Nb-GCLOCK: A Non-blocking Buffer Management Based on the Generalized CLOCK

Makoto Yui♣, Jun Miyazaki†, Shunsuke Uemura‡ and Hayato Yamana♠

♣ *Research Institute of Information Technological Biology, Waseda University, Japan*
† *Graduate School of Information Science, Nara Institute of Science and Technology, Japan*
‡ *Faculty of Informatics, Nara Sangyo University, Japan*
♠ *Department of Computer Science and Engineering, Faculty of Science and Engineering, Waseda University, Japan*
{makoto-y|miyazaki|uemura}@is.naist.jp, yamana@waseda.jp

***Abstract*— In this paper, we propose a non-blocking buffer management scheme based on a lock-free variant of the GCLOCK page replacement algorithm. Concurrent access to the buffer management module is a major factor that prevents database scalability to processors. Therefore, we propose a non-blocking scheme for bufferfix operations that fix buffer frames for requested pages without locks by combining Nb-GCLOCK and a non-blocking hash table. Our experimental results revealed that our scheme can obtain nearly linear scalability to processors up to 64 processors, although the existing locking-based schemes do not scale beyond 16 processors.**

I. Introduction

Recent hardware trends toward multithreading for improved performance, including multi-core and multithreaded chip design, have raised critical challenges in software engineering [1]. It has also presented issues to the database community both in research [2], [3] and open source database development. Open source DBMSs, such as PostgreSQL, MySQL and Apache Derby [4], have had to face scalability problems with the increases in the number of processors. The open source DBMSs did not scale beyond four processors before revising their synchronization mechanisms in the buffer management modules.

In general, there are basically three approaches to cope with concurrency issues of synchronization:

(a) Do not acquire locks, and use a data structure that does not require locking [5]. The synchronization mechanism that avoids acquiring locks is called *non-blocking synchronization*.
(b) Reduce lock granularity. Fine lock granularity reduces lock contentions, although it may increase the overhead of locks themselves, i.e., the total time for acquiring and releasing locks.
(c) Use a more lightweight lock mechanism. Spinlock is efficient if threads are only likely to be blocked for a short period of time, as it avoids overhead from process re-scheduling or context switching in operating systems.

The open source databases dealt with the scalability issues by making improvements using (b) and (c). However, several empirical studies have shown that they have scalability limits of around 16 processors [6], [7], [8].

Database systems now demand a CPU scalability beyond 16 processors because the number of CPU cores per chip is doubling for each CPU manufacturing process in about two-year cycles. In addition, massively multithreaded processors, e.g., Sun's UltraSPARC T2 (64 processors) [9] and Azul System's Vega-2 7200 Series (768 processors) [10], have already been released as industrial products.

Most of the past research efforts on database buffer management have focused on improving their efficiency with respect to buffer hit rates on various workloads. Consequently, the literature contains very little research focusing on the concurrency of buffer management, and most of the difficulties remain to be handled by individual developers' empirical knowledge. In this paper, we propose a scalable buffer management scheme that employs non-blocking synchronization instead of acquiring locks. To the best of our knowledge, this paper is the first attempt to adopt non-blocking synchronization in buffer management.

One reason why concurrency in buffer management has not been intensively discussed is that large-scale multiprocessors have not been widespread, and also the main concerns were improving buffer hit rates and minimizing I/Os. However, the *bufferfix* operation that fixes a buffer frame for a required page [11] is not necessarily a IO-bound job. Although disk I/Os in a *bufferfix* operation certainly take place when synchronization to disk is required for a replacement victim (i.e., the replaced page that keeps its dirty flag on), modern DBMSs reduce such disk I/Os by *preflushing* dirty pages and preemptively selecting non-dirty pages for the replacement victim [11]. This means that the number of page replacements due to the *bufferfix* operation can be minimized if a large amount of memory is available and a large buffer pool can be used. In this case, the *bufferfix* operation becomes a CPU-bound task and, therefore, the CPU scalability issue in buffer management becomes particularly problematic in multiprocessor systems. Actually, *fix* and *unfix* operations to a buffer frame are the basic operations most frequently called in DBMSs. Thus, the efficiency of *fix* and *unfix* operations becomes extremely important because they lead to frequent contentions in the critical sections [12].

Several non-blocking algorithms for hash tables have already been proposed [13], [14]. In this paper, we focus on concurrency of page replacement algorithms and utilize an existing *wait-free* hash table for searching buffer frames. Then, we propose our Nb-GCLOCK page replacement algorithm, which is a non-blocking variant of the *generalized CLOCK*

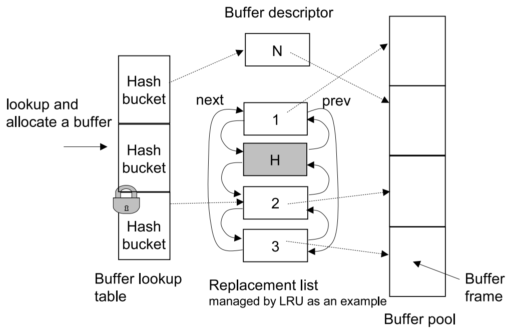


Fig. 1. Typical organization of a buffer manager.

(GCLOCK) page replacement algorithm [15]. We also verify the effectiveness of our non-blocking page replacement algorithm with respect to its concurrency and throughput using Sun UltraSPARC T2 [9]. The experimental results revealed that our scheme can obtain nearly linear scalability to processors up to 64 processors, although the existing locking-based schemes do not scale beyond 16 processors.

The rest of this paper is organized as follows. Section **II** introduces the background of the need for non-blocking page replacement. We explain why existing buffer management schemes cause scalability bottlenecks to processors. In Section **III**, we describe the details of our non-blocking page replacement algorithm. In Section **IV**, we evaluate our proposed scheme through experiments. We refer to related works in Section **V** and conclude the paper in Section **VI**.

II. Background

In this section, we explain the background of our research to address open and known problems in buffer management by giving examples. A buffer manager typically consists of a *buffer lookup table* for searching buffer frames, *buffer descriptors* to manage a page replacement policy, and a *buffer pool* as shown in Fig. 1. The *buffer lookup table* is usually constructed as a hash table [16], [11]. The purpose of the buffer lookup table is to map a page identifier to the corresponding memory page currently holding its contents. As for the page replacement policy, LRU, CLOCK, and their refinements [17], [18] are widely used. We refer to the module that manages a page replacement policy the *replacement list*.

A. Internal Locking in Buffer Manager

Access to a shared buffer cache has a significant scalability problem, particularly on multiprocessor systems. When accessing a buffer management module, the operations to the critical sections must acquire their mutual exclusions. For example in Fig. 1, to look up a buffer in the buffer pool, a *shared lock* is obtained in the *buffer lookup table*. To alter the page assignment of a buffer, an *exclusive lock* is acquired on the buffer manager. This lock must be held while adjusting the *replacement list* and changing the *buffer lookup table*. This is because the reference in *buffer lookup table* still has a different page identifier immediately after changing the page allocation of a buffer frame.

Suppose then that concurrent requests from multiple users are given. If one paging request causes a page fault and holds an exclusive lock, the exclusive lock prevents the others from holding either a shared or exclusive lock. Since system-wide *mutexes* tend to appear for each scan of pages, it would cause "mutex ping-pong" in multiprocessor and multithreaded environments. Moreover, high traffic access to a lock may causes the *convoy phenomenon* [12]. *Convoying* occurs when a thread holding a lock is descheduled by some kind of interrupt, e.g., by a page fault. Then, other threads that require the lock will queue up, unable to progress. Even after the lock is released, it may take some time to drain the queue, in an analogous fashion that an accident can slow the flow of traffic even after the debris has been cleared away.

PostgreSQL (version 8.2), MySQL (version 5.0.30) and Apache Derby coped with the lock contention problems in their buffer pools by adopting finer-grain locking schemes. They took a conventional and conservative approach [11] for refining the concurrency of a hash table, called *lock-striping*, which is a technique that divides a giant lock into clusters so as to reduce contentions. On the other hand, we attempt a more aggressive approach to synchronization toward massively multithreaded environments, rather than using the conservative one.

B. Revising Concurrency in Page Replacement Algorithms

We address here concurrency issues of page replacement algorithms by taking three practical examples: *Least Recently Used* (LRU), 2Q [17], and *Generalized CLOCK* (GCLOCK) [15].

