A General Approach to Scalable Buffer Pool Management

Xiaoning Ding, Jianchen Shan, and Song Jiang

Abstract—In high-end data processing systems, such as databases, the execution concurrency level rises continuously since the introduction of multicore processors. This happens both on premises and in the cloud. For these systems, a buffer pool management of high scalability plays an important role on overall system performance. The scalability of buffer pool management is largely determined by its data replacement algorithm, which is a major component in the buffer pool management. It can seriously degrade the scalability if not designed and implemented properly. The root cause is its use of lock-protected data structures that incurs high contention with concurrent accesses. A common practice is to modify the replacement algorithm to reduce the contention on the lock(s), such as approximating the LRU replacement with the CLOCK algorithm or partitioning the data structures and using distributed locks. Unfortunately, the modification usually compromises the algorithm's hit ratio, a major performance goal. It may also involve significant effort on overhauling the original algorithm design and implementation. This paper provides a general solution to improve the scalability of a buffer pool management using any replacement algorithms for the data processing systems on physical on-premises machines and virtual machines in the cloud. Instead of making a difficult trade-off between the high hit ratio of a replacement algorithm and the low lock contention of its approximation, we design a system framework, called *BP-Wrapper*, that eliminates almost all lock contention without requiring any changes to an existing algorithm. In BP-Wrapper, we use a dynamic batching technique and a prefetching technique to reduce lock contention and to retain high hit ratio. The implementation of BP-Wrapper in PostgreSQL adds only about 300 lines of C code. It can increase the throughput by up to two folds compared with the replacement algorithms with lock contention when running TPC-C-like and TPC-W-like workloads.

Index Terms—Buffer pool management, replacement algorithm, lock contention, multi-core

1 Introduction

Data processing systems, such as databases, usually use high-end servers with many cores for high performance. For example, it is common to equip a database server with 16 or more cores. In the cloud, an Amazon RDS database instance can have up to 32 virtual CPUs (VCPUs) [1]. On such a server, a large number of worker threads run concurrently to maximize computing power of the system. With the high computing power, the overall system performance is often determined by how fast the worker threads can access the data they process. As a common practice, a data-processing system maintains a buffer pool in the user memory space for its worker threads to cache the data sets that are actively accessed. The system carefully manages the buffer pool using sophisticated policies to minimize costly disk I/O operations.

Worker threads access the buffer pool concurrently at a very high frequency. With the increasing number of cores (or VCPU count), the buffer pool management must be highly scalable and efficient to effectively cope with the growing processing concurrency. Otherwise, it can become a serious system bottleneck, leading to significant performance degradation. In fact, such performance problems have been widely observed in various systems, such as PostgreSQL [2], MySQL [3], and Oracle [4].

The performance issue is caused by the contention on the locks used in the buffer management, particularly the lock that protects a core data structure used for data replacement algorithm. The algorithm makes decisions on which data pages should be cached in memory to effectively serve requests for on-disk data. To implement the algorithm, the threads maintain a data structure to track their data-access history. They update the data structure in response to page accesses so that replacement decisions can be made based on the history recorded in the data structure. To guarantee the integrity of the data structure, the updates must be made in a serialized fashion. Thus, a lock is required to synchronize the updates. Lock contention happens when the lock is held by one worker thread and other threads must wait in the form of busy-waiting and/or context switches.

Many replacement algorithms and their implementations have been proposed. They organize and manage deep page-access history to maximize hit ratios and minimize the cost incurred by disk I/O operations. These algorithms usually take actions upon each I/O access, either a hit or a miss in the buffer, which mainly include a sequence of updates in the data structure to record the access history. Though the operations are usually designed to be simple and efficient to minimize overhead, in a production system where a large number of threads access on-disk data frequently and concurrently, the high frequency and concurrency may lead to serious lock contention. Performance degradation caused

- *X. Ding and J. Shan are with Computer Science Department, New Jersey Institute of Technology, Newark, NJ 07041. E-mail: {xiaoning.ding, js622}@njit.edu.*
- *S. Jiang is with the Department of Electrical and Computer Engineering, Wayne State University, MI 48202. E-mail: sjiang@wayne.edu.*

Manuscript received 13 Mar. 2015; revised 14 Aug. 2015; accepted 4 Sept. 2015. Date of publication 1 Oct. 2015; date of current version 20 July 2016. Recommended for acceptance by X. Gu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2484321

by lock contention can be significant in large-scale systems. This subject has been a major research issue for years (e.g., [5], [6], [7]). Our experiments show that contention on the lock associated with replacement management may reduce database throughput by a factor of 3 in a 16-processor system.

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, gave up advanced replacement algorithms due to the performance problems caused by lock contention. Instead, they resorted to the clock-based approximations of the LRU replacement [2].

Clock-based approximations, such as CLOCK [8], CLOCK-PRO [9], and CAR [10], are usually more scalable than their corresponding original algorithms (LRU, LIRS, or ARC). 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 the accesses are buffer hits, clock-based approximations set the reference bits or increment the counters, rather than modifying the circular list(s) or the clock hand(s). So they do not need to lock the list(s). Only when there are misses do they need to lock the list(s) and the clock hand(s), and search for victim pages for replacement. Since the lock is less frequently requested than that in the original algorithms, the contention for the lock is reduced.

However, many sophisticated replacement algorithms do not have clock-based approximations. For replacement algorithms that do have clock-based approximations, their clock-based approximations usually cannot achieve high hit ratios as high as they do. By making minimal updates on their data structure, clock-based approximations record very limited history access information, specifically, whether a page has been accessed or how many times it has been accessed. It loses richer history information, especially cross-access information, such as relative order in which a sequence of pages are accessed. This would compromise hit ratio. For some replacement algorithms, such cross-access information is indispensable. Thus, they do not have clock-based approximations. Examples include SEQ [11], DULO [12], and the buffer replacement policy used in DB2 [13]. They need to know in which order the buffer pages are accessed to detect access sequences and sequential/random access patterns.

Lock contention can also be reduced by reducing lock granularity, or using distributed locks [5], [6], [7], [14], [15], with which a buffer is divided into multiple partitions and pages are distributed through hashing. Each partition has its own data structure for tracking history of its accesses and uses a local lock to coordinate concurrent updates to the data structure. Since only accesses to the same partition compete on the same lock, lock contention can be ameliorated.

However, in the distributed lock approach hash collisions and hot pages can still cause contention. Moreover, as the recorded history information is localized within 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.

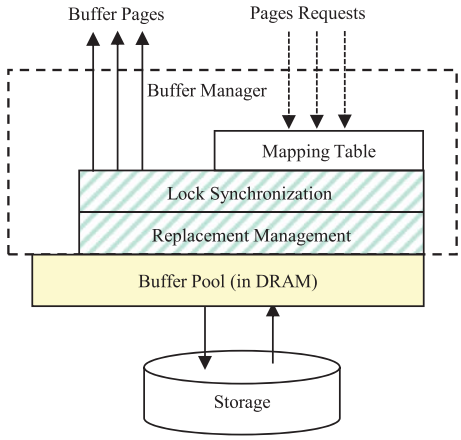

Fig. 1. The diagram of a buffer manager.

In summary, existing efforts on the research and development of replacement algorithms in data processing systems have been focused on addressing the trade-offs between high hit ratios of high-performance algorithms and low-lock-contention implementation in systems. Instead of making a trade-off between these two objectives, our goal is to retain the performance advantages of a replacement algorithm and provide an efficient framework that makes the buffer pool with any replacement algorithm (almost) lock-contention free. With a small FIFO queue maintained for each worker thread, our framework provides two key scalability supports, which can be universally applied to any replacement algorithms. One is *dynamic batch execution*, which reduces number of conflicting lock requests by dynamically batching accesses and amortizes lock contention overhead among a batch of page accesses. The other one is *prefetching*, which reduces the average lock-holding time by pre-loading necessary data managed by the replacement algorithm into the processor cache. We name the framework employing **B**atching and **P**refetching as *BP-Wrapper*. Our prototype implementation of BP-Wrapper in PostgreSQL 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.

