Efficient Lock Free Privatization

Yehuda Afek[1], Hillel Avni[1], Dave Dice[2], and Nir Shavit[1,2]

[1] Tel-Aviv University, Tel-Aviv 69978, Israel,
[2] Sun Labs at Oracle, 1 Network Drive, Burlington MA 01803-0903
hillel.avni@gmail.com

Abstract. Working on shared mutable data requires synchronization through barriers, locks or transactional memory mechanisms. To avoid this overhead a thread may privatize part of the data and work on it locally. By privatizing a data item a thread is guaranteed that it is the only one accessing this data, i.e., that it accesses the data item in exclusion.
The most robust and yet lock-free privatization algorithms, are lock-free reference counting (LFRC). These algorithms attach a counter to each node, which counts the number of references to the node. However, these counters are shared by all threads in the system and thus are contention prone, and must be updated with expensive atomic operations such as CAS.
We present a new privatization algorithm, Public Guard (PG); an algorithm which eliminates most of the contention of LFRC algorithms, while maintaining their robustness and non blocking nature. Our evaluation shows that PG improves performance by up to 50% in many work loads.
Another problematic issue with LFRC, that we address in this paper, is that a counter of a private node, may be accessed by a slow thread. This may prevent LFRC from freeing memory to the system. In another contribution of this paper we suggest a method with minimal overhead to allow LFRC to reclaim memory.

1 Introduction And Related Work

A privatization algorithm is a technique that allows the thread to make part of the data it accesses private, guaranteeing that no other thread can access it. In the literature, privatization is usually part of the memory management algorithm: threads privatize buffers before freeing them back into the allocatable memory pool.

However, in many situations it is important to provide privatization that is unrelated to memory management. For example, to allow threads to operate on chucks of data without the overhead of synchronization, or to allow them to move records between data structures. Thus, in its general form, we will say that

Definition 1. *Privatization is the process of thread π verifying that the data of an object O is not accessed nor accessible by any thread in the system except ξ.*

Meta data that is related to a private node may be accessed by the system. This meta data may reside inside or outside the private node.

We identify three privatization algorithms:

1. **Guards:** In this group each thread has a set of designated pointers (guards) that are pointing to all the nodes it is about to access. In privatization the algorithm scans all the guards to verify that no thread accesses the privatized node. Examples are Herlihy et al. PTB [1] and Michael hazard pointers [2].
2. **Epoch:** Here when a thread wants to privatize a node it deposits it in a to-be private group of nodes. Then the algorithm verifies that all the threads in the system passed through a code segment where they are guaranteed not to access any shared data. Then it frees all the nodes in the group. Hart EBR and QSBR [3] are members of this group. Dice et al. [4] created a transactional memory flavor of it.
3. **LFRC:** Count all references to each node. Both global, from the heap, and local from thread local stacks. When the count drops to zero, the node may be privatized. Representatives are Valois [5] and Detlef [6].

There are also hybrids of the above. Herlihy et al. [1] introduces SLFRC which combines guards with LFRC, and Gidenstam [7] mixes LFRC for global references with guards for local ones. Our new PG merges the robustness of LFRC with the thread designation of guards.

Privatization serves for two purposes; a thread which ends up privatizing a node n, can either free the node, or work on it exclusively, i.e., it serves both for privatization and for memory management.

Before we continue to the pros and cons of the above types, we make the following definition:

Definition 2. *Privatization is lock-free if a thread which does not accesses node n can not stop other threads from privatizing n.*

All epoch-based algorithms we encountered (e.g., [4, 3]) are blocking. I.e a thread that never exits its critical section prevents privatization across the system. So, while having extremely low overhead in the optimistic case, epoch-based may hang the system in the general case.

Guards are not scalable to arbitrary data structures, due to two reasons. First, there might be a lot of guards in the system making the process of privatizing awkward. The second is that in real applications it is very tricky to tell what guards are necessary.

LFRC algorithms are both lock-free and scalable. If a node is not used by a thread, that thread can not prevent the node privatization, which makes LFRC lock-free. It is scalable because no matter how many references exist in the system, the process of privatizing an item has the same overhead.