LRU is typically arranged as a double-linked list to keep the LRU chains as shown in Fig. 1: adding new items to the head, removing items from the tail, and moving any existing items to the head when referenced (touched). When using LRU, the *replacement list* always needs to be locked when it is accessed. Thus, the LRU algorithm is effective for single-threaded applications but becomes very slow in a multithreaded environment. CLOCK [19], which has an approximately equivalent performance to LRU, is often used as the substitution [20]. CLOCK does not require a giant lock when an entry is touched. It needs only one atomic operation, e.g., setting a reference bit on or incrementing a weighting counter, on the touched entry.

GCLOCK is an efficient variation of CLOCK and uses a weighting counter instead of the use-bit of a buffer page. The references to a page P_i increment the corresponding counter $RC(i)$. In the basic GCLOCK, $RC(i)$ is initialized to 1 upon the first fetch of P_i and incremented by one every time P_i is touched. When a buffer fault occurs, a circular search is initiated, decrementing stepwise the weighting counters until the first entry with a value of 0 is found. GCLOCK improves CLOCK concerning buffer hit rates because only GCLOCK of them takes reference frequency into consideration.

Furthermore, LRU is known to be inefficient, in terms of buffer hit rates, for sequential scans and large *Inter-Reference Gaps* (IRGs) [20]. A burst of references to infrequently used pages, such as sequential scans, may cause replacement of commonly referenced pages in the cache. In [17], the authors present the scan-resistant 2Q algorithm, which divides cache items into hot and cold ones; the full version of the 2Q algorithm uses three FIFOs for managing items. The buffer manager in Oracle universal server employed a variation of

LRU that uses two separated hot/cold queues for the LRU chain management [21]. 2Q and similar algorithms as well as plain LRU management have a global contention point on their replacement lists, which degrade the scalability of buffer managers on multiprocessor systems.

To cope with sequential scans expected by database workloads, PostgreSQL 8.0 and us initially moved from LRU to the 2Q algorithm. However, as did the PostgreSQL community, we finally realized that 2Q has an unavoidable synchronization penalty on multithreaded systems. Therefore, PostgreSQL has shifted to CLOCK mostly due to the contention penalty. Similarly, we shifted to a GCLOCK refinement that employs a novel non-blocking scheme instead of a lock-based one. These facts imply that lock contentions affect the overall performance on the current hardware though minimizing paging I/Os is certainly a requirement. We provide detailed performance evaluations of these algorithms in Section **IV**.

C. Spinlock on SMT Environment

Conventional multiprocessor systems widely use spinlocks to guard *critical sections*. Both MySQL and PostgreSQL use *spin-wait loops* with backoff as their spinlock algorithms for most hardware architectures.

There are several variations of the spinlock; and past studies have shown that the Test-and-Test-and-Set (TTAS) lock with exponential backoff or a queue lock is one of the most promising spinlock protocols [22]. Spinlock is generally effective if threads are only likely to be blocked for a short period of time, since the cost of acquiring and releasing a lock is smaller than a sleep lock, though a *spin-wait loop* consumes one processor resource. However, a spin loop can be especially wasteful where logical processors share execution resources. When such loops are executed on a processor supporting Intel *Hyper-Threading* technology, they can induce an additional performance penalty due to memory-order violations and consequent pipeline flushes caused upon their exit. To ensure the proper order of outstanding memory operations, the processor incurs a severe penalty. In order to overcome this issue, Intel recommended embedding a PAUSE instruction in a spin loop [23]. PAUSE instruction introduces a slight delay in the loop and de-pipelines its execution to prevent it from aggressively consuming valuable processor resources. Zhou et al. discussed the benefits and pitfalls of using SMT processors for (in-memory) database operations in [24]. They used, in the spin-loop waiting, *PAUSE* instructions on Pentium 4.

In summary, a spinlock requires a special care (i.e., special instructions) on each hardware architecture when database operations are executed on SMT processors; on the other hand, our non-blocking buffer management scheme does not acquire any locks for searching and allocating buffer pages, and thus it is free from such difficulties in spinlocks.

III. Non-blocking GCLOCK Page Replacement Algorithm

This section explains our non-blocking buffer management scheme based on a lock-free variant of the GCLOCK page replacement algorithm, named *Nb-GCLOCK*. The key idea of our algorithm is to make buffer management fully non-blocking and optimistic including disk I/Os so that the throughput of transaction processing can be maximized. Nb-GCLOCK is entirely lock-free in both cache-hit and cache-miss cases.

Our Nb-GCLOCK algorithm basically follows the properties of GCLOCK [15] except that it allows non-blocking accesses. The reasons why we selected GCLOCK as the baseline algorithm of our non-blocking page replacement are as follows:

1) CLOCK variants are widely used due to their advantages, i.e., low overhead and high concurrency.
2) The properties and performance of GCLOCK are well analyzed and established [15], [25]. While the simple CLOCK respects only the *recency* of buffer references, GCLOCK takes *frequency* as well as *recency* into account.
3) The probability of contentions generated by concurrent accesses to shared variables is low. The contention indicates such a state that two or more processes concurrently access the same memory location. A typical CLOCK uses a single bitmap or few bitmaps to manage reference frequency. When a bitmap is frequently updated due to access skew, CLOCK becomes inefficient on cache-coherent shared memory multiprocessors due to *false sharing*. On the other hand, GCLOCK keeps a weighting counter for each buffer frame, and thus contentions rarely occur.

The unique feature of Nb-GCLOCK is that it adopts a *lock-free linearizable* page replacement. *Linearizability* is a non-blocking property: a pending invocation of a total method is never required to wait for another pending invocation to complete. Non-blocking algorithms have two important properties [26]. If some operations are guaranteed to complete within finite time, the algorithm is defined as *lock-free*. A *lock-free* algorithm guarantees that at least one process keeps its role progressing. If all operations are guaranteed to complete within a finite time, the algorithm is defined as *wait-free*. The first and second definitions guarantee the *liveness* and *fairness* properties, respectively. From this viewpoint, our proposed buffer management scheme guarantees a *lock-free* operation. We prove in Appendix that our Nb-GCLOCK is both *linearizable* and *lock-free*.

Our scheme takes a strategy that keeps trying its non-blocking operation after temporarily abandoning its execution and allows other threads to be executed when all buffer frames in the buffer pool are pinned. Due to this decision, it is impossible to guarantee that all processing will complete in a finite time when we consider the case where all pages in the buffer pool are pinned, although this is an extremely rare case. The behavior depends on whether applications that uses the cache allow failures at the buffer allocation. In a typical buffer management, a transaction is aborted when all pages are pinned [20], [11]. This abnormal condition had never been reached through our experiments.

As mentioned in Section **II**, we use an wait-free hash table for a *buffer lookup table* to achieve non-blocking synchronization on the buffer management. The non-blocking linearizable hash table has been actively developed in the literature [13], [14]. We used an existing non-blocking hash table [27]; the non-blocking hash table provides (almost[1]) *wait-freedom* — every operation has a bound on the number of steps it will take before completing. We utilized the wait-free hash table for ensuring the overall lock-freedom of Nb-GCLOCK.

In fact, a wait-free hash table is important to ensure the lock-freedom; however, it is not a dominant factor in terms of the scalability and other concurrent hash tables including blocking-ones can be the replacement because hash tables are naturally parallelizable [26] or, using a more technical term, *disjoint-access-parallel* [28], meaning that concurrent method calls are likely to access disjoint locations, implying that there is little need for synchronization. A certain concurrent hashing scheme, e.g., concurrent cuckoo hashing [26] and java.util.ConcurrentHashMap [29], use a (hash) bucket-level synchronization scheme that divides locks into buckets, and thus the contention is expected to be reduced. The point is then that hash tables can be expected to be parallelized; however, operations on replacement algorithm are generally serialized as discussed in Section **II-B** and [30].

A. Nb-GCLOCK Algorithm

While *disjoint-access-parallel* can be expected for hash tables, existing *replacement lists* including LRU and CLOCK variants are not naturally parallelizable. This section describes our carefully-designed Nb-GCLOCK algorithm in Fig. 3 and Fig. 4.

All operations to AtomicInteger and AtomicBoolean are atomically executed by using synchronization primitives such as *compare-and-swap* (CAS) and *Load-Link/Store-Conditional* (LL/SC). In SPARC V9, such an atomic operation is achieved by a native CAS instruction.