2 Background on Buffer Management

In a data processing system, a buffer stores a collection of buffer pages of fixed sizes and is shared by worker threads. Data pages read from hard disks are cached in the buffer for possible reuses. A buffer manager uses some data structures such as linked lists and mapping tables (e.g., hash tables or trees) to organize the metadata of the buffer pages, including identifiers of the cached data pages, status, and pointers to form linked lists and mapping tables.

Fig. 1 shows the diagram of a typical buffer manager. When a thread requests a data page, it first checks whether the page is cached in the buffer. The mapping table is used to speed up the searching. If the page is found (a hit), the operations described by a replacement algorithm are

performed to update the data structures to reflect the page access. For example, with LRU, the page is removed from the LRU list and inserted back to the MRU end of the list. Then the page is returned to complete the request. In another scenario where the requested data page is not in the buffer (a miss), the algorithm selects a victim page and evicts the data in the page to make room for caching the requested data page. When the data page is read into memory, the buffer page is moved to the MRU end of the list and returned to satisfy the request.

The buffer manager is a central component frequently used by all worker threads upon each page request. Simultaneous updates on its data structures have to be carried out in a controlled fashion to maintain its data structure integrity. Usually lock synchronization is used for this purpose.

The use of locks to synchronize mapping table searching and updating does not limit system scalability. The mapping table usually uses distributed locking or hierarchical locking. At the same time, since the mapping table is not changed upon hits, concurrent accesses are allowed upon hits, and exclusive accesses are only required upon misses, which are usually rare, compared to hits. Both factors above help maintaining high scalability. For example, in a hash table, the metadata of buffer pages are distributed into a large number of small hash buckets, each of which is protected by a local read/write lock. Even when multiple threads need to access the same bucket, they can search the bucket concurrently. Only when searches fail (i.e., misses), the exclusive accesses to a bucket are required.

We focus on the lock contention in the replacement management because (1) the replacement management may use one lock for its entire data structure, which is a single point of hot spot, and (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 to exclusively conduct the replacement management operations. The highly contented lock may dramatically degrade system performance on a multicore/multiprocessor system.

3 BP-Wrapper

A lock contention problem can be analyzed with a queuing model, in which requests are serviced in a critical section sequentially. Lock contention happens when there is more than one outstanding request, and the intensity of the contention increases with the number of outstanding requests. Based on the Little's law, the average number of outstanding requests is determined by two factors—the arrival rate of requests and the average request service time. For the lock contention problem in the replacement management, these factors correspond to (1) the frequency in which a lock is requested, and (2) the time to complete the required operations, including lock acquisition, replacement operations, and lock release.

To reduce the two factors contributing to lock contention, 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 to make them highly scalable.

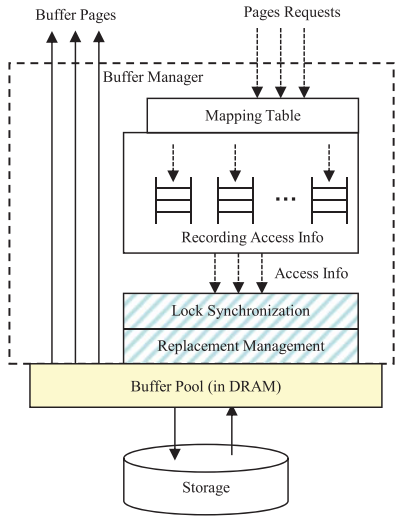

Fig. 2. A buffer manager using the dynamic batching technique.

3.1 Reducing Lock Requests with Dynamic Batching

We use a technique called *dynamic batching* to reduce the frequency of lock requests. The basic idea is to acquire a lock after receiving a set of page accesses. Once a lock is acquired, access history about the set of the accesses is committed to the replacement's data structure together. In this way, the lock acquisition frequency is significantly reduced. The rationale of the technique is that it is unnecessary to carry out the history-recording operations on the data structure immediately after a page access, and we still ensure that all the operations are conducted when a lock is held.

There are two unique properties of a replacement algorithm that allow us to effectively apply the batching technique to significantly reduce the frequency of acquiring the lock. First, postponing the operations on the data structure in replacement algorithms, such as LRU stack or LIRS stacks [16], will not keep the threads from accessing their requested data from the buffer, and thus will not affect the correctness of data processing. Second, in a system with millions of pages (e.g., a server used in our experiments has 64 GB memory, or eight million 8 KB pages), postponement of the operations for recording a few (e.g., 64) recent page accesses into the lock-protected data structure would cause little impact on the performance of the replacement algorithm. Furthermore, the order in which the batched operations are committed on the data structure does not change with the adoption of the dynamic batching technique.

The dynamic batching technique is shown in Fig. 2. A FIFO queue is set for each worker thread. When a thread requests a page, the pointer to the page is recorded in the FIFO queue of the thread. When the number of accesses recorded in the queue reaches a pre-determined threshold, named as *batch threshold*, the thread can start to request a lock to commit the access history in the queue in a batching fashion. After committing the history, the queue is emptied. With the FIFO queue, a thread can access multiple pages without requesting a lock for running the page replacement algorithm.

In the batching technique, an alternative is to use one common FIFO queue shared by multiple threads. However,

```
   /* a FIFO queue of a thread */
1  Page *Queue[S];
   /* current queue position to receive next page */
2  int Tail = 0;
   /* minimal number of pages before a committing */
3  #define batch_threshold T
   /* multiple lists organizing free pages */
4  List Free_Page_Lists[N_FREE_LISTS];

   /* called to commit pages with the lock secured */
5  void commit_pages() {
6    For each page P in Queue[]
7      do what is specified by the replacement algorithm
           upon a hit on P;
8  }

   /* called upon a page hit */
9  void replacement_for_page_hit(Page *this_access) {
10   Queue[Tail] <-- this_access;
11   Tail = Tail + 1;
12   if (Tail >= batch_threshold) {
       /* a non-blocking attempt to acquire lock */
13     trylock_outcome = TryLock();
14     if(trylock_outcome is a failure)
15        if( Tail < S) return; else Lock();
16     commit_pages();
17     UnLock();
18     Tail = 0;
19   }
20 }

   /* called upon a page miss */
21 Page *replacement_for_page_miss(int Block_Index) {
22   List_Index = mod(Block_Index, N_FREE_LIST);
23   List = Free_Page_Lists[List_Index];
24   if List is not empty {
25     remove a free page P from List (atomic operation);
26     replacement_for_page_hit(P);
27     return P;
28   }
29   else {
30     Lock();
31     commit_pages();
32     do what is specified by the replacement algorithm
         upon a miss on block Block_Index;
33     do what is specified by the replacement algorithm
         to reduce buffer size by N_FREE_PAGES;
34     UnLock();
35     List <-- free pages released on line #33;
36     Tail = 0;
37     return the page selected on line #32;
38   }
39 }
```

Fig. 3. Pseudo-code for the replacement-algorithm-independent framework that uses the dynamic batching technique.

we choose to use a private FIFO queue for each thread for two reasons: 1) 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, such as SEQ [11], because they need the ordering information for detection of access patterns. 2) Recording access information into private FIFO queues incurs the least synchronization and coherence cost.

Fig. 3 presents the pseudo-code describing the dynamic batching technique, including lock-related operations upon page hits (replacement_for_page_hit()) and page misses (replacement_for_page_miss()). 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. One condition is that the queue is full. In this case, a lock must be explicitly requested (Lock() on line 15). The other condition is that there are sufficient number of accesses in a queue (not fewer than *batch_threshold*) and the lock is currently not held by any other threads (indicated by the return value of TryLock()). To effectively reduce the frequency of lock requests, *batch_threshold* and the queue size need to be reasonably large.

We use multiple lists (N_FREE_LIST) to organize the free pages in the buffer. When there is a page miss, one of the lists is selected. If the list is not empty, a free page is removed from the list to service the page miss. The removal can be conducted with an atomic (lock-free) operation or with the protection of a local lock. Then, similar to handling a hit, the access is also recorded in the queue (replacement_for_page_hit() on line 26). If the list is empty, in addition to committing the accesses and looking for a victim page for the current page miss, the thread must also get a batch of extra victim pages (F pages as described on line 33) to replenish the free page list. Though these operations must be carried out when the thread is holding the lock, with a reasonably large batch size (e.g., $F = 64$), the frequency of the lock requests for handling misses can be kept very low.