However, LFRC comes with a price:

1. It has memory overhead as a reference count is appended to each data item.
2. Updating the reference count requires a CAS, which is an expensive operation, that may fail in the presence of contention.

3. Unlike epoch-based and guard types, the meta data in LFRC is embedded in the node. Thus there is always a point where a thread must touch the node, through touching its meta data, before it knows that the node is private.

Our PG is an LFRC algorithm, but it replaces the simple integer reference counter with a structure of combined counters. The structure has an integer counter for global references, an array of local integer counters for local references, and a seqlock to facilitate the privatization process.

PG reduces the overhead of updating the reference count, for local references, by each thread counting independently in its own local counter. This reduction improves the performance as it prevents retries and eliminates slow atomic operations like CAS and F&A.

As mentioned above, privatization is used frequently in the context of memory management. There LFRC has a big advantage: it is automatic. A thread does not need to free a node explicitly, and when the total reference count is zero the node is freed. In real software, it can be difficult to tell from the application when a node can be freed, thus requirement of explicit freeing is likely to create both memory leaks and other bugs.

When we talk about memory management we actually talk about two different things:

1. **Recycling:** maintain a group of nodes, with the same size and same structure that can be used and reused but not freed to the OS.
2. **Reclamation:** Free memory to the OS so it can be used for any purpose.

When using LFRC privatization the application can recycle nodes but is not allowed to reclaim them. The reason is, that in a reclaimed node the reference count too may be reused for other purposes. When a thread reads it, not knowing that the node was privatized, it gets meaningless result or even a segmentation fault. One way to solve this problem is Herlihy [1] SLFRC, where they protect accesses to meta data with a guard. However, this method has a price per node that grows with the number of threads. In this paper we introduce an algorithm to reclaim nodes that are privatized with LFRC, with overhead that is amortized over multiple nodes. Although this algorithm uses epoch-based construction, we show the amount of unreclaimed memory is bounded, unlike all other pure epoch-based algorithms.

2 PG in a Nutshell

Here is the idea of the PG algorithm. In PG each node has three fields that are used for privatization:

1. `g_cnt` is used to count references from the heap to the node.
2. `l_cnt` is an array of integers with an entry for each thread in the system. Each thread uses its entry in the array to count references from its local stack to the node.

3. `inc` is a seqlock [8], which is locked when the node is private.

PG has a separate method and API for updating global and local references. When a reference from the heap is added, a thread must use CAS to update g_cnt. However, in the much more common case, when a thread updates a local reference count, it just uses regular reads and writes on its entry in l_cnt, without any possible contention.

The l_cnt access pattern is similar to TLRW-bytelock's [9], and reuses its slot allocation algorithm (though we did not find it beneficial to use byte size counters).

If, upon decrementing any counter, a thread finds that the global count equals zero, it automatically tries to privatize the node (either because it tries to privatize the node or for memory management).

Privatization has the following steps after finding the global count equals zero:

1. `inc` is sampled and, if locked, the node is already private, so done.
2. if `g_cnt` $\neq 0$, done.
3. `l_cnt` is scanned and, if any entry is $\neq 0$, done.
4. Try to lock `inc` by incrementing it with a CAS from sampled value, if failed done.
5. Privatization has succeeded.

Upon completing the above sequence, exactly one thread succeeds in privatizing the node, which makes the node actually private (assuming all thread well behave and follow the rules as below).

PG works under the assumption that threads are well behaved. This implies three rules that all threads must follow:

1. Before accessing any node, protect it by incrementing its associated local reference count, and when done decrement that counter.
2. When adding or removing a global reference, update the global counter with the corresponding API.
3. Global count in PG serves also as a flag. If a thread does not want to privatize a node, nor to release it then it should verify that global counter $\neq 0$.

If a thread wants to delete a node n from a data structure and then work on it not in private, or insert it to another structure, it must verify g_cnt $\neq 0$. This can be done by calling `PgInc(n)` before deletion and `PgDec(n)` after insertion.