1) Organization of the Buffer Frame: The left half of Fig. 3 describes the *Frame* class that defines cached entries. A *Frame* instance is associated with a single key, a single value, and two other control variables. In buffer management, K represents a page identifier and V represents a page itself. A *wcount* instance keeps a weighting count of the entry, and a *pinning* instance is responsible for judging whether the frame is currently in use. Playing a vital role, a *pinning* instance represents an *evicted* condition when the value is -1.

The reason why we represent *evicted* and *pinned* states with a single *pinning* instance as shown in Fig. 2 is to achieve an atomic update on these states using a synchronization primitive without acquiring a lock. We introduce the state for the Frame instance whose pinning value is -1 *evicted* (see Fig. 2).

Table I explains the roles of non-obvious methods in the *Frame* class. The *pin* and *unpin* operations follow FIX and UNFIX operations of the FIX-USE-UNFIX protocol, which is generally used in buffer management [11]. The *pin*/*unpin* and *tryEvict*/*evictUnshared* methods atomically change a pinning value. The state of the pinning value changes by these four methods as shown in Fig. 2. The *pin* method increases the pinning value P only when P is greater than or equal to 0. "gt 1" in Fig. 2 represents the state where the pinning value is greater than 1. As seen in the transition, the pinning state does not change to the other states when once evicted. The

[1]The wait-freedom is violated when a hash table is resized, but *buffer lookup table* is designed not to incur resizing.

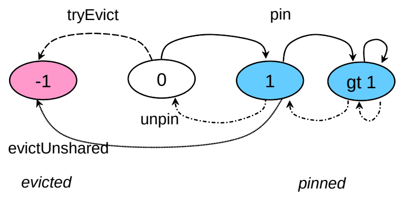


Fig. 2. State machine of a pinning instance.

volatileGetValue	returns the associated value after interleaving memory barrier for volatile load. The memory barrier is reduced to no-op in x86 or SPARC [31].
CASValue	atomically sets the field value V to the given updated value if the current value is identical to expected value.
tryEvict	atomically sets the frame evicted if the frame is not evicted, and returns true if successfully evicted and otherwise returns false.
evictUnshared	atomically sets the frame evicted.

TABLE I
ROLE OF NON-OBVIOUS METHODS IN THE FRAME CLASS.

pinning value is always one or more when an *unpin* operation is carried out. *tryEvict* succeeds only when the pinning value is 0.

2) Bufferfix Algorithm: The operation to fix a buffer frame for a requested page is called *bufferfix* in the literature [11]. The corresponding procedure to the bufferfix operation in our Nb-GCLOCK is the *fixEntry* method, except that a caller is responsible for the page fixing operation.

The right half of Fig. 3 describes the *BufferCache* class and its algorithm. *BufferCache* contains a wait-free hash table instance HASHTBL and a replacement list CLOCKBUF as its member variables. For readability, the algorithm of CLOCKBUF is separately described in Fig. 4. The *BufferCache* contains two methods: *fixEntry* for retrieving a page slot (i.e., buffer frame) and *addEntry* for allocating a page slot for the given key. In a Frame instance of Fig. 3, "key" is always stable and never be changed. The "value" is changed only through *CASValue(null, page)* in Fig. 6. *CASValue* succeeds only when another page on memory is not yet allocated to the frame.

The *addEntry* method is called when the condition in Line 50 of Fig. 3 becomes false, when a non-evicted page associated with the specified page identifier does not exist in HASHTBL. Buffer flushing is required when an evicted page has a dirty flag on in the purge operation of *addEntry*. This I/O in the purge operation is minimized in modern DBMSs as mentioned in Section **I**. The invocation of *fixEntry* fixes a frame for the specified key and increments the weighting count of the fixed frame by one.

A buffer frame may be evicted at the instant calling *pin* method. However, once *pin* method succeeds, the buffer frame never becomes inconsistent as long as its state is "gt 1" in Fig. 2. Consequently, the following Theorem 1 supports the physical stability of existing buffer frames. Moreover, a newly allocated buffer frame is not shared and thus clearly consistent.

***THEOREM** 1:* Every time an existing Frame instance F is returned by the *fixEntry* method, the *pinning* value of F is incremented by one.

Proof: [Proof of Theorem 1] An existing Frame instance F is returned by the fixEntry invocation only when the

```
class Frame {
 1 K key; V value;
 2 AtomicInteger wcount = new AtomicInteger(1);
 3 AtomicInteger pinning = new AtomicInteger(1);
 4 Frame(K key, V value) {
 5  this.key = key;
 6  this.value = value;
 7 }
 8 V volatileGetValue() {
 9  memory fence for volatile load
10  return value;
11 }
12 boolean CASValue(V expect,V update) {
13  return CAS(value, expect, update);
14 }
15 void incrWC() {
16  wcount.increment();
17 }
18 boolean decrWC() {
19  return wcount.decrement();
20 }
21 boolean tryEvict() {
22  return pinning.CAS(0, -1);
23 }
24 void evictUnshared() {
25  pinning.CAS(1,-1);
26 }
27 int pinCount() {
28  return pinning.get();
29 }
30 boolean pin() {
31  int x;
32  do {
33    x = pinning.get();
34    if(x <= -1)
35      return false;
36  } while(!pinning.CAS(x, x + 1));
37  return true;
38 }
39 void unpin() {
40  pinning.decrement();
41 }
 } //end Frame
```

```
class BufferCache {
42 HashTable HASHTBL;
43 ClockBuffer CLOCKBUF;
44 BufferCache(int size) {
45  HASHTBL = new HashTable(size);
46  CLOCKBUF = new ClockBuffer(size);
47 }
48 Frame fixEntry(K key) {
49  Frame entry = HASHTBL.get(key);
50  if(entry != null && entry.pin()) {
51   entry.incrWC();
52   return entry;
53  } else {
54   return addEntry(key, null);
55  }
56 }
57 Frame addEntry(K key, V value) {
58  for(;;) {
59   Frame newEntry = new Frame(key, value);
60   Frame removed = CLOCKBUF.add(newEntry);
61   if(removed != null) {
62    if(HASHTBL.remove(removed.key, removed))
63     purge the removed page
64   }
65   Frame prevEntry = HASHTBL.putIfAbsent(key, newEntry);
66   if(prevEntry != null) {
67    if(!prevEntry.pin()) {
68     if(HASHTBL.replace(key,prevEntry,newEntry)) {
69      newEntry.setValue(prevEntry.getValue());
70      return newEntry;
71     }
72     newEntry.evictUnshared();
73     continue; //jump to Line 59
74    }
75    newEntry.evictUnshared();
76    prevEntry.incrWC();
77    return prevEntry;
78   }
79   return newEntry;
80  }
81 }
} //end BufferCache
```

Fig. 3. Pseudo code of the buffer cache.

condition in Line 50 or 67 of Fig. 3 becomes true or false, respectively. Whenever these conditions are met, it is clear that the pinning value of F was incremented by one according to the *pin* specification. ■

***COROLLARY** 1:* From Theorem 1, a Frame instance successfully evicted by *evictUnshared* is never used outside the Frame class.

3) CLOCK-sweep Algorithm: Selecting and swapping a replacement victim in the buffer pool of CLOCK is called the *clock-sweep* operation.

Fig. 4 describes the *ClockBuffer* class which manages the Nb-GCLOCK page replacement policy. It contains four member variables: an atomic array "POOL" as the buffer pool, an atomic "Free" counter responsible for managing the number of free-slots in the buffer pool, an atomic "CLOCKHAND" representing a circulating clock hand, and a "SIZE" field representing the capacity of the buffer pool. *AtomicArray* class used for POOL provides atomic operations to an array. A method invocation *CAS(index, expect, update)* on AtomicArray atomically sets the existing value of a specified index to an updated value if the current value is identical to the expected one.

The ClockBuffer class has a single entry point on the *add* method. The *add* method fixes the given frame to the buffer pool. The *swap* method is invoked when the *add* method replaces an existing frame with the new frame according to GCLOCK page replacement policy. The *moveClockHand* method moves the clock hand in a style of atomic add instructions. We used this "add" scheme because "set" instructions to a clock hand are not robust for multithreaded accesses. Therefore, we add a "delta" (i.e., an absolute difference).

We now provide theorems to give consistency to the *add* algorithm.