When the committing procedure is activated, 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 dynamic batching technique to provide *any* algorithm with efficient access to the data structures that have to be lock-protected, because the description of the replacement algorithm itself is independent of the dynamic batching technique. No design of an existing replacement algorithm, which may have required significant effort for its development, has to be modified to accommodate the technique to reduce its lock overhead. This is in contrast to the work transforming an algorithm to its CLOCK approximation for reduced lock contention but with a compromised replacement performance.

3.2 Reducing Locking Cost with Dynamic Batching

There are two major types of cost that affects the time used for replacement management upon a buffer access—the locking cost, which is incurred by acquiring and releasing the lock, and the operation cost, which is incurred by updating the data structure for replacement management. Replacement algorithms are usually designed to carry out simple operations upon each page access. Thus, lock operations, especially lock acquisition, may dominate the total cost of replacement management. The cost increases on lock contention when a thread requests a lock which is currently held by another thread, since it must wait for the current lock holder to complete the operations in the critical section and release the lock. The cost also increases when the system scales up.

In the cloud, worker threads run on virtual CPUs in virtual machines (VMs). Lock operations incur higher costs on VMs than they do on physical machines. On VMs, spin locks can dramatically reduce system throughput due to the lock-holder preemption (LHP) problem. The LHP problem is caused because, when a VCPU holding a spin lock is preempted by the virtual machine monitor (VMM), other VCPUs waiting for the lock have to spin for a long time to get the lock. Thus, software usually uses block-based locks

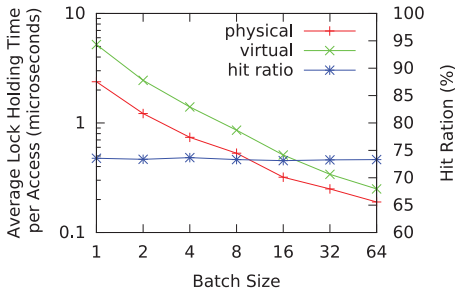

Fig. 4. Average lock holding time (including locking cost) per page access and buffer hit ratio when batch size is varied from 1 to 64 on a physical machine and a virtual machine. The workload is DBT-1 with a 128 MB buffer. The number of processors or VCPUs is 16.

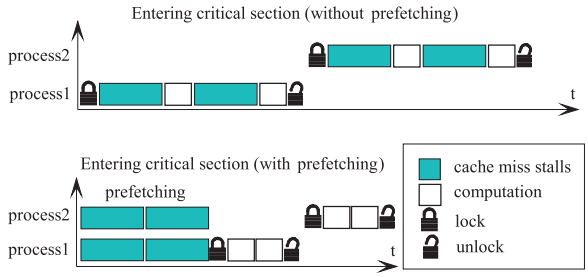

Fig. 5. Using prefetching to move the cache miss penalty out of the lock holding period.

on VMs. Usually, a thread waiting for a lock blocks itself after it has been spinning for a brief time period. When there is contention on a block-based lock, the lock waiter thread and sometimes the corresponding VCPU may be blocked. When the lock is released, they must be woken up and scheduled before the thread can request the lock again. This significantly increases the lock acquisition cost [17].

With the dynamic batching technique, a lock is requested only when a thread has accumulated more accesses than the batch threshold. Thus, the cost of acquiring a lock and releasing the lock is effectively amortized. Meanwhile, the dynamic batching technique uses TryLock() calls to reduce the overhead incurred by spinning and/or blocking a thread when the thread waits for a lock. TryLock() makes an attempt to get a lock. If the lock is currently not held by other threads, the caller thread of TryLock() gets the lock. Otherwise, TryLock() fails without blocking the caller thread, which can proceed and process the data it just reads from the buffer pool. Though TryLock() helps reducing the locking cost, we do not use it for every page access to keep it from producing too many lock acquisition attempts and reducing the chance for a TryLock() to succeed.

To illustrate the effectiveness of the dynamic batching technique on reducing locking cost, we have conducted an experiment on a 16-core physical machine and a 16-VCPU virtual machine (detailed configuration is described in Section 5) to measure the duration in which the lock is requested and held by a thread (i.e., lock-holding time) for processing a certain number (batch size) of page accesses for the 2Q replacement algorithm [18]. We varied the batch size from 1 to 64. Fig. 4 shows the lock holding time, which includes the locking cost, averaged over the batch size. It is evident that the time is much larger with smaller batch sizes. Though the execution time in the critical section for each access can hardly be reduced, the measurements clearly show the effectiveness of the batching technique on reducing locking cost. The experiment also shows that a small number of batch size, such as 64, is sufficient to significantly reduce the locking cost, and the hit ratio does not change with a such small batch size.

3.3 Reducing Operation Cost with Prefetching

When multiple accesses are committed together, the operation cost (i.e., the time spent on examining and updating replacement management data structures) also accumulates and increases proportionally. We use a prefetching technique to reduce the cost. With the technique, we read the data that would be accessed in the critical section by the replacement management immediately before a lock is requested. Taking the LRU algorithm as an example, we pre-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 other fields of the lock data structure. A side-effect of the read is that the data are loaded into the processor's cache and the cache misses otherwise experienced by the critical section code can be removed. The potential benefit of the technique is illustrated in Fig. 5.

The prefetching (read) operation on the shared data without a lock does not compromise the integrity of the shared data structure used in the replacement algorithm. It only loads the data into processor cache, and does not modify any data. The prefetched data will be re-read in the critical section. If the data have been changed by other threads before they are used by the thread that prefetches them, hardware mechanism built in processors will automatically invalidate the data from the cache or update the data with the latest values to keep data coherent; thus re-reading will return the correct values of the data.

4 Combining BP-Wrapper and Other Techniques to Further Reduce Lock Contention

With a conventional design of a replacement algorithm, the operations on its data structures are carried out inside a critical section protected by a lock. BP-Wrapper does not change the operations or the data structures. It only changes when and how these operations are carried out. Thus, BP-Wrapper can be combined with other techniques that involve the changes to the data structures and/or operations of replacement algorithms, in order to further reduce lock contention. In this section, we discuss the benefits and methods to combine BP-Wrapper with other contention-reducing techniques. We use two techniques as examples. One technique uses distributed locks and the other uses lock-free operations to handle accesses to hot pages.

The distributed lock approach divides a buffer pool into multiple partitions, each of which has its own data structure and is protected by a local lock. Since each partition can be considered as a buffer pool with a small size, a natural method to combine the distributed lock approach and BP-Wrapper is to apply BP-Wrapper techniques to each partition in the same way as applying it to a buffer pool, as shown in Fig. 6. The combination helps reducing the cost

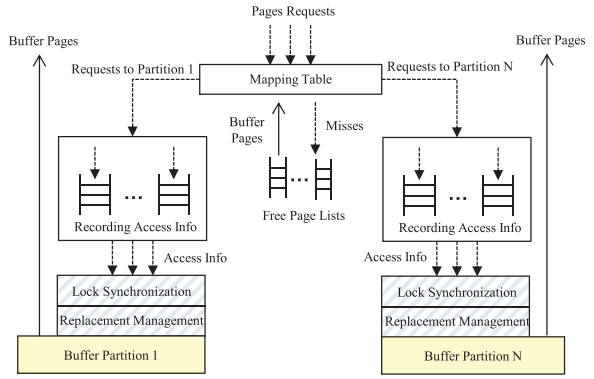

Fig. 6. The combination of BP-Wrapper and the distributed-lock approach.

```
1  Page *Queue[S];
2  int Tail = 0, Access_Count = 0;
3  bool bpwrapper_enabled = false;
4  #define batch_threshold T

5  void replacement_for_page_hit(Page *this_access) {
6    if bpwrapper_enabled
7        Access_Count = Access_Count + 1;
8    if this_access is in hot region
9        lock-free operations defined in the original
               replacement algorithm;
10   else {
11       bpwrapper_enabled = true;
12       Queue[Tail] <-- this_access;
13       Tail = Tail + 1;
14   }
15   if (Access_Count >= batch_threshold) {
16     pre-read data to be used in the critical section
17     trylock_outcome = TryLock();
18     if(trylock_outcome is a failure)
19        if( Access_Count < S) return; else Lock();
20     commit_pages();
21     UnLock();
22     Tail = 0; Access_Count = 0;
23     bpwrapper_enabled = false;
24   }
25 }
```

Fig. 7. Pseudo-code illustrating the combination of BP-Wrapper and the approach of using lock-free operations for hot pages.

incurred by lock operations and the lock contention in hot partitions. To use BP-Wrapper in a partition, a queue is created in each worker thread to accumulate the accesses to the partition. When the number of accesses accumulated in the queue exceeds the batch threshold, the thread will try to acquire the lock and commit the accesses. Depending on whether the distributed-lock implementation of a replacement algorithm selects victim pages locally or globally on misses, the lists for free pages can be created dedicatedly for each partition (when victim pages are selected locally) or shared by all the partitions (when victim pages are selected globally).