Figure 1 demonstrates a situation where two processors want to access the same object at the same time. In LFRC both need to update the same address, i.e. the reference count, before and after the access, thus they need to use CAS, which is resource consuming and may fail. In PG, each processor updates its relative `l_cnt` entry with a simple write, an operation that is cheaper and uninterrupted.

In summary, accessing an object for read or write is generally far more common than handling it for memory management. LFRC involves a CAS operation

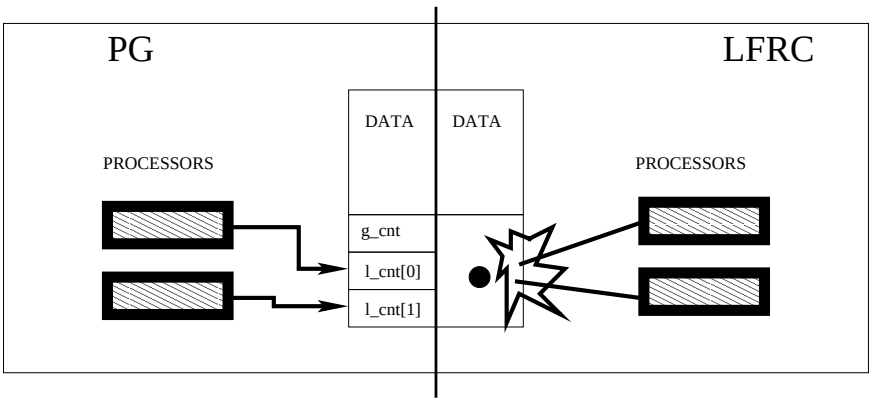

Fig. 1. PG vs. LFRC behavior for simple access.

for each such access, which PG manages to avoid. This is the root of PG better performance.

3 PG Privatization Algorithm

In this section we call Valois [5] algorithm simply LFRC, and let it represent all previous LFRC algorithms. To let the reader see where PG is better than LFRC, we explain LFRC and PG together.

Then we show how to let both LFRC and PG reclaim memory which they privatized, with minimal and amortizable overhead.

3.1 Overview of PG and LFRC

In Figure 2 we see the mandatory fields of a node which is used with LFRC or PG. LFRC has one field which holds both the total reference count and a claim bit, which is an indication to the status of the node, i.e., whether it is private. In PG this field is broken into three separate fields. Local counter (l_cnt) per thread for local references, a global counter (g_cnt) for global references and incarnation (inc). Incarnation is a seqlock, i.e., a combination of a counter and a lock. When incarnation is locked (= odd value), the node is private. When a node is shared again, incarnation field is unlocked (= even value) and incremented in one operation.

Decrement Reference Count and Free When LFRC or PG remove a local or global reference, they decrement the reference count. LFRC has one reference count and PG maintains one for global references and an array of reference counters per thread in each node for local references. When reference count, or in PG, when global reference count, drops to zero the algorithm tries to privatize the node. That is why this section explains decrement and privatization together.

Field	LFRC	PG	Description
data	V	V	Application dependant.
links	V	V	References to other nodes.
p_cnt	V	X	Counts references and includes an indication the node is private.
l_cnt	X	V	An array of local reference counters per thread.
g_cnt	X	V	A counter of global references.
inc	X	V	Incarnation: a seqlock, that is used in the privatization process.

Fig. 2. LFRC and PG node structures

```
int LfrcDecrementAndTAS(int *cnt)
LD01:    repeat
LD02:        old = *cnt
LD03:        new = old - 2
LD04:        if(new == 0)
LD05:            new = 1
LD06:    until (CAS(cnt, old,new) == TRUE)
LD07:    return ((old - new) & 1)
```

Fig. 3. LFRC decrement reference count of a node

Figure 3 shows the actual decrement of the reference count of a node in LFRC. If the reference count drops to zero the LFRC tries to privatize it by setting the claim bit. Both decrementing and privatizing are done with a CAS in line LD06 of `LfrcDecrementAndTAS`. `LfrcDecrementAndTAS` is called from `LfrcRelease` in Figure 4.

PG uses two functions to perform reference count decrement. `PgDoneAccess` in Figure 5 decrements the local counter by a simple write. As most reference are local, and as LFRC uses CAS to decrement local count, this is a place where PG avoids many CAS. In Figure 6 `PgDec` uses CAS to decrement the global reference count of a node.