***THEOREM** 2:* A given entry is always fixed to a free space in the buffer pool whenever decrementing the FREE instance succeeds in Line 15.

Proof: [Proof of Theorem 2] It is clear that at least one free space is ensured at the instant when decrementing the FREE instance succeeds in Line 15. However, another thread may seize free space upon entering *swap* in Line 14 immediately after the success. To cope with this case, the *swap* method avoids using any free space and gives free space to other threads in Line 28. Thus, the *add* method fixes the given $entry$ to free space in the buffer pool in Line 19 whenever decrementing the FREE instance succeeds. ■

***THEOREM** 3:* On *add* method call, the same Frame instance will never be returned to a different invocation.

Proof: [Proof of Theorem 3] When *add* method call returns a non-null value, the *swap* method is called to return an evicted Frame instance. The evicted instance will never be managed in the *replacement list* (see Line 60 to 64 of Fig. 3).

Between Lines 27 and 47 in the for-loop of *swap* method,

```
class ClockBuffer {
 1 AtomicArray POOL;
 2 AtomicInteger FREE;
 3 AtomicCounter CLOCKHAND = new AtomicCounter(0);
 4 int SIZE;
 5 ClockBuffer(int size) {
 6  this.POOL = new AtomicArray(size);
 7  this.FREE = new AtomicInteger(size);
 8  this.SIZE = size;
 9 }
10 Frame add(Frame entry) {
11  do {
12   int free = FREE.get();
13   if(free == 0)
14     return swap(entry);
15   if(FREE.CAS(free, free - 1))
16     break;
17  } while(true);
18  int idx = CLOCKHAND.get();
19  while(!POOL.CAS(idx%SIZE, null, entry))
20    idx++;
21  CLOCKHAND.increment();
22  return null;
23 }
23 Frame swap(Frame entry) {
24  int numpinning = 0;
25  int start = CLOCKHAND.get();
26  for(int i=start%SIZE;;i=(i+1)%SIZE) {
27   Frame e = POOL.get(i);
28   if(e == null) continue;
29   int pincount = e.pinCount();
30   if(pincount == -1) { // evicted?
31     if(POOL.CAS(i,e,entry)) {
32       moveClockHand(i, start);
33       return e;
34     }
35     continue;
36   }
37   if(pincount > 0) { // pinned?
38    if(++numpinning>=size)
39      yield this thread and allow others to execute
40    continue;
41   }
42   if(e.decrWC() <= 0) {
43    if(e.tryEvict() && POOL.CAS(i,e,entry)) {
44      moveClockHand(i, start);
45      return e;
46    }
47   }
48  } //end for
49 } //end swap
50 void moveClockHand(int curr, int start) {
51  int delta;
52  if(curr < start)
53    delta = curr + size - start + 1;
54  else
55    delta = curr - start + 1;
56  CLOCKHAND.add(delta);
57 }
} //end ClockBuffer
```

Fig. 4. Pseudo code of the ClockBuffer.

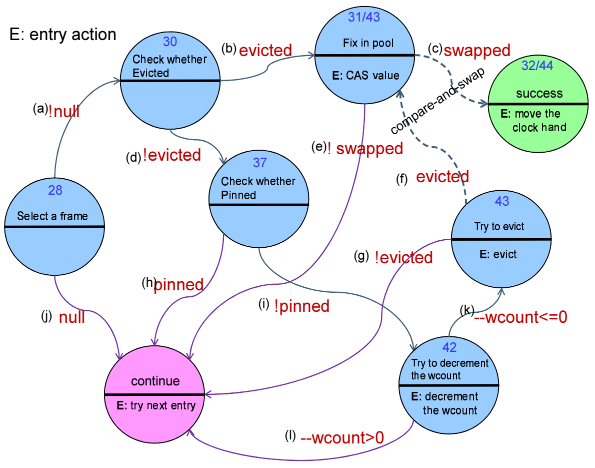


Fig. 5. State transitions in clock-sweep.

the state of a Frame instance e changes as shown in Fig. 5. When e is returned through the state 32/44, a *compare-and-swap* operation removes e from the buffer pool at the transition (c). Therefore, the same Frame instance will never be returned to a different invocation. ■

B. Optimistic and Concurrent I/Os

In certain scenarios, there are race conditions in which multiple threads attempt the same I/O operation on a fixed frame concurrently. A process of a conventional buffer management thus waits until the *io_in_progress* lock on a fixed frame is released when someone else has already started I/O on the buffer [11]. Though our Nb-GCLOCK makes *bufferfix* operations non-blocking, the *page-in* operation to a fixed frame seems to remain an open problem.

To make this *page-in* operation non-blocking, our non-blocking scheme acts optimistically as shown in Fig. 6.

```
 1 Frame slot = PAGE_CACHE.fixEntry(pageId);
 2 try {
 3  V page = slot.volatileGetValue();
 4  if(page == null) {
 5    page = read-in a page of the pageId from disk
 6    slot.CASValue(null, page);
 7  }
 8  do application logic for the page
 9 } finally {
10  slot.unpin();
11 }
```

Fig. 6. Usage of a buffer in our scheme.

Existing I/O schemes acquire a lock before reading a page and releases the lock after associating the page to a frame. Taking a lock before reading a page from disk may be reasonable, since *lseek* and *read* system calls also require mutex exclusion. On the other hand, our scheme does not delay the concurrent I/O. Our optimistic I/O scheme is non-blocking by utilizing a *pread* system call (Line 5), a memory barrier (Line 3) and a *Compare-and-Swap instruction* (Line 6). The *pread*/*pwrite* system calls enable efficient I/O to the same file descriptor from multiple threads. They are even thread-safe and thus I/Os using them do not need user-level locking mechanisms.

Note then that the following would fill in the background of our concurrent I/O strategies.

- Current secondary storage systems including cheap SATA disks and SSD storages have I/O command queuing facilities that allows optimally re-ordering the execution of I/O commands. SATA II NCQ supports a command depth of 32 and SCSI disks can queue up to 255 commands.
- I/O requests can be reordered based, for example, on a famous elevator algorithm by I/O schedulers of modern operating system kernels and are not directly passed to device drivers.

Though detailed analysis using several storage systems and operating systems is beyond the scope of this paper, we provide a performance evaluation between a traditional blocking I/O scheme and our optimistic and concurrent I/O scheme in

Section **IV-A.2**.

IV. Experimental Evaluation

In order to evaluate the effectiveness of Nb-GCLOCK, we compared Nb-GCLOCK with LRU, GCLOCK [15], and the full version of 2Q [17]. We need to clarify how much our proposed technique improves the performance for CPU-bound and I/O-bound jobs. One of the main factors in determining whether a job is CPU-bound or I/O-bound is the number of disk I/Os, which depends on the buffer hit rates as well as the ratio of dirty pages in replacement victims. Therefore, we focus on buffer hit rates at first in Section **IV-A.1** and then give considerations to the scalability to the number of processors when buffer hit rates change.

As for the workloads, we followed the example provided in the paper [17] in which the authors tried a mixed workload containing both random accesses with *Zipfan* distributions and scans because database workloads generally contain scans. For the parameters of 2Q, we used 20% and 30% of the buffer spaces for *K1in* and *K1out*, respectively. TTAS lock with exponential backoff is used for the synchronization of blocking algorithms.

We performed experiments on a real machine with a Sun UltraSPARC T2 processor (Sun SPARC Enterprise T5120 box). The detailed specification is shown in Table II. The processor has eight CPU cores, and each core is able to handle eight threads concurrently. Thus, the processor is capable of processing up to 64 concurrent threads. We used the Sun JDK 1.6 for the runtime environment in all of the experiments.

Operating System	Solaris 10 8/07
Core (Threads/Core)	8 (8)
Processor frequency	1.2 GHz
Main memory	16 GB
Disk	SAS (10000 rpm)
L2 cache per core	4M

TABLE II
Specifications of Sun SPARC Enterprise T5120.

A. Experiments on Mixed/Zipfan Distributions

The experiments in this section used artificially generated workloads using a *Zipfan* input distribution [32] with parameters α = 0.5 and α = 0.86. If there are N pages, the probability of accessing a page numbered i or less is $(i/N)^{\alpha}$, a setting of α = 0.86 gives an 80/20 distribution, while a setting of α = 0.5 give a less skewed distribution (about 45/20). When running the Zipf simulator, we modified the workloads so that it would occasionally start scans. We used mixed workload of Zipf with 20% scans of 100 pages, and the page size commonly used throughout our experiments is 8 KB. To emulate a multi-user scenario, the workloads are concurrently issued from multiple threads in which each thread uses its own simulator and workload. We simulated a database consisting of 4,000,000 pages (32GB) and used the buffer capacity ranging from 4,096 to 32,768 pages.