To reduce lock contention, another approach is to manage hot pages (frequently accessed) and cold pages (infrequently accessed) separately and handle hot pages with lock-free operations. Since usually most accesses are to hot pages and there is no need to request locks to handle them, lock contention can be effectively reduced. For example, the Oracle Universal Server divides a buffer into a hot region and a cold region, and maintains the hot region as a FIFO queue and the cold region as a conventional LRU queue [19]. Accesses to the pages in the hot region do not move the page and thus do not need to acquire the lock. Accesses to the pages in the cold region will move the pages into the hot region while holding the lock. When the length of the hot region exceeds a threshold, the page at the tail of the FIFO queue are moved to the cold region when holding the lock.

The effectiveness of the above approach is determined by the access patterns of the workloads. It is effective when the accesses have strong temporal locality and the hot region is large enough to host hot pages. When the accesses do not have strong temporal locality (e.g., random accesses) or when hot pages cannot be completely held in the hot region, a substantial portion of accesses will hit the cold region. For example, when looping through a table larger than the hot region, all the accesses will hit the cold region. When the accesses to the cold region cannot be effectively reduced, severe lock contention still may be incurred by the lock requests of these accesses.

The combination with BP-Wrapper addresses the above problems. Fig. 7 shows the pseudo-code for handling page hits (the pseudo-code for handling page misses is similar to that in Fig. 3). In the combined approach, BP-Wrapper is utilized to reduce the lock contention incurred by buffer misses and the accesses to cold pages. It is enabled when there is a miss or an access to the cold region (line 11). Then it starts to accumulate accesses. Since the accesses to the hot region does not change the data structure, they are not accumulated in the queue, and are handled in the same way as that with the original replacement algorithm (line 9). BP-Wrapper counts the number of accesses since last time it commits the accesses. If the number of accesses exceeds *batch_threshold*, it tries to get the lock and commit the accumulated accesses. After committing the accesses, BP-Wrapper is disabled (line 22). Thus, in a worker thread, if most accesses hit the hot region, BP-Wrapper is inactive. When more accesses hit the cold region, BP-Wrapper becomes more active. If all the accesses hit the cold region, BP-Wrapper behaves exactly the same as that in Fig. 3.

BP-Wrapper can also be combined with clock-based approximations, which show high scalability on page hits, but may suffer lock contention on misses. In the combination, BP-Wrapper can be utilized for the operations processing misses in a way similar to that shown in Fig. 3.

5 Performance Evaluation

We have implemented the proposed BP-Wrapper framework on the postgreSQL database system version 8.2.3. PostgreSQL used LRU and 2Q replacement algorithms in its previous versions. Since version 8.1, postgreSQL adopted the CLOCK replacement algorithm in order to improve the scalability of its buffer management on multi-processor/multicore systems. The CLOCK replacement algorithm does not need a lock upon hit accesses. In this sense, it reduces lock contention and provides high scalability. In the experiments, we will compare the performance of BP-Wrapper against the clock-based replacement management implemented in postgreSQL.

We have also implemented an emulator, in which multiple threads periodically access a shared buffer managed by LRU replacement algorithm. In the emulator, we implemented BP-Wrapper framework and the distributed-lock method to coordinate concurrent accesses. In the emulation, we can conveniently control the frequency in which buffer

pages are accessed by the threads. With the emulator, we want to compare the performance of BP-Wrapper against the distributed-lock approach under different workloads. We also show the performance advantages obtained by combining BP-Wrapper with the distributed-lock approach and by handling misses in a scalable way. We ran the emulator on a physical machine and in a virtualized environment.

5.1 Evaluation with PostgreSQL

Our evaluation with postgreSQL has two parts. In the first part, we focus only on the scalability issue, and show that, using BP-Wrapper a high-performance 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 entire working set of a benchmark and pre-warm the buffer. Thus there are no misses incurred no matter which replacement algorithm is used. There are two reasons for us to choose this experiment setting. One is that 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. The other reason is that the CLOCK implementation has optimal scalability when there are no misses and should show the best performance when the system is scaled up. Thus we can test the scalability of an implementation by measuring how close its performance is to that of the CLOCK implementation.

In the second part of our evaluation, we vary the buffer size and show the overall performance of a buffer with improved hit ratio and scalability. There exist a variety of replacement algorithms that can provide performance much better than the CLOCK algorithm in terms of hit ratio, but whose performance does not scale with access concurrency. While LRU can be transformed into the CLOCK algorithm, many of other algorithms are very hard, if not impossible, to be effectively transformed and thus are not appropriate choices in an environment of high concurrency. If no actions are taken to reduce lock contention, the performance advantages of such a replacement algorithm can be compromised in a large-scale system. Our techniques help it retain its performance advantage by improving its scalability.

5.1.1 Tested Systems and Implementation Issues

We first modified postgreSQL by replacing its CLOCK algorithm with the 2Q algorithm, as a representative of the replacement algorithms with high hit ratios [18]. This modified system, which was not optimized for 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 dynamic batching technique and the prefetching technique separately, and named the systems as *pgBatching* and *pgPref*, respectively. We also enabled both techniques together, and name the system as *pgBat-Pre*. The stock postgreSQL is denoted as *pgClock*. These tested systems are summarized in Table 1. We also implemented systems by replacing the 2Q algorithm of the last four systems in the table with the LIRS [16] and MQ [20] replacement algorithms, respectively. We did not observe significant performance differences between the experiments with these algorithms and those with their 2Q counterparts, so we do not show their results.

TABLE 1
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	Dynamic Batching
pgPref	2Q	Prefetching
pgBat-Pre	2Q	Dynamic Batching and Prefetching

The implementation of batching and prefetching requires only limited modification of the baseline system. We added 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.

5.1.2 Experiment Setup and Workloads

We carried out our experiments on both a traditional unicore SMP platform and a multi-core platform. One platform is an SGI Altix 350 SMP server with 16 1.4 GHz Intel Itanium 2 processors (16 cores in total) and 64 GB memory. The storage is a 2 TB LUN on an IBM FAStT600 turbo storage subsystem. The LUN consists of 9 250 GB SATA disks in an 8+P RAID5 configuration. Operating system is Red Hat Enterprise Linux AS release 3 with SGI ProPack 3 SP6. The other platform is a DELL PowerEdge 1900 server, which has two 2.66 GHz quad-core Xeon X5355 processors (eight cores in total). The memory size is 16 GB. 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 *DBT-1* and *DBT-2* from OSDL database test suite [21], 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 [22]. The database generated for the experiments includes information on 100,000 items and 2.9 million customers. *DBT-2*