Both functions are calling `PgTryPrivatize` from Figure 7 which checks (PT05 and LD04) that global count and local count (PT06-PT08) are zero and only than tries to privatize by locking the incarnation field of the node with a CAS (PT09). Thus privatizing a node both in LFRC and PG involves a CAS, and incurs similar overheads.

Safe Read An application that wants to access a node, must read its pointer with a safe read. This operation increments the reference count of the node and then verifies the node is still pointed from where it was pointed before.

In line LS05 of Figure 8, `LfrcSafeRead` uses `AtomicAdd` to increment a node reference count.

`PgSafeRead` from Figure 9 uses a simple increment function (PS05) to add a local reference. As safe read is a very frequent operation this simpler command saves work for `PgSafeRead`.

```
void LfrcRelease(node *n)
LR01:    if(n==NULL)
LR02:        return
LR03:    if (LfrcDecrementAndTAS(&n->p_cnt)==0)
LR04:        return
LR05:    foreach (L in n.links)
LR06:        LfrcRelease(L)
LR07:    Privatize (n)
```

Fig. 4. LfrcRelease privatizes a node and decrements all its links reference counts

```
void PgDoneAccess(node *n)
PD01:    If(n==NULL)
PD02:        return
PD03:    decrement(n->l_cnt[SELF_ID])
PD04:    PgTryPrivatize(n)
```

Fig. 5. PG remove a local reference

New Node Allocation The allocation operation returns a node from a free nodes pool, which might be global or local and hands it to the application. If the pool is global the algorithms must be aware of ABA risks, which makes the function less trivial. In this section we focus only on allocating from a global pool.

In line LN02 in Figure 10, LfrcNew reads the first object in the free list with a SafeRead. The reason is that after p was acquired with a `SafeRead`, p→next can not change as long as it is in the free pool, and p can not be freed again after it was allocated. If p was read without `LfrcSafeRead / PgSafeRead`, it could have been allocated and freed again and have a new `p→next` and now the old `p→next` might be pointing to a currently allocated node. This would cause a double allocation which is erroneous.

Here is the scenario, in a more formal way, how threads T_1 and T_2 allocate a node twice if not using `SafeRead`:

1. T_1 reads p from freelist and p_{T_1}=p→next.
2. T_2 allocates p .
3. T_2 allocates p_{T_1}.
4. T_2 frees p again with a new p_{T_2}=p→next.
5. T_1 arrives at line LN05 of LfrcNew and replaces freelist with p_{T_1} which has been allocated already by T_2.
6. T_2 allocates p_{T_1} again.

In line LR07 of LfrcRelease function and line PT11 of PgTryPrivatize a node may be recycled into a global shared pool or privatized, i.e., accessed without synchronization by the thread who holds a reference to it. As we recall in LFRC New had to call `SafeRead` so the node will not be freed again during a critical

```
void PgDec(node *n)
PC01:    if(n==NULL)
PC02:        return
PC03:    while (!CAS(n->g_cnt, n->g_cnt, (n->g_cnt-1)))
PC04:    PgTryPrivatize(n)
```

Fig. 6. PG PgDec removes a global reference and then tries to privatize the node.

```
void PgTryPrivatize(node *n)
PT01:    retry:
PT02:        cur = n->inc
PT03:        if locked(cur)
PT04:            return
PT05:        if (n->g_cnt == 0)
PT06:            for(id = 1...MAXID)
PT07:                if(n->l_cnt[id])
PT08:                    return
PT09:            if(CAS(&n->inc, cur, cur + 1))
PT10:                tmp = n
PT11:                Privatize n
PT12:                foreach l in tmp->links
PT13:                    PgDec(l)
```

Fig. 7. PG Try to privatize the node.

part of the function. In PG this is more complicated as a thread checks both g_cnt and l_cnt and incarnation before freeing a buffer so when extracting n from a free pool `n→inc` must be locked, at least one of `n→l_cnt` entries and `n→g_cnt` must be non zero.