1) Relation to Buffer Hit Rate: Fig. 7 shows the relationship between buffer capacities and the buffer hit rates when 64 threads concurrently ran the above mentioned 80/20 workload on UltraSPARC T2. With decrease in the buffer capacity, 2Q shows better buffer hit rates than others. This result is natural

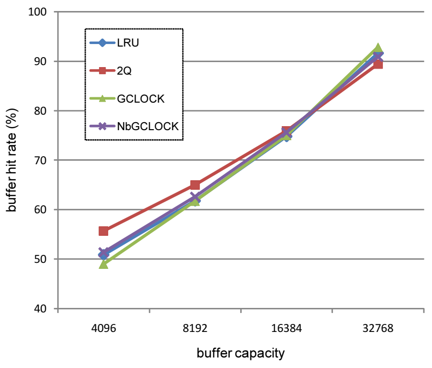


Fig. 7. Relationship between buffer capacity and buffer hit rate (64 threads).

because 2Q is effective for sequential scans and is expected to provide better buffer hit rates than LRU/CLOCK [17]. Of course, enough buffer capacity minimizes the differences as the result shows in Fig. 7; however, highly concurrent accesses cause almost random access to buffers. 2Q has two predetermined parameters (*K1in* and *K1out*), which need to be carefully tuned. Tuning buffer regions with a little overhead is crucial [17], and it becomes the reason that 2Q lost its advantages where the buffer capacity is 32768.

We also evaluated here the batching and prefetching[2] technique introduced in [30] for page replacement of LRU and 2Q. We set the configuration parameters of Bp-Wrapper as follows: the FIFO queue size to 64 and batch threshold to 32 as used in [30]. Fig. 8 shows throughputs of the experiment varying the buffer capacity and *Zipf* distributions. The batching technique, accumulating a set of page accesses to make corresponding replacement operations within one lock-holding period, is apparently efficient when all pages are cached as evaluated in [30]. However, the scheme has a pitfall when the moderate cache misses caused in Fig. 8 because a cache miss induces the batching; the batching involves the larger lock holding time, and thus, the serious contentions happened in the critical section. It was much better to acquire a spin-lock to LRU/2Q in our setting (much more processors and cache-misses than ones of [30]) because the *bufferfix* operation takes a short period of time and the spin-lock avoided overhead from process re-scheduling or context switching. From our experience, Bp-wrapper was beneficial in reducing lock acquisition cost of sleep locks.

Since our Nb-GCLOCK basically follows GCLOCK, the hit rate shows similar tendency to that of GCLOCK. We focus, in the following experiments, on buffer hit rates because throughput of GCLOCK variants apparently depends on buffer hit rates.

2) I/O in Progress and Concurrent I/Os: As described in Section **III-B**, our scheme utilizes non-blocking (at user-level) I/O scheme using a *pread* system call, a memory barrier and a *Compare-and-Swap* instruction. In a general blocking scheme, redundant I/Os never happen; However, transactions tend to be blocked. On the other hand, when our non-blocking I/O scheme of Fig. 6 is utilized, it does not need to block transactions; however it causes additional I/Os as shown in

[2]Thier scheme simply accesses the head nodes and the siblings of touched nodes in FIFO lists without CPU prefetching instructions.

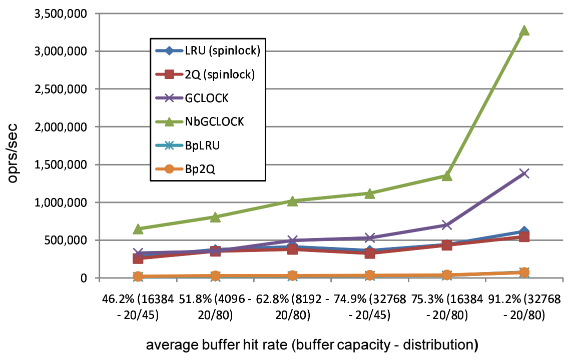


Fig. 8. Throughputs obtained when varying buffer capacity and workload distributions.

Table III. Then, it needs to be proven that

- to what extent contentions are expected on the critical section (i.e., Lines 3 to 7 in Fig. 6), and
- which strategy is effective for (massively) parallel workloads.

Fig. 9 shows an experimental result on the 80/20 workload on the UltraSPARC T2 comparing the proposed scheme with the existing lock-based schemes. In Fig. 9, "non-blocking I/O" and "blocking I/O" respectively denote our I/O scheme introduced in Section **III-B** and a general blocking I/O scheme using *lseek* and *read* system calls.

To answer the first question, we counted CAS failures generated at Line 6 of Fig. 6, and the results are shown in Table III. From Table III and Fig. 9, our non-blocking scheme is effective even when contentions occur at a probability of 1.7%, which lead us to expect that the proposed technique becomes more effective as the buffer capacity increases while a certain threshold may exist. The Nb-GCLOCK scheme incurred more contentions in CAS operations compared to those in other schemes in Table III because more failures can be observed as the system throughput increases. This increasing throughput without compromising buffer hit rates is the ideal goal of caching database pages. On the other hand, the long lock holding time of blocking I/Os leads to more serious contentions that cause context-switches and process rescheduling and overshadows the throughput.

It is notable that we used a single disk in the experiments, and thus this approach could be more effective on a high-throughput disk configuration such as RAID 0.

3) Scalability to Processors: We ran a series of experiments that varies the number of processors. In the experiments, we disabled/enabled processors by using the *psradm* command provided by Solaris.

The first experiment measured the scalability to processors when all pages are resident in memory. This experiment intended to see the scalability limit expected by each algorithm in light of adopting the non-blocking scheme to high I/O throughput configurations. The experimental results in Fig. 10 show that our non-blocking scheme denoted as "Nb-GCLOCK(stripe)" is nearly scalable up to 64 processors. The scalability of "Nb-GCLOCK(atomic)", which uses an AtomicInteger class for the CLOCKHAND in Fig. 4, declined between 33 and 64 processors because the naive implementation of our Nb-GCLOCK(atomic) has a global contention point

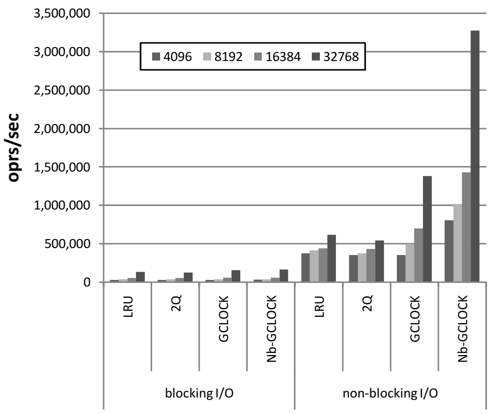
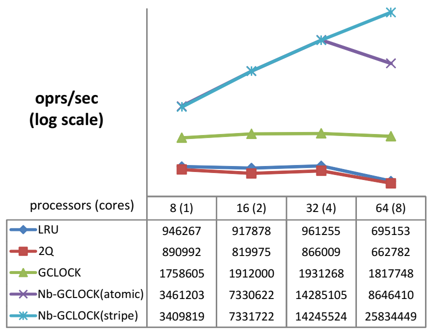


Fig. 9. Comparison between "blocking I/O" and "non-blocking I/O".



processors (cores)	8 (1)	16 (2)	32 (4)	64 (8)
LRU	946267	917878	961255	695153
2Q	890992	819975	866009	662782
GCLOCK	1758605	1912000	1931268	1817748
Nb-GCLOCK(atomic)	3461203	7330622	14285105	8646410
Nb-GCLOCK(stripe)	3409819	7331722	14245524	25834449

Fig. 10. Scalability to processors when pages are resident in memory.

on the CLOCKHAND. The AtomicInteger uses *compare-and-swap* operations for each decrement/increment operation, and thus the bus lock decreases its scalability. Moreover, the costful write operations cause CPU store buffers to flush.