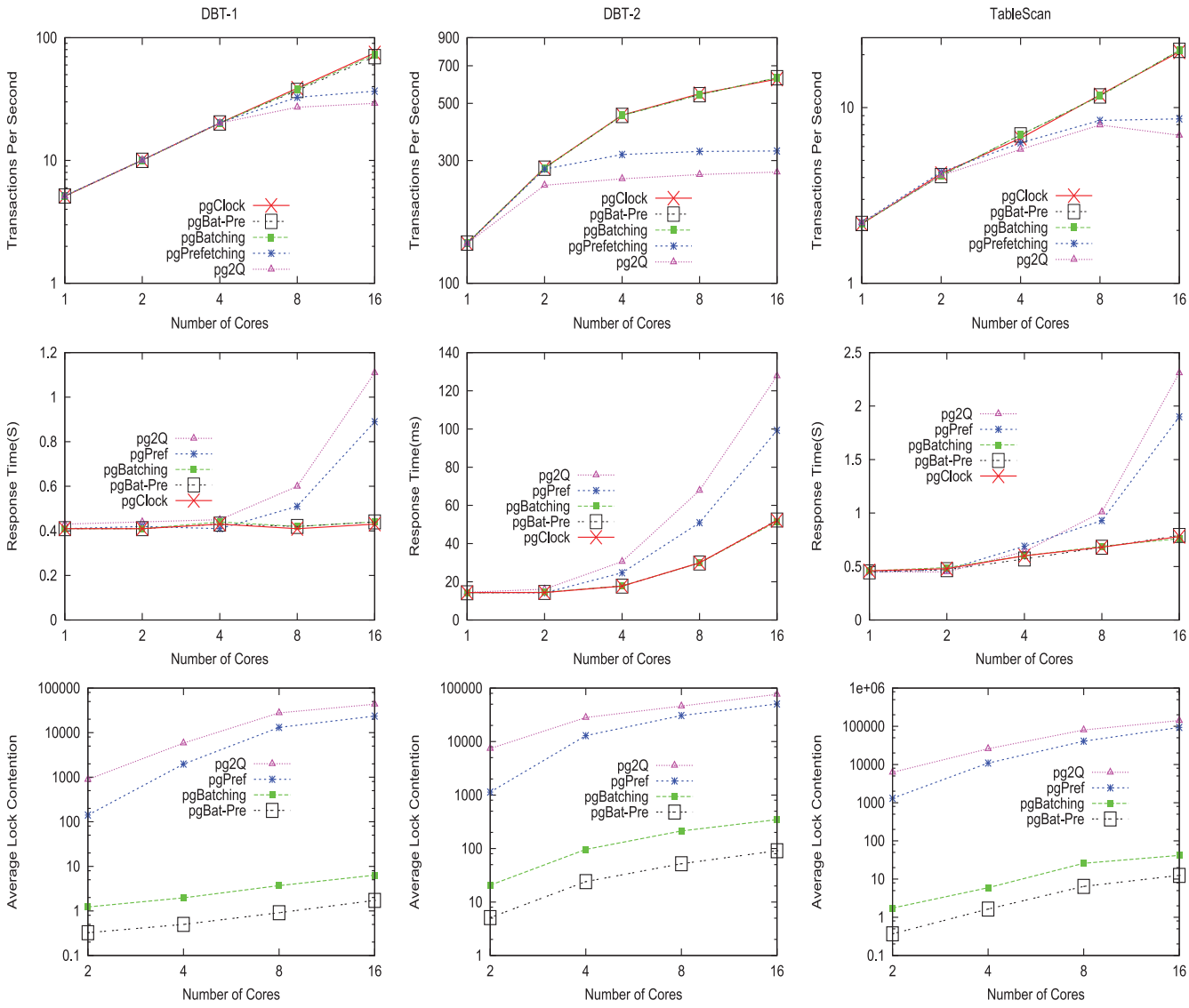

Fig. 8. Throughput, average response time, and average lock contention of five postgreSQL systems with workloads *DBT-1*, *DBT-2*, and *TableScan* on SGI Altix 350 when the number of cores increases from 1 to 16. We do not show the average lock contention for pgClock or when one core is used because the values are too small to fit in the graphs.

derives from the TPC-C specification version 5.0 [23] 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.

5.1.3 Experiments on Scalability

In the experiments, we evaluate the scalability of the five different postgreSQL systems under the three workloads. We increase the numbers of cores used by postgreSQL from 1 to 16 on the Altix 350 server and from 1 to 8 on the PowerEdge 1900 server 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 core when it fails to get a lock due to contention, we make the system over-committed and the cores always busy by keeping more active postgreSQL back-end processes than the number of cores 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*. The number of free page lists is 64 and pages are reclaimed in batches of size 64.

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 throughput (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 throughput, average response time, and average lock contention for the three workloads on the different systems in Figs. 8 and 9 .

When the number of cores is increased, as we expect, the throughput of *pgClock* increases almost linearly with it, and its average response time increases moderately with

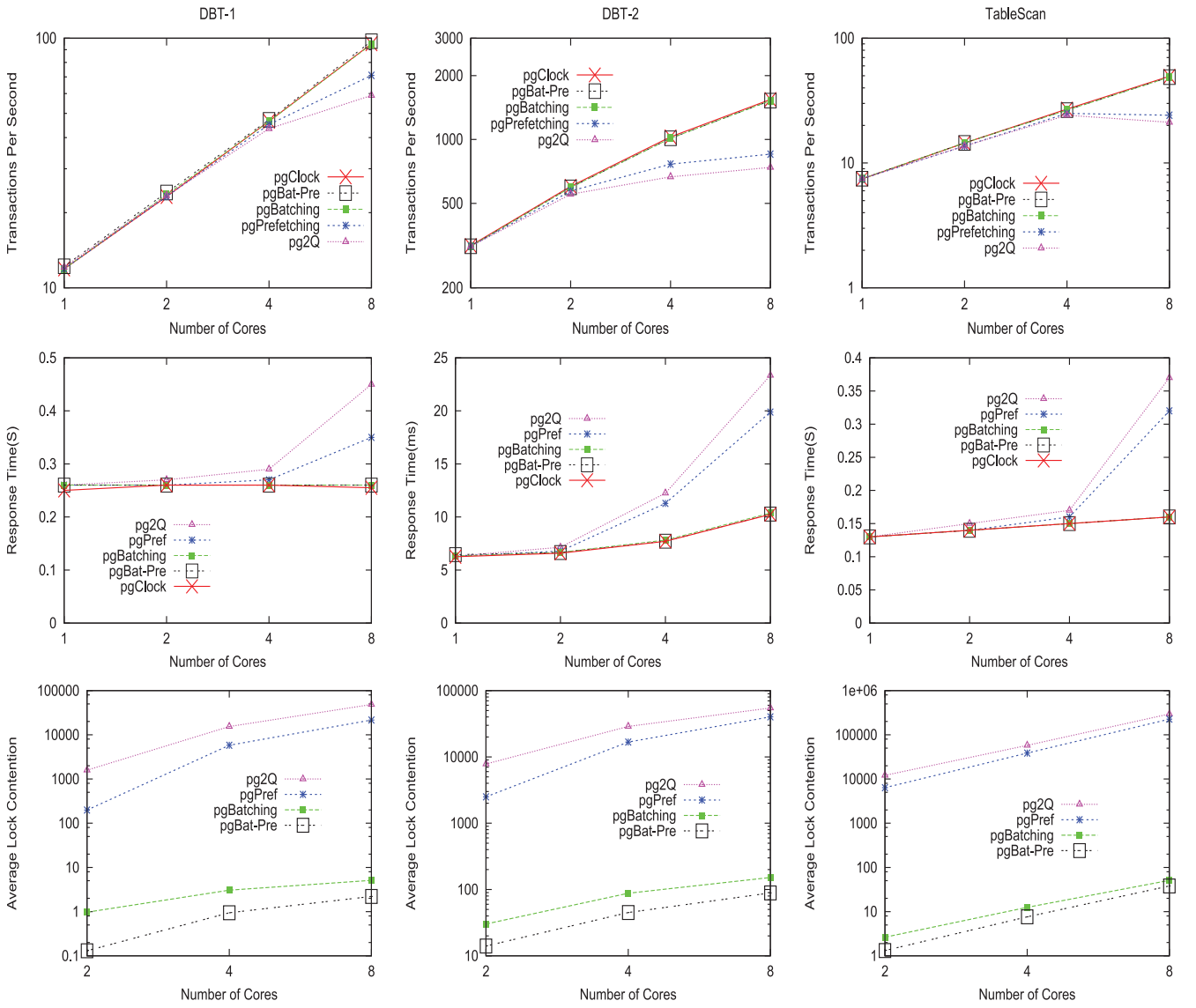

Fig. 9. Throughput, average response time, and average lock contention of five postgreSQL systems with workloads *DBT-1*, *DBT-2*, and *TableScan* on DELL PowerEdge 1900 when number of cores increases from 1 to 8. We do not show the average lock contention for pgClock or when only one core is used because the values are too small to fit in the graphs.

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 cores.