Figure 11 is the `PgNew` which allocates from a global shared pool.

The function is using `PgSafeRead` in line PN02 to read from the free list. Then, in line PN03 it increments the global count as well. If it manages to extract the node in PN06 it returns it decrements the local count in PN08 and returns the new node in PN09. Otherwise it decrements the node counters in PN10-PN11 and retries.

3.2 PG and LFRC usage for Reclamation

As explained in section 1 LFRC and PG can be used to recycle memory but not for reclamation (i.e., recycling but not reclamation). The following algorithm maintain the lock-free property of PG and LFRC, but gives them the ability to reclaim memory. To reclaim memory we need to know if there is a sleeping thread that is about to access a reference count of a free node. This can happen only in the `SafeRead` function, before checking if a node is still pointed from the same place. In order to reclaim, the system must know there is no thread in that section, so we make it as swift as possible and wrap it with a local seqlock

```
node *LfrcSafeRead(node **n)
LS01:    forever
LS02:        q = *n
LS03:        if (q == NULL)
LS04:            return NULL
LS05:        AtomicAdd(q->p_cnt, 2)
LS06:        if(q == *n)
LS07:            return q
LS08:        LfrcRelease(q)
```

Fig. 8. LFRC Safe read function

```
node *PgSafeRead(node **n)
PS01:    forever
PS02:        q = *n
PS03:        if (q == NULL)
PS04:            return NULL
PS05:        (q->l_cnt[ID])++
PS06:        if(q == *n)
PS07:            return q
PS08:        PgDoneAccess(q)
```

Fig. 9. PG Safe read function

(rec_epo]) which is incremented when we enter or exit that section. We also add the field `rec_epo`, as shown in Figure 12, in LFRC and PG nodes.

When a thread has too many nodes in its local pool it scans all the seqlocks and waits for all threads to get out of that section. For this solution we modify `SafeRead` as depicted in Figure 13. In SR05, just before incrementing the reference count, the local rec_epo seqlock is incremented to locked state. After increment completion (SR10) or failure (SR08), rec_epo is incremented locally to unlocked state. Our tests show this method has literally the same performance as `SafeRead` that does not allow reclamation.

The spirit of this technique is like Herlihy's which uses a guard for that critical section, but the same absolute overhead which they have per node is here per all freed nodes.

In order to verify memory consumption is limited the free function counts the number, N, of nodes it freed locally. If N equals a threshold H, the thread takes a snapshot S_1 of all critical counts. If N>H the node is freed to a global pool and another snapshot S_2 is taken. If all threads which were in the critical section in S_1 made progress in S_2, all the local pool is reclaimed. S_2 is created only once, and if there is a thread which made no progress the scan stops and continues from where it stopped in the next free operation.

```
node *LfrcNew()
LN01:    forever
LN02:        p = SafeRead(&free_list)
LN03:        if (p == NULL)
LN04:            Assert(Out of memory)
LN05:        if(CAS(&free_list, p, p->next) == TRUE)
LN06:            ClearLowestBit(&p->p_cnt)
LN07:            return p
LN08:        LfrcRelease(p)
```

Fig. 10. LFRC allocation from a global pool

```
node *PgNew()
PN01:    forever
PN02:        q = PgSafeRead(&free_list)
PN03:        PgInc(q)
PN04:        if(p == NULL)
PN05:            Assert(Out of memory)
PN06:        if(CAS(&free_list, p, p->next) == TRUE)
PN07:            increment(p->inc)
PN08:            PgDoneAccess(p)
PN09:            return p
PN10:        PgDoneAccess(p)
PN11:        PgDec(p)
```

Fig. 11. PG allocation from a global pool

Lemma 1. *Assume J is the number of threads in the system, H is the threshold and M is maximal number of simultaneous used nodes, maximal unreclaimed nodes is less than M+((H+1)*(J-1))*

Proof. T allocates a node if its local pool and the global pool are empty. So the maximal number each other thread may hold is H+1, i.e., threshold and another one it privatized but did not free into global pool. So we have J-1 threads which hold H+1 nodes each and one which holds 0 nodes.

