scalability issue due to lock contention in the implementation of advanced replacement algorithms into DBMS systems. We proposed an efficient and scalable framework, BP-wrapper, in which the batching and prefetching techniques can be used with any replacement algorithms without modification of the algorithms. Without algorithm modification, the performance advantage of the original replacement algorithms will not be compromised, and human effort is also saved. The only cost of the framework is a small FIFO queue for each transaction-processing thread, which keeps the thread's most recent access information.

We have implemented the framework in postgreSQL 8.2.3 and tested it with a TPC-W-like workload, a TPC-C-like workload, and a synthetic workload. Our performance evaluation shows that BP-Wrapper can increase system throughput by nearly two-fold compared to the implementation of an unmodified replacement algorithm, such as LRU and 2Q, and achieve a scalability as good as the one that does not use lock on hit accesses, such as the clock algorithm. We plan to evaluate BP-Wrapper on larger systems, especially the ones with more multi-core processors, in production environments with real-world workloads.

VII. Acknowledgements

This research was supported by National Science Foundation grants CCF-0602152, CCF-0702500, CCF-0702380, and CNS-0834393. The authors thank Dr. Wenguang Wang at Apple Inc. for bringing the lock contention issue to their attention and inspiring their initial investigation on the issue in the database system. They also thank the anonymous reviewers for their constructive comments. They appreciate the comments and suggestions from Dr. Bill Bynum.

References

[1] S. Jiang and X. Zhang, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2002, pp. 31–42.

[2] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1994, pp. 439–450.

[3] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003, pp. 115–130.

[4] T. E. Anderson, "The performance of spin-lock alternatives for shared-memory multiprocessors," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, 1990, pp. 6–16.

[5] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[6] X. Zhang, R. Castañeda, and E. W. Chan, "Spin-lock synchronization on the butterfly and KSR1," *IEEE Parallel Distrib. Technol.*, vol. 2, no. 1, pp. 51–63, 1994.

[7] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, "The oracle universal server buffer manager," in *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds. Morgan Kaufmann, 1997, pp. 590–594.

[8] H. Schöning, "The ADABAS buffer pool manager," in *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1998, pp. 675–679.

[9] F. J. Corbato, "A paging experiment with the multics system," in *In Honor of Philip M. Morse*, Feshbach and Ingard, Eds. Cambridge, Mass: MIT Press, 1969, p. 217.

[10] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *Proceedings of the Annual USENIX Technical Conference '05*, Anaheim, CA, 2005.

[11] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004, pp. 187–200.

[12] G. Glass and P. Cao, "Adaptive page replacement based on memory reference behavior," in *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1997, pp. 115–126.

[13] "DB2 for z/OS: DB2 database design," 2004, URL: http://www.ibm.com/developerworks/db2/library/techarticle/dm-0408whitlark/index.html.

[14] M. Blasgen, J. Gray, M. Mitoma, and T. Price, "The convoy phenomenon," *SIGOPS Oper. Syst. Rev.*, vol. 13, no. 2, pp. 20–25, 1979.

[15] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, 2001, pp. 91–104.

[16] The Open Source Development Laboratory, "OSDL - database test suite," URL: http://old. linux-foundation.org/lab_activities/kernel_testing/osdl_database_test_suite/.

[17] Transaction Processing Performance Council, "TPC-W," URL: http://www.tpc.org/tpcw.

[18] Transaction Processing Performance Council., "TPC-C," URL: http://www.tpc.org/tpcc.

[19] W. Wang, "Storage management for large scale systems," Ph.D. dissertation, Department of Computer Science, University of Saskatchewan, Canada, 2004.

[20] L. Huang and T. cker Chiueh, "Charm: An I/O-driven execution strategy for high-performance transaction processing," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, 2002, pp. 275–288.

[21] S. Yamamura, A. Hirai, M. Sato, M. Yamamoto, A. Naruse, and K. Kumon, "Speeding up kernel scheduler by reducing cache misses," in *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, 2002, pp. 275–285.

[22] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[23] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993, pp. 289–300.

[24] N. Shavit and D. Touitou, "Software transactional memory," in *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, 1995, pp. 204–213.

[25] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.

[26] D. Gawlick and D. Kinkade, "Varieties of concurrency control in IMS/VS fast path," *IEEE Database Eng. Bull.*, vol. 8, no. 2, pp. 3–10, 1985.

[27] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "McRT-STM: a high performance software transactional memory system for a multi-core runtime," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 187–197.