System *pg2Q* can maintain its scalability only when the number of cores is less than 4. On the Altix 350, its throughput saturates when the number of cores is greater than 8 for *DBT-1* and *TableScan*, and 4 for *DBT-2*, respectively, and the average response time increases significantly when additional cores are added. For workload *TableScan*, its throughput even drops by 12.7 percent when the number of cores is increased from 8 to 16. When 16 cores are used, its throughputs are 61.1, 56.5, and 66.5 percent 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 cores (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 cores reaches 4. The average lock contention numbers indicate that lock contention is more intensive on PowerEdge 1900 than that on the Altix 350, especially with benchmark *TableScan*. When eight cores are used, the average lock contentions of *pg2Q* on the PowerEdge 1900 are 74.4, 18.5, and 270.2 percent more than those on the Altix 350 for the three workloads, respectively. Thus, lock contention causes more performance degradation on the PowerEdge 1900 than on the Altix 350. With 8 cores, the throughputs of system *pg2Q* are 37.9, 52.1, and 57.2 percent less than those of pgClock on the PowerEdge 1900 system, while the throughputs of system *pg2Q* are 30.1, 51.5, and 32.6 percent 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.

Lock contention on the PowerEdge 1900 server is more intensive because 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, is accelerated by the prefetching modules, while the operations of replacement management protected by the lock can hardly be accelerated by the prefetching modules because they usually access memory randomly. Therefore, a larger proportion of time is spent on the critical section on the PowerEdge 1900 server than that on the Altix 350. This makes lock contention more intensive on the PowerEdge 1900 server.

Compared with *pg2Q*, *pgPref* reduces the average lock contention by 33.7 to 82.6 percent on the Altix 350 server and by 20.8 to 87.5 percent on the PowerEdge 1900 for the workloads. 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 percent and the average response times of *pgPref* are smaller than those of *pg2Q* by up to 25.2 percent. 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 larger-scale systems with more cores.

The scalability of system *pgPref* is as poor as that of *pg2Q*, because prefetching individually cannot reduce lock contention sufficiently, especially when more than four cores are used, as shown in the plots of average lock contention. For example, when two cores are used on the Altix 350 server, it reduces average contention by 82.4 percent over *pg2Q* on average for the three workloads. When more cores are used, it reduces the contention by smaller percentages, 60.2 percent for four cores, 46.3 percent for eight cores, and 38.6 percent for 16 cores. 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 cores.

System *pgBatching* demonstrates almost the same scalability as that of *pgClock*, the optimal algorithm in scalability when there are not any misses. Its throughput curves and average response time curves overlap with those of *pgClock* very well when the number of cores is scaled up. As shown in the figures for average lock contention, system *pgBatching* improves scalability through reducing lock contention by a factor from 197 to over 9,000. We notice that average lock contention in *pgBatching* with 16 cores is even much lower than that of *pgPref* and *pg2Q* with two cores.

Using both dynamic 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 diminish with the further decrease of lock contention. When 16 cores are used, both *pgBatching* and *pgBat-Pre* have around or fewer than 400 lock contentions in a million page accesses. When the number of cores keeps increasing, the lock contention would be more serious and further reduction of lock contention by *pgBat-Pre* would help improve system performance.

Under the above settings, we have also performed sensitivity study on two key parameters—the size of the FIFO queue and the batch threshold. The results can be found in [24].

5.1.4 Overall Performance

We have evaluated the scalability of the systems by setting the buffer size equal to the data sizes of the workloads, and 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 section, we evaluate the overall performance of three systems *pgClock*, *pg2Q*, and *pgBat-Pre* on the PowerEdge 1900 using eight cores when we change the buffer size from 32 to 1,024 MB, 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 6.8 and 5.6 GB respectively, not all the accesses can be satisfied from the buffer.

Fig. 10 shows the changes of hit-ratio and throughput in the three systems. When memory size is small (less than 256 MB), the systems are I/O bound. Systems *pg2Q* and *pgBat-Pre* produce higher throughputs than system *pgClock* by maintaining higher hit ratios. The performance advantages of these two systems over *pgClock* increase with the number of cores, as shown on the last row in Fig. 10 when the memory size is 128 MB. This is because, when more cores are used, the space in the buffer pool is more intensively contended and the replacement algorithm plays an increasingly important role to minimize I/O operations. The figure also shows that the throughput with *pg2Q* is slightly lower that with *pgBat-Pre* when the number of cores is increased to 4. This indicates that lock contention may still happen and can degrade system throughput in this scenario, though I/O throughput is the major performance factor.

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 512 MB, 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, when the memory size increases, the hit ratio curves of *pg2Q* and *pgBat-Pref* overlap very well. This indicates that our techniques do not hurt hit ratios.

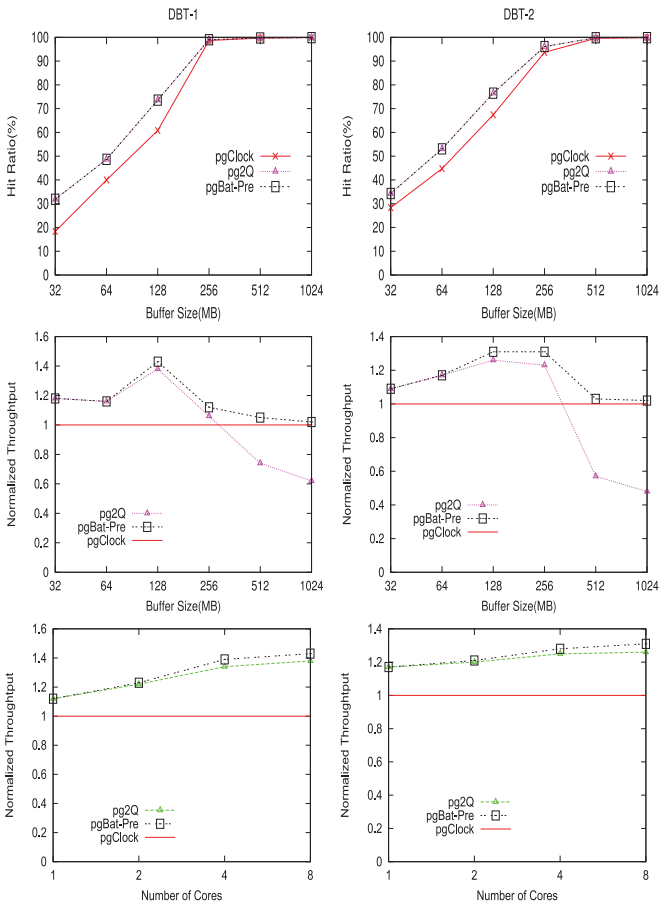

Fig. 10. Hit ratios and normalized throughputs of three postgreSQL systems (pgClock, pg2Q, and pgBat-Pre) with workloads *DBT-1* and *DBT-2* on the PowerEdge 1900 when the number of cores is 8 (the first two rows), and the normalized throughputs of these systems when the number of cores is increased from 1 to 8 and the buffer pool size is 128 MB (the third row). Throughputs are normalized against those of the pgClock system.

5.2 Evaluation with a Buffer Emulator

We have tested the performance of BP-Wrapper under database workloads and compared it against the replacement management with a clock-based approximation. In this section, we compare it against another major approach to achieving scalable buffer management—distributed locks. We have implemented an emulator of buffer management, in which 16 threads running on 16 cores share a buffer pool. In a loop, each thread repeatedly accesses the buffer pool for a data page, performs necessary replacement management operations, and spends a period of random length on computation. We choose emulation because we want to control the intensity of lock contention by adjusting the average length of computation between consecutive buffer accesses. The data pages accessed by the thread are randomly chosen. The number of accesses to the data pages follows a Zipfian distribution. As we want to conduct some of the experiments on a virtual machine, we use mutex for lock synchronization in the emulator.