4 Evaluation

We tested our algorithms on 8 processors Intel Core i7 Processor I7-920, running 64-bit RedHat Linux.

Field	LFRC	PG	Description
rec_epo	V	V	Reclamation epoch: a seqlock, that is used for reclamation.

Fig. 12. LFRC and PG with reclamation

```
node *SafeRead(node **n)
SR01:    forever
SR02:        q = *n
SR03:        if (q == NULL)
SR04:            return NULL
SR05:        Lock rec_epo
SR06:        increment reference_count
SR07:        if(q == *n)
SR08:            Unlock rec_epo
SR09:            return q
SR10:        Unlock rec_epo
SR11:        decrement reference count
```

Fig. 13. Generic safe read function with reclamation for LFRC and PG

The graphs demonstrate the performance of two algorithms, PG and LFRC [5]. Both algorithms are run with local buffer pools (PGL and LFRCL) and with global buffer pool (PGG and LFRCG).

For each configuration/test (each in a separate figure) we generate three graphs. The first is the average time in nano-seconds taken per one operation, over 5 runs. The second is the average number of CAS operations during that time, which explains the overhead of LFRC. The third is the average number of times in the operation that it tries to privatize the node i.e., the number of times PG was in line PT09 of `PgTryPrivatize` or the number of times LFRC was in line LD06 of `LfrcDecrementAndTAS`. This graph is the overhead created by PG. PG tries to free a node when the global references drop to zero, but it is not aware of local references. Thus it does more retries than LFRC which slows its operation.

The operations are on a list where insert and delete operations take a lock, while search operations are lock-free. In this test a lock-free search manipulates local references proportionally to the length of the list, but an insert or delete do exactly one local and one global reference. We show three tests:

1. Figure 14 demonstrates various thread numbers with 1K elements and 10% updates. Here, the amount of CAS is proportional to the number of operations and thus PG is better by a constant coefficient. The number of free retries is higher in PG but its absolute number is too small too make a difference.
   For this test we added, in figure 15 cache misses as counted by VTune^{TM} Performance Analyzer 9.1 for Linux. Although PG is better than LFRC for this work load on all thread counts, it has either equal or slightly more cache misses. This makes sense as everywhere LFRC gets a miss PG suffers one as well, while PG touches more memory locations.
2. In figure 16 there are 8 threads with 10 elements and various update rates. Up to about 80% updates we can see PG with local pools is best, and there LFRC with local pools passes it. The reason is that as the number of PG

freeing tries grows and overshadows LFRC CAS number, which drops as the number of searches goes down.

3. Figure 17 shows 8 threads with 10% updates and various elements number. We see in 10% updates, the number of CAS, which in LFRC is proportional to the duration of the transaction, makes PG always better. The number of free tries is very small except in very small lists where PG is not better than LFRC.