Our Nb-GCLOCK(stripe) employs a striped counter as shown in Fig. 11, which scales beyond 32 processors. On the other hand, it can be concluded that the existing locking-based schemes did not scale more than 16 processors according to the results in Fig. 10.

```
AtomicInteger cnt[];
int get() {
 int sum = 0;
 for(AtomicInteger i: cnt)
  sum += i.get();
 return sum;
}
void add(int x) {
 int idx = hash value of current thread % cnt.length;
 cnt[idx].add(x);
}
void increment() {
 add(1);
}
```

Fig. 11. Internal design of AtomicCounter class.

We also conducted a performance measurement on varying the number of processors when disk I/Os were performed by using *pread*. The results in Fig. 12 showed that only the proposed scheme can obtain at least log-linear performance relative to the number of processors up to 64 processors. "E$Nb-GCLOCK" denotes the expected scalability to processors according to the result of Nb-GCLOCK on 8 processors.

Based on the above experimental results, we conclude that

buffer capacity	LRU		2Q		GCLOCK		Nb-GCLOCK	
	Contention	CPU time (sec)	Contention	CPU time (sec)	Contention	CPU time (sec)	Contention	CPU time (sec)
4096	10619 (0.04%)	1000 (26%)	5342 (0.03%)	894.3 (23.3%)	10177 (0.05%)	983.9 (25.6%)	88396 (1.7%)	3699.8 (96.3%)
8192	5650 (0.02%)	907.9 (23.6%)	3329 (0.01%)	831.9 (21.7%)	7993 (0.03%)	1068.1 (27.8%)	65034 (0.1%)	3690.3 (96.1%)
16384	2477 (0.01%)	720.6 (18.8%)	2077 (0.008%)	721.3 (18.8%)	6157 (0.01%)	1080.8 (28.1%)	53028 (0.06%)	3683 (95.9%)
32768	733 (0.002%)	544.3 (14.2%)	998 (0.003%)	648.8 (16.9%)	5538 (0.006%)	1059.2 (27.6%)	60447 (0.03%)	3561.1 (92.7%)

TABLE III
CONTENTIONS GENERATED BY PREAD.

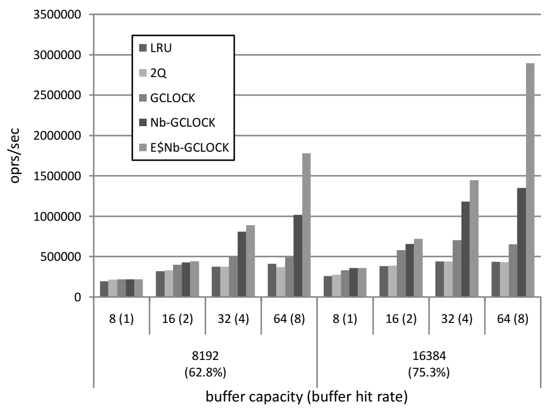


Fig. 12. Scalability to processors when using pread for disk I/O.

our non-blocking scheme is much more scalable than existing schemes in certain situations and has a significant advantage over the existing blocking schemes, as confirmed with Fig. 10 where all pages are resident in memory and Fig. 12 where disk I/Os are performed by pread under adequate hit rates.

B. Experiments on x86-64 Architecture

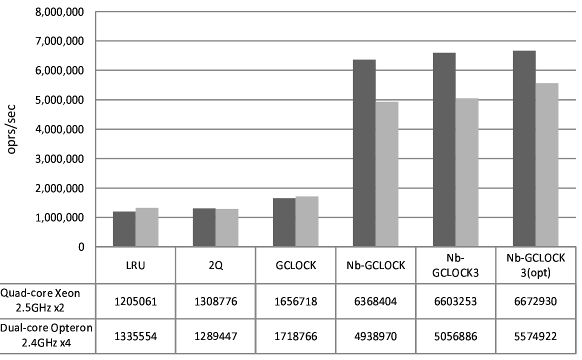
In Section **IV-A**, our non-blocking scheme showed significant performance improvement on a SUN UltraSPARC T2. However, it still needs to prove its efficiency on other architectures. We have thus conducted an experiment on two different x86-64 architectures as listed in Table IV.

Operating System	Linux 2.6.22 OpenSUSE 10.3	Linux 2.6.5 SuSE (SLES) 9
CPU model	Quad core Xeon E5420	Dual Core Opteron 880
Architecture	SMP	ccNUMA
Core (Chips)	8 (2)	8 (4)
Processor frequency	2.5 GHz	2.4 GHz
Main memory	8 GB	32 GB
Disk	SATA 2 (7200 rpm, NCQ)	Ultra320 SCSI (10000 rpm)
L2 cache per core	6 MB	1 MB

TABLE IV
SPECIFICATIONS OF EACH X86-64 MACHINE.

We compared our non-blocking buffer management scheme using Nb-GCLOCK to the existing locking-based schemes with respect to throughputs where all pages are resident in memory and we assume eight concurrent accesses (i.e., the same as the number of processors) to the module. We used LRU and 2Q for the page replacement algorithms, as with the existing locking-based schemes, and performed the 80/20 workloads for each algorithm.

Fig. 13 shows the results of the experiment. Our schemes (all variations of Nb-GCLOCK) outperform the existing locking-based ones by more than 5 and 4 times on the Xeon SMP architecture and the Opteron ccNUMA architecture, respectively. Note that this performance gain for 8 concurrent accesses is similar to the one expected on the SUN UltraSPARC T2 (at most 4.78 times in Fig. 10).



	LRU	2Q	GCLOCK	Nb-GCLOCK	Nb-GCLOCK3	Nb-GCLOCK 3(opt)
■ Quad-core Xeon 2.5GHz x2	1205061	1308776	1656718	6368404	6603253	6672930
■ Dual-core Opteron 2.4GHz x4	1335554	1289447	1718766	4938970	5056886	5574922

Fig. 13. Experiment on X86-64 architecture (8 threads).

The clear differences appearing on Nb-GCLOCK between the two architectures can be attributed to the greater number of contentions among chips performed on the Opteron configuration as the throughput increases. Of course, CAS incurs more latency on a four-chip configuration than a two-chip configuration because CAS (i.e., cmpxchg) causes bus locks. To reduce *false sharing* in the Opteron configuration, “Nb-GCLOCK3(opt)” striped the memory location of array elements used for the buffer pool as depicted in Fig. 14.

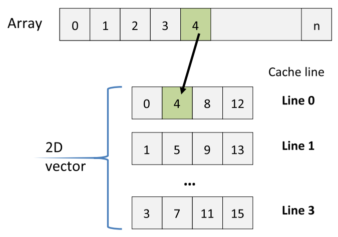


Fig. 14. Expressing an array as a two-dimensional vector.

The “Nb-GCLOCK3” means that the maximum value of Nb-GCLOCK’s weighting counter is restricted to 3. This effort is introduced to reduce CAS instructions because CMPXCHG is very costly on Intel x86 multiprocessor systems while an UltraSPARC T2 processor has a very cheap CAS instruction. UltraSPARC T2 and Opteron multiprocessor systems show better CAS and CMPXCHG performance [33], [34].

Based on the above results, we conclude that even under medium multithreaded environments of x86 architectures, our proposed non-blocking scheme can provide better performance than the existing lock-based schemes and is thus the algorithm of choice.

C. TPC-C Benchmark on Derby

We have implemented our non-blocking buffer management scheme on Apache Derby [4][3]. Apache Derby is an open

[3]The source code is publicly available on http://code.google.com/p/derby-nb/.

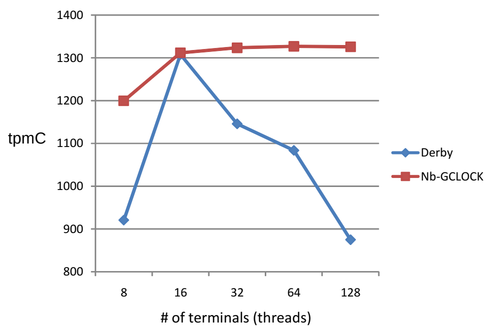


Fig. 15. Comparison of tpmC between Nb-GCLOCK and Derby's CLOCK.

source RDBMS written in Java and shows competitive performance to other open source RDBMSs [35]. Apache Derby used a striped hash table for the buffer lockup and takes a standard CLOCK for the page replacement scheme. We compared our non-blocking scheme (Nb-GCLOCK) on T5120 (see Table II), with varied number of threads, to Derby's one (Derby) with respect to transactions per minute C (tpmC); the rating from the TPC-C benchmark [36] representing OLTP workloads. As the scaling factor of TPC-C, we set the number of warehouses $W = 16$, which gives 2.1GB databases. We used 400MB that can hold 19 % of the database as the size of the buffer pool. The evaluation result using a TPC-C 5.4 benchmark suite is shown in Fig. 15.