The buffer pool is managed with the LRU replacement algorithm. We choose the LRU replacement algorithm because it is widely used and the distributed-lock approach can be used to reduce its lock contention while maintaining similar hit ratio. With the distributed lock approach, the

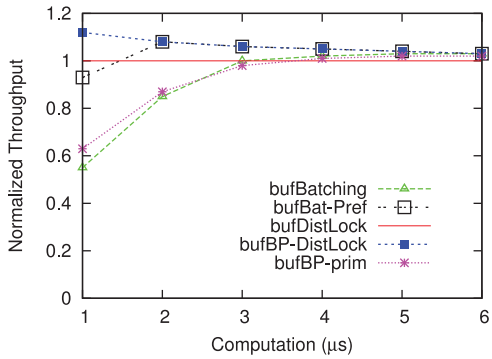

Fig. 11. Normalized throughputs of *bufBatching*, *bufBat-Pref*, *bufBP-DistLock*, and *bufBP-prim* (relative to those of *bufDistLock*) when the average computation time is varied from 1 to 6μs.

buffer pool is divided into 64 partitions. (On our system, dividing the buffer pool into more partitions cannot further improve performance.) The buffer pages in each partition is organized into an LRU list and is protected by a local lock. For brevity, we name the buffer pool under the original LRU replacement management without any techniques to reduce lock contention *bufLRU*, and name the buffer pool using the distributed-lock approach *bufDistLock*. When the dynamic batching technique is used, the buffer pool is named *bufBatching*. With fully-fledged BP-Wrapper framework, the buffer pool is named *bufBat-Pref*.

We emulate a buffer pool with one million buffer pages. The number of data pages is ten million, and the hit ratio is roughly 90 percent. Similar to the research in [14], we perform the experiments using a RAM disk to hold the data pages. Thus, the performance is not affected by I/O overhead.

We use system throughput as the performance metric, which measures the total number of accesses to the buffer pool per second. We collect the throughput of *bufDistLock*, *bufBatching*, and *bufBat-Pref* when we vary the average length of computation from 1 to 6 μs to adjust the intensity of lock contention. Considering a memory access may take about 100 ns, 1 μs is a very short time period for the computation between consecutive buffer page accesses. It incurs intensive lock contention such that the system throughput is significantly reduced—based on our experiment, with a 1μs average computation time, the throughput of *bufLRU* is 15x lower than that of *bufDistLock* or *bufBat-Pref*. Since the system throughputs change dramatically when the average length of computation time is varied from 1 to 6 μs, to clearly show the performance difference, we normalize the throughputs of *bufBatching* and *bufBat-Pref* against that of *bufDistLock* with the same average length of computation time. We show the normalized throughputs in Fig. 11.

Generally, BP-Wrapper performs similarly or slightly better than the distributed-lock approach, except when the average computation time is very short. BP-Wrapper has better performance because the dynamic batching technique significantly reduces the number of locking operations, including lock acquisitions and releases, which are costly. With BP-Wrapper, a thread acquires a lock for a batch of accesses. With the distributed-lock approach, however, a thread has to acquire a lock for each page access. The figure also shows that, in BP-Wrapper, the prefetching technique

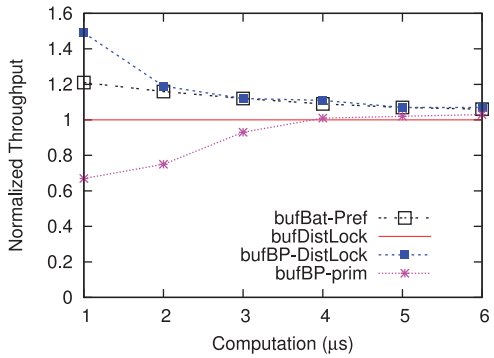

Fig. 12. Normalized throughputs of *bufBat-Pref*, *bufBP-DistLock*, and *bufBP-prim* (relative to those of *bufDistLock*) on a virtualized platform when the average computation time is varied from 1 to 6μs.

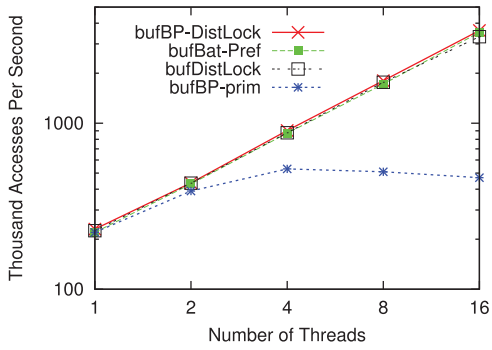

Fig. 13. Throughputs of *bufBP-Distlock*, *bufBat-Pref*, *bufDistLock*, and *bufBP-prim* when they handle misses.

can effectively reduce lock contention and increase throughput when lock contention is intensive and the dynamic batching technique alone cannot reduce the lock contention to a desirable level. For example, when the average length of computation is 1μs, the throughput of *bufBat-Pref* is almost 2x as high as that of *bufBatching*. When the average computation time is 1μs, the throughput of *bufBat-Pref* is 7 percent lower than that of *bufDistLock*. This is because, with the distributed-lock approach, the threads can update the data structures for different buffer pool partitions in parallel.

Fig. 11 also shows the potential to combine BP-Wrapper and the distributed-lock approach by using the BP-Wrapper framework in each buffer pool partition. When the average length of computation is 1μs, the combination (*bufBP-DistLock*) can increase system throughput by 12 percent over that achieved by the distributed-lock approach (i.e., *bufDistLock*).

A preliminary version of BP-Wrapper [24] needs to acquire a lock on each buffer miss to reclaim a page, and thus has low scalability when handling misses. On a system with hard disk storage, since the system is I/O-bound even when the hit ratio is high (e.g., 80 percent), the scalability problem with handling misses may not be evident. However, since storage class memory (e.g., SSDs) may gradually replace hard disks in data processing systems, misses will become less costly on future systems, and the scalability of handling misses will become an increasingly important performance factor. To illustrate this, Fig. 11 also shows the performance of the preliminary version of BP-Wrapper (denoted by *bufBP-prim*). With *bufBP-prim*, the poor scalability with handling misses can significant reduce system throughput (e.g., 29 percent, relative to that of *bufBat-Pref*, when the average length of computation is 1μs). With longer computation time, misses are less frequent. Though the speed gap between *bufBP-prim* and *bufBat-Pref* reduces, the throughput with *bufBP-prim* is still lower than that with *bufBat-Pref*.

To test the performance of BP-Wrapper in the cloud, we perform the emulation on a virtual machine with 16 virtual CPUs. The virtual machine is co-located with another virtual machine on the same physical machine with 16 physical cores. We run streamcluster in the second virtual machine, which is a benchmark from PARSEC suite [25]. Fig. 12 shows the throughput of *bufBP-prim*, *bufBat-Pref*, and *bufBP-DistLock* when we vary the average computation time from 1μs to 6μs. The throughput is normalized to that of *bufDistLock*. As shown in the figure, BP-Wrapper demonstrates larger performance advantage on the virtual machine than it does on a physical machine. When the average computation time is $1\mu s$, the throughput of *bufBat-Pref* is 21 percent higher than that of *bufDistLock* on the VM. This is because BP-Wrapper significantly reduces the frequency in which lock is requested and lock acquisition is more costly on virtual machines than it does on physical machines. The performance advantage diminishes gradually when average computation time increases. This is because, with longer computation periods, lock contention become less intensive and there is less opportunity for BP-Wrapper to improve performance. The similar trend is also observed for *bufBP-DistBlock*. Due to the high cost of lock acquisitions on virtual machines, the performance gap between *bufBP-prim* and *bufBat-Pref* or *bufBP-DistLock* is wider than that on the physical machine.

The experiments above test the overall performance when handling both hits and misses. To test the scalability of BP-Wrapper when it handles misses, we have modified the threads in the emulator such that they always access new pages. Thus, all the accesses are cold misses, no matter what the replacement algorithm is. The average computation time is 3μs. We vary the number of threads from 1 to 16 and collect the throughputs (numbers of accesses per second), which are as shown in Fig. 13. The performance of *bufBP-prim* saturates with four threads and then declines with more threads. The other three systems show scalable performance. The system *bufDistLock* is scalable since pages can be reclaimed from different partitions in parallel. Systems *bufBat-Pref* and *bufBP-DistLock* show slightly better performance than *bufDistLock* since lock acquisitions are less frequently with BP-Wrapper than with *bufDistLock*.

6 Related Work

Some preliminary results of this work have been presented in [24]. The closest related work includes the adaption of the techniques in BP-Wrapper in multi-level distributed caches [26], and in infinispan key/value data grid [27]. The 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.