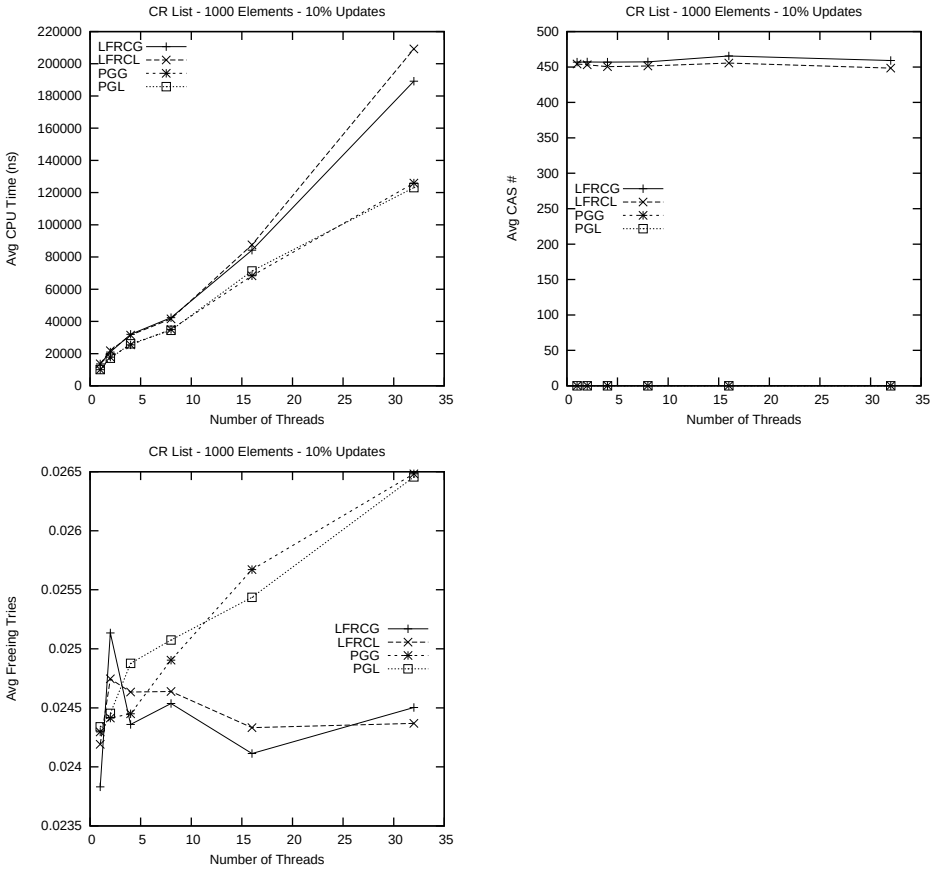

Fig. 14. Average time vs. average CAS on various threads number.

5 Conclusion

In this paper we introduced PG, the first LFRC which does not use CAS [5] nor HP [7] for local references. We showed it has superior performance compared with previous LFRC although it has slightly more cache misses. This fact

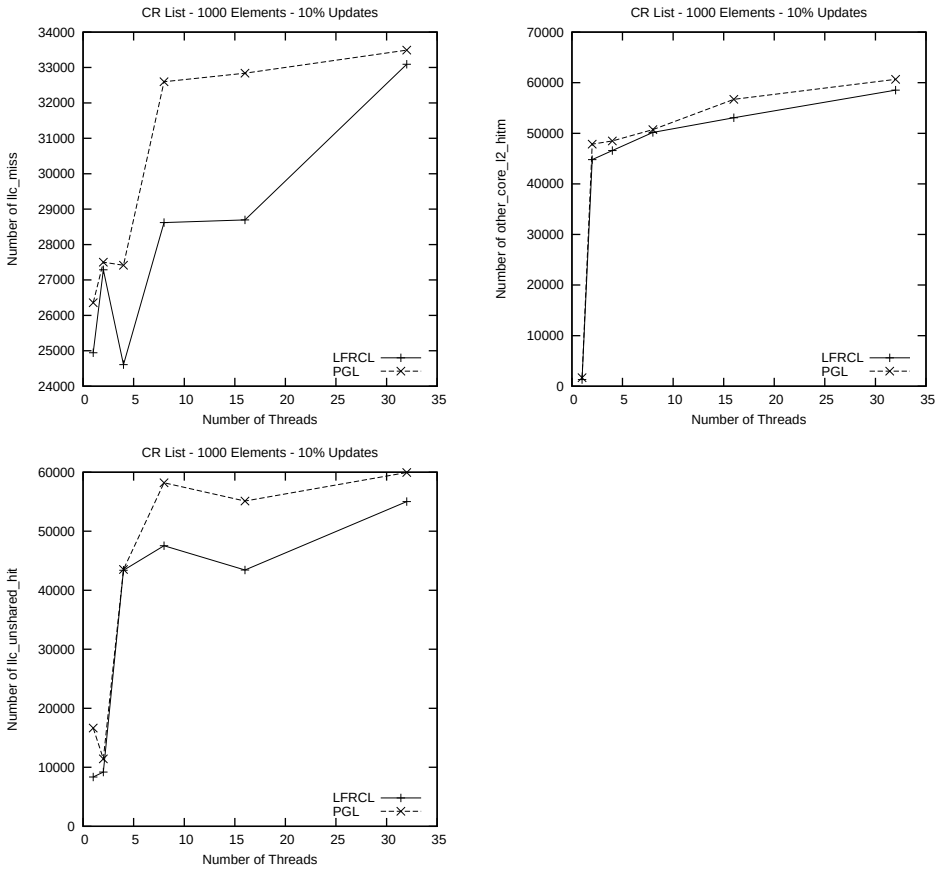

Fig. 15. Number of cache misses various threads number.

emphasizes that PG advantage is in the algorithmic reduction in the number of CAS operations. PG outperforms LFRC in all workloads that are not dominated by allocation and free operations.