From this result, it can be concluded that our non-blocking scheme does not decrease the throughput on the TPC-C setting. On the other hands, Derby's buffer management scheme clearly decreases the system throughput when the number of concurrent accesses grows beyond 16. There appeared only a marginal increase in throughput beyond 16 threads because, as discussed in [8], Derby decreases its throughput due to the contentions on the shared latch of the root node of the B-Tree index. It is known that revising other database modules is indispensable toward a high-throughput transactional processing [37], [6] because each module can reduce the overall throughput, and we showed in Section **IV-A** that Nb-GCLOCK improves the largest bottleneck, according to [37], in buffer management modules. Again, it is confirmed by Fig. 15 that our Nb-GCLOCK does not reduce the throughput.

The throughput obtained in buffer manager actually matters on the scalability; contentions on synchronization happen in the buffer manager under high throughput accesses. If there are few contentions and the throughput on 16 threads is low, the difference in CPU scalability between CLOCK and Nb-GCLOCK becomes slight. We assume it as the major reason of the similar performance where the number of threads is 16. We are considering from Fig. 15 that hit-rates were a major dominant factor when less than 9 threads and contentions become problematic when the number of threads is greater than 16, concerning the original Derby with CLOCK page replacement. This result is consistent with our conclusion in Section **IV-A.3** that the existing locking-based schemes have scalability limits on around 16 processors.

V. Related Work

Tsuei et al. [38] designed experiments to investigate how the database size, the buffer size, and the number of CPUs impact database performance, in particular, throughputs and buffer hit rates on Symmetric Multiprocessor (SMP) systems. They investigated the impact of buffer size on performance using the TPC-C workload and observed that an adequate memory buffer size is relatively small compared to the database size. They also suggested a rule-of-thumb of 10-15% of the database size to achieve more than an 80% buffer hit rate. In fact, DBT-1 and DBT-2 of OSDL database test suite [39], which derive TPC-W and TPC-C respectively, generates high hit-rates (greater than 95%) on PostgreSQL 8.2 when a moderate buffer space (256MB or more) to the database size (6.8GB and 5.6GB) can be expected [30]. As shown in our experiment in Fig. 9, Nb-GCLOCK becomes very effective when the buffer hit rate is about 80% or more. These requirements for buffer capacity can realistically be considered acceptable because 64-bit processors, which have a huge address space, have become widespread and, moreover, DRAM has become dense and cheap.

To avoid lock contentions on the LRU chain, ADABAS [40] splits the buffer pool into several physical regions, where each region has its own LRU chain. This approach can reduce buffer hit rates, especially when the distribution of hash values has skew. Moreover it is unsuited for massively multithreading settings because buffer hit rates apparently decrease when dividing a LRU chain into more finer regions according to a number to processors while it is mandatory for reducing contentions. In addition, they did not discuss how buffer hit rates change by dividing the LRU chain.

Bp-wrapper [30] introduced a batching technique to database buffer management. The batching technique can be categorized as a sort of *lazy synchronization* [41], [26]; postpones the *physical* work (adjusting the buffer replacement list) and immediately returns the *logical* operation. Bp-wrapper works with any replacement algorithm and eliminates lock contention on buffer hits. However, according to the results in [30], the advantage is limited to LRU variants and does not accelerate throughputs of CLOCK variants because the scheme does not avoid lock contention on buffer misses. If one of concurrently accessed threads (e.g., 1/10) encounters a buffer miss and acquire an exclusive lock, Bp-wrapper must order a blocking operation. On buffer hits, CLOCK does not need to acquire locks and thus their scheme did not accelerate the current buffering scheme (CLOCK) of PostgreSQL. On the other hand, our scheme is an effective replacement of CLOCK variants on many-core settings.

We introduced, in Section **IV-A.3**, a striped counter for the highly contending CLOCK hand. Shared counting on shared memory multiprocessors has been studied, for example in [42], [43].

Transactional memory is another approach to address the issue of lock contention. While hardware transactional memory has not been publicly available, various implementations of software transactional memory (STM) exist. STM improves system scalability through enabling optimistic concurrency control in a similar way to non-blocking algorithms. However, the overhead proposed by current STM implementations is known to reduce system throughputs and overshadow their promise [44], though STMs improve scalability. In contrast, our lock-free scheme does not decrease system throughputs as shown through the experiments. Recall that lock-freedom guarantees a certain throughput: any active thread taking a bounded number of steps ensures global progress.

To the best of our knowledge, OS community has not yet been working on non-blocking synchronization on page replacement policy though they developed a concurrent radix-tree for searching a buffer frame and utilized *Read-Copy-Update* (lock-free on read operation but not completely lock-free when write operation happened) data structures. Buffer management strategies between database systems and operating systems are quite different in terms of requirements for power-saving capability; operating systems must consider mobile devices and laptop computers while database systems, in general, run on server machines. We assume that improving power saving capability of Nb-GCLOCK would be challenging and analyzing tradeoff between resource utilization of the system and the power saving capability is interesting toward a greener computing.

It is to be noted that describing a non-blocking implementation, including ours, in C-style requires a non-blocking malloc and a safe memory reclamation scheme, e.g., *hazard pointer* [45] or a concurrent garbage collector, as used in [46].

VI. Conclusions

In this paper, we proposed a lock-free variant of the GCLOCK page replacement algorithm, named Nb-GCLOCK. We introduced a non-blocking scheme for *bufferfix* operations that fix buffer frames for requested pages without locks by combining Nb-GCLOCK and a wait-free hash table. Our experimental results revealed that our scheme can obtain nearly linear scalability to processors up to 64 processors, although the existing locking-based schemes do not scale beyond 16 processors. We assume Nb-GCLOCK, in which a nonblocking synchronization is firstly introduced to database buffer management, takes major step towards in development of database systems for the many-core era. Lock-free algorithms, including ours, generally works much better than blocking ones for massively multithreading settings though it is hard to expect the exact performance when processors become more than 64 because large difference in the numbers of processors causes non-negligible changes in the computer architectures.

Gray et al. suggested in [11] that a future database system might introduce its page replacement at random since it could have a huge buffer pool. This can be a strategy of choice in a certain situation, though the behavior of the worst case cannot meet the needs of critical systems. Ensuring lock-freedom operation, as in our scheme, is considered preferable for such practical requirements.

The proposed scheme is effective not only for database buffer management but also for general purpose caching that requires synchronization. One example of such an application would be scalable query result caching for web applications, where the requests come from over 1000 clients simultaneously [47].

Acknowledgment

This work was supported in part by MEXT (Grant-in-Aid for Scientific Research on Priority Areas #21013035), MEXT (Grant-in-Aid for Young Scientists (B) #21700111) and JSPS research fellowships for young scientists from the Japan Society for the Promotion of Science.

References

[1] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, March 2005.

[2] J. Cieslewicz, J. Berry, B. Hendrickson, and K. A. Ross, "Realizing Parallelism in Database Operations: Insights from a Massively Multithreaded Architecture," in *Proc. DaMoN*, 2006.

[3] J. Cieslewicz, K. A. Ross, and I. Giannakakis, "Parallel Buffers for Chip Multiprocessors," in *Proc. DaMoN*, 2007.

[4] Apache Software Foundation, "The Apache Derby Project," http://db.apache.org/derby/.

[5] J. D. Valois, "Lock-free Data Structures," Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, NY, USA, 1996.

[6] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: A Scalable Storage Manager for the Multicore Era," in *Proc. EDBT*, 2009.

[7] D. Tolbert, "Scaling PostgreSQL on SMP Architectures – An Update," in *The PostgreSQL Conference*, 2007.

[8] A. Morken, "Apache Derby SMP scalability," Master's thesis, Norwegian University of Science and Technology, June 2007.

[9] Sun Microsystems, Inc., "UltraSPARC T2 Processor," http://www.sun.com/processors/UltraSPARC-T2/.

[10] "Azul Systems, Inc." http://www.azulsystems.com/.

[11] J. Gray and A. Reuter, *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, 1992.