6.1 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 [19], ADABAS [28], Mr.LRU replacement algorithm [29], and set-associative page cache [14] 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. So, the approach is not applicable to many replacement algorithms, such as 2Q [18] and LIRS [16]. To address the problem, Mr. LRU and set-associative page cache choose 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 and set-associative page cache, have serious drawbacks. First, they cannot be used to implement replacement algorithms that need to detect access sequences, such as SEQ [11], because pages in the same sequence may be distributed into multiple lists. Second, 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. Third, to reduce contention a buffer has to be partitioned into a large number of lists. Each list has a much smaller size than the buffer size. With small lists, the 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.

6.2 Reducing Lock Holding Time

The longer the lock holding time, the more serious the contention would be. Thus, reducing lock holding time is another effective approach to reduce lock contention.

The TSTE (two stage transaction execution) strategy used in Charm [30] separates disk I/O and lock acquisition into two mutually exclusive stages and ensures that all the data pages a transaction needs are already in local memory before they are locked. In this method, TSTE reduces the delay caused by lock contention 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 [31] reports 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 running the replacement algorithm and by executing a replacement algorithm in a batch mode for multiple accesses.

6.3 Wait-Free Algorithms and Transactional Memory

As lock synchronization can cause issues like performance degradation and priority inversion, wait-free synchronization [32] addresses these issues by guaranteeing that each thread completes the operation of accessing the shared resource in a limited number of steps regardless of the execution progress of other threads. 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. A lock-free implementation of the GCLOCK page replacement algorithm has been developped to reduce the lock contention of the algorithm on page misses [33]. But the implementation is only for a specific algorithm. We have not seen a wait-free implementation of an advanced replacement algorithm.

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 [34] or software [35] that implements transactional memory. It improves system scalability through enabling optimistic concurrency control [36], [37]. 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 [38]. 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.

7 Conclusion

In this paper, we address the scalability issue due to lock contention in the implementation of replacement algorithms for the management of buffer cache. We proposed an efficient and scalable framework, BP-wrapper, in which the dynamic batching and the 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 minimized. The only cost of the framework is a small FIFO queue for each worker thread, which keeps the thread's most recent access information.

We have implemented the framework and tested it with a TPC-W-like workload, a TPC-C-like workload, and synthetic workloads. 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 uses distributed locks or that does not use lock, such as the CLOCK algorithm.

Acknowledgments

This work is supported in part by the US National Science Foundation under grants CNS-1409523 and CNS-1217948.

REFERENCES

[1] (2015). Amazon. Amazon relational database service—DB instance class [Online]. Available: http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.DBInstanceClass.html

[2] T. Lane. (2005). Design notes for bufmgrlock rewrite [Online]. Available: http://grokbase.com/t/postgresql/pgsql-hackers/052df59bqt/design-notes-for-bufmgrlock-rewrite

[3] P. LLC. (2011). Improved buffer pool scalability [Online]. Available: http://www.percona.com/doc/percona-server/5.5/scalability/innodb_split_buf_pool_mutex.html

[4] G. Harrison. (2014). Resolving latch contention [Online]. Available: http://www.toadworld.com/platforms/oracle/w/wiki/371.resolving-latch-contention.aspx

[5] T. E. Anderson, "The performance of spin-lock alternatives for shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, Jan. 1990.

[6] 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.

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

[8] F. J. Corbato, *A Paging Experiment With the Multics System* in Honor of Philip M. Morse, Feshbach and Ingard, Eds. Cambridge, MA, USA: MIT Press, 1969, p. 217.

[9] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *Proc. Annu. Conf. USENIX Annu. Techn. Conf.*, 2005, p. 35.

[10] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 187–200.

[11] G. Glass and P. Cao, "Adaptive page replacement based on memory reference behavior," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 1997, pp. 115–126.

[12] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: An effective buffer cache management scheme to exploit both temporal and spatial locality," in *Proc. USENIX Conf. File Storage Technol.*, 2005, p. 8.

[13] DB2 for z/OS: DB2 database design. (2004) [Online]. Available: http://www.ibm.com/developerworks/db2/library/techarticle/dm-0408whitlark/index.html

[14] A. Da Zheng and A. S. Szalay, "A parallel page cache: IOPS and caching for multicore systems," in *Proc. USENIX Conf. Hot Topics Storage File Syst.*, 2012, p. 5.

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

[16] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2002, pp. 31–42.

[17] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan, "Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications," in *Proc. USENIX Annu. Techn. Conf.*, 2014, pp. 73–84.

[18] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. Int. Conf. Very Large Databases*, 1994, pp. 439–450.

[19] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, "The oracle universal server buffer manager," in *Proc. 23rd Int. Conf. Very Large Databases*, pp. 590–594.

[20] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proc. USENIX Annu. Techn. Conf.*, 2001, pp. 91–104.

[21] (2007). The Open Source Development Laboratory. OSDL - database test suite [Online]. Available: http://old.linux-foundation.org/lab_activities/kernel_testing/osdl_database_test_suite/

[22] (2015). Transaction Processing Performance Council. TPC-W [Online]. Available: http://www.tpc.org/tpcw

[23] (2015). Transaction Processing Performance Council. TPC-C [Online]. Available: http://www.tpc.org/tpcc

[24] X. Ding, S. Jiang, and X. Zhang, "BP-Wrapper: A system framework making any replacement algorithms (almost) lock contention free," in *Proc. IEEE 25th Int. Conf. Data Eng.*, 2009, pp. 369–380.

[25] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *Proc. 5th Annu. Workshop Model., Benchmarking Simul.*, Jun. 2009, http://www-mount.ece.umn.edu/~jjyi/MoBS/2009/MoBS_2009_Advance_Program.html

[26] Y. Xu and Y. Han, "MLB-Wrapper: Distributed high scalable BP-wrapper," in *Proc. Int. Conf. Inf. Technol. Manage. Sci.*, 2012, pp. 649–660.

[27] V. Blagojevic. (2010). Infinispan eviction, batching updates and LIRS [Online]. Available: http://blog.infinispan.org/2010/03/infinispan-eviction-batching-updates.html

[28] H. Schöning, "The ADABAS buffer pool manager," in *Proc. 24th Int. Conf. Very Large Databases*, 1998, pp. 675–679.

[29] W. Wang, "Storage management for large scale systems," Ph.D. dissertation, Dept. Comput. Sci., Univ. Saskatchewan, Saskatoon, SK, Canada, 2004.

[30] L. Huang and T. cker Chiueh, "Charm: An I/O-driven execution strategy for high-performance transaction processing," in *USENIX Annu. Techn. Conf.*, 2002, pp. 275–288.

[31] S. Yamamura, A. Hirai, M. Sato, M. Yamamoto, A. Naruse, and K. Kumon, "Speeding up kernel scheduler by reducing cache misses," in *Proc. FREENIX Track: USENIX Annu. Techn. Conf.*, 2002, pp. 275–285.

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

[33] M. Yui, J. Miyazaki, S. Uemura, and H. Yamana, "Nb-GCLOCK: A non-blocking buffer management based on the generalized CLOCK," in *Proc. IEEE 26th Int. Conf. Data Eng.*, 2010, pp. 745–756.

[34] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. 20th Annu. Int. Symp. Comput. Archit.*, 1993, pp. 289–300.

[35] N. Shavit and D. Touitou, "Software transactional memory," in *Proc. 14th Annu. ACM Symp. Principles Distrib. Comput.*, 1995, pp. 204–213.

[36] 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.

[37] 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, Jun. 1985.

[38] 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 *Proc. ACM SIGPLAN 11th Symp. Principles Practice Parallel Program.*, 2006, pp. 187–197.

Xiaoning Ding received the PhD degree in computer science and engineering from the Ohio State University. He is an assistant professor at New Jersey Institute of Technology. His interests are in the area of experimental computer systems, such as distributed systems, virtualization, operating systems, and storage systems.

Jianchen Shan received his BS and MS degrees in 2008 and 2011 both from Shanghai University, China. He is currently working toward the PhD degree in the Department of Computer Science, New Jersey Institute of technology. His research interests include parallel and distributed computing and cloud computing.

Song Jiang received the PhD degree in computer science from the College of William and Mary. He is an associate professor at Wayne State University. His research interests include file and storage systems, operating systems, and high-performance computing.