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

[13] O. Shalev and N. Shavit, "Split-Ordered Lists: Lock-Free Extensible Hash Tables," *Journal of the ACM*, vol. 53, no. 3, pp. 379–405, May 2006.

[14] C. Purcell and T. Harris, "Non-blocking Hashtables with Open Addressing," in *Distributed Computing (DISC)*. Springer Berlin, 2005, pp. 108–121.

[15] A. J. Smith, "Sequentiality and Prefetching in Database Systems," *ACM Trans. Database Syst.*, vol. 3, no. 3, pp. 223–247, September 1978.

[16] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM Trans. Database Syst.*, vol. 9, no. 4, pp. 560–595, 1984.

[17] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *Proc. VLDB*, 1994, pp. 439–450.

[18] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," in *Proc. USENIX*, April 2005, p. 35.

[19] 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.

[20] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, third ed. ed. McGraw-Hill Science/Engineering/Math, August 2002.

[21] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. Macnaughton, "The Oracle Universal Server Buffer," in *Proc. VLDB*. Morgan Kaufmann Publishers Inc., 1997, pp. 590–594.

[22] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Money Multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, January 1990.

[23] Intel Corporation, "Using spin-loops on Intel Pentium 4 processor and Intel Xeon processor," Order Number: 248674-002, May 2001.

[24] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah, "Improving Database Performance on Simultaneous Multithreading Processors," in *Proc. VLDB*. VLDB Endowment, 2005, pp. 49–60.

[25] V. F. Nicola, A. Dan, and D. M. Dias, "Analysis of the generalized clock buffer replacement scheme for database transaction processing," *SIGMETRICS Perform. Eval. Rev.*, vol. 20, no. 1, pp. 35–46, June 1992.

[26] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[27] C. Click, "high-scale-lib," http://sourceforge.net/projects/high-scale-lib.

[28] A. Israeli and L. Rappoport, "Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives," in *Proc. PODC*, 1994, pp. 151–160.

[29] D. Lea, "JSR 166: Concurrency Utilities," http://jcp.org/en/jsr/detail?id=166, 2004.

[30] X. Ding, S. Jiang, and X. Zhang, "BP-Wrapper: A System Framework Making Any Replacement Algorithms (Almost) Lock Contention Free," in *Proc. ICDE*, 2009.

[31] D. Lea, "The JSR-133 Cookbook for Compiler Writers," http://gee.cs.oswego.edu/dl/jmm/cookbook.html.
[32] D. E. Knuth, *Art of Computer Programming*. Addison-Wesley Professional, April 1998, vol. 3.
[33] K. Russell and D. Detlefs, "Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing," *SIGPLAN Not.*, vol. 41, no. 10, pp. 263–272, October 2006.
[34] "mubench," http://mubench.sourceforge.net/.
[35] O. Sandstå, D. Tjeldvoll, and K. A. Hatlen, "Apache Derby Performance," in *ApacheCon*, 2005.
[36] T. Council, "Tpc-c," http://www.tpc.org/tpcc/.
[37] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP Through the Looking Glass, and What We Found There," in *Proc. SIGMOD*. New York, NY, USA: ACM, 2008, pp. 981–992.
[38] T.-F. Tsuei, A. N. Packer, and K.-T. Ko, "Database buffer size investigation for OLTP workloads," in *SIGMOD*. ACM, 1997, pp. 112–122.
[39] The Open Source Development Laboratory, "OSDL Database Test Suite," http://osdldbt.sourceforge.net/.
[40] H. Schöning, "The ADABAS Buffer Pool Manager," in *Proc. VLDB*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 675–679.
[41] M. Moir, "Laziness Pays! Using Lazy Synchronization Mechanisms to Improve Non-Blocking Constructions," *Distributed Computing*, vol. 14, no. 4, pp. 193–204, December 2001.
[42] M. Herlihy, B.-H. Lim, and N. Shavit, "Scalable Concurrent Counting," *ACM Trans. Comput. Syst.*, vol. 13, no. 4, pp. 343–364, November 1995.
[43] S. Moran, G. Taubenfeld, and I. Yadin, "Concurrent Counting," in *Proc. PODC*. New York, NY, USA: ACM, 1992, pp. 59–70.
[44] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software Transactional Memory: Why Is It Only a Research Toy?" *Commun. ACM*, vol. 51, no. 11, pp. 40–46, 2008.
[45] M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.
[46] J. Dybnis, "nbds," http://code.google.com/p/nbds/.
[47] D. Kegel, "The C10K problem," http://www.kegel.com/c10k.html.
[48] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, July 1990.
[49] ——, "Axioms for Concurrent Objects," in *POPL*, 1987, pp. 13–26.

APPENDIX

A. Linearizability Proof

We prove that our Nb-GCLOCK algorithm is linearizable [48]. According to [49], [26], the definition of linearizability is equivalent to the following:

- All function calls have a linearization point at some instant between their invocation and response.
- All functions appear to occur instantly at their linearization points, behaving as specified by the sequential definition.

Every execution path of *fixEntry* (see Fig. 3) has at least one linearization point. We chose the following linearizable points for each execution path returning through Lines 54, 70, 77, and 79, respectively:

- The *pin* operation for the returned Frame instance at Line 50,
- The *replace* operation at Line 68,
- The *pin* operation for the returned Frame instance at Line 67, and
- The *putIfAbsent* operation at Line 65.

All of the above methods are apparently linearized because each of them incurs at least one compare-and-swap or LL/SC.

***LEMMA** 1:* If $entry$ is null or already evicted in Line 50, then *pin* fails; otherwise, *pin* succeeds and the execution steps into Line 51.

***LEMMA** 2:* If $prevEntry$ does not exist in HASHTBL at Line 68, then *replace* fails; otherwise, *replace* succeeds and the execution steps into Line 69.

***LEMMA** 3:* If $prevEntry$ is not evicted in Line 67, then *pin* succeeds; otherwise, *pin* fails and the execution steps into Line 75.

***LEMMA** 4:* If and only if an entry associated with the key does not exist in Line 65, *putIfAbsent* returns null and the execution steps into Line 79.

Lemma 1, Lemma 2, Lemma 3, and Lemma 4 derive the following theorem:

***THEOREM** 4:* The *fixEntry* algorithm in Fig. 3 is linearizable.

B. Lock freedom

The lock-freedom property for Nb-GCLOCK means that a thread executing the *fixEntry* operation completes in a finite number of steps unless other threads are infinitely making progress. Conversely, Nb-GCLOCK is lock-free if each loop in fixEntry starves (until some value of the looping condition LC changes) only when other threads are continually completing their invocations with changing some value of LC. The processes that change LC must complete their invocations.

We give a succinct proof of the lock-freedom by extracting every possible loops in fixEntry and proving those loops are interfered with the progress only by other terminating processes. The following is the complete list of the possible loops derived by successive calls of the fixEntry method: Line 58-80 of Fig. 3 and Line 26-48 of Fig. 4, referring to as *loop-in-addEntry* and *loop-in-swap* respectively. We have the following theorems concerning the loops.

***THEOREM** 5: loop-in-addEntry* retries the loop only when another invocation of addEntry with the same *key* value was concurrently succeeded.

Proof: [Proof of Theorem 5] Another invocation of addEntry has succeeded when *loop-in-addEntry* enters Line 73 because the condition on Line 68 becomes false only when other invocation of addEntry succeeded. *loop-in-addEntry* always retries the loop through Line 73, and thus, it is proved that *loop-in-addEntry* retries the loop only when another invocation of addEntry with the same *key* value concurrently succeeded. ■

***THEOREM** 6: loop-in-swap* infinitely retries the loop only when other invocations of *fixEntry* are concurrently succeeding.

Proof: [Proof of Theorem 6] From Theorem 2 and the possible state transitions to *continue* in Fig. 5, it follows that *loop-in-swap* infinitely retries the loop only when other invocations of *fixEntry* are concurrently succeeding.[4] ■

The satisfaction of Theorem 5 and Theorem 6 derives the following theorem:

***THEOREM** 7:* The *fixEntry* algorithm in Fig. 3 is lock-free.

[4]A CAS success is a local success, whereas a CAS failure means another CAS succeeded; the state machine advances when the CAS is succeeded. It is notable that the nonblocking property of lock-freedom does not rule out starving in situations where it is explicitly intended (Line 40 of Fig. 4).