Another contribution is a method to allow LFRC and PG free memory to the system, and bound the maximal amount of allocated memory. This is the first time LFRC can free memory without HP[7, 1] that has significant overhead or DCAS [6] which is not common in today hardware.

6 Acknowledgements

This paper was supported by European Union grant FP7-ICT-2007-1 (project VELOX).

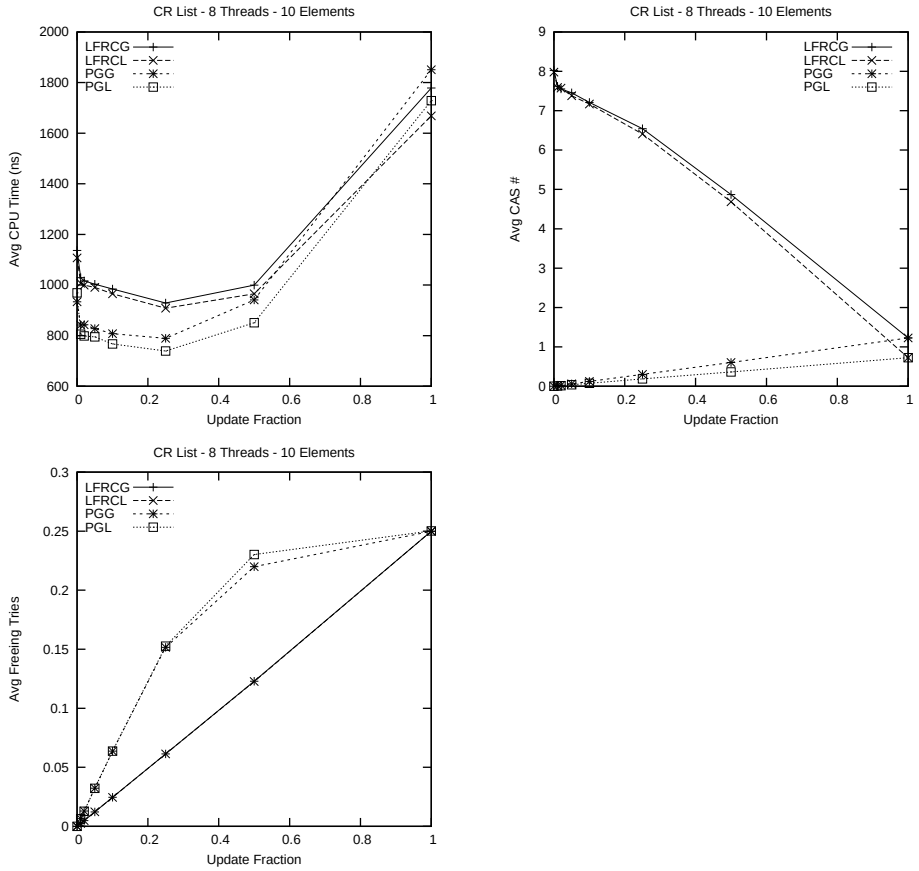

Fig. 16. Average time vs. average CAS on various update rates.

References

1. Herlihy, M., Luchangco, V., Martin, P., Moir, M.: Nonblocking memory management support for dynamic-sized data structures. ACM Trans. Comput. Syst **23** (2005) 2005
2. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6) (2004) 491–504
3. Hart, T., McKenney, P., Brown, A.: Making lockless synchronization fast: performance implications of memory reclamation. Parallel and Distributed Processing Symposium, International **0** (2006) 4
4. D. Dice, A.M., Shavit., N.: Implicit privatization using private transactions. In: TRANSACT. (2010)
5. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (1995) 214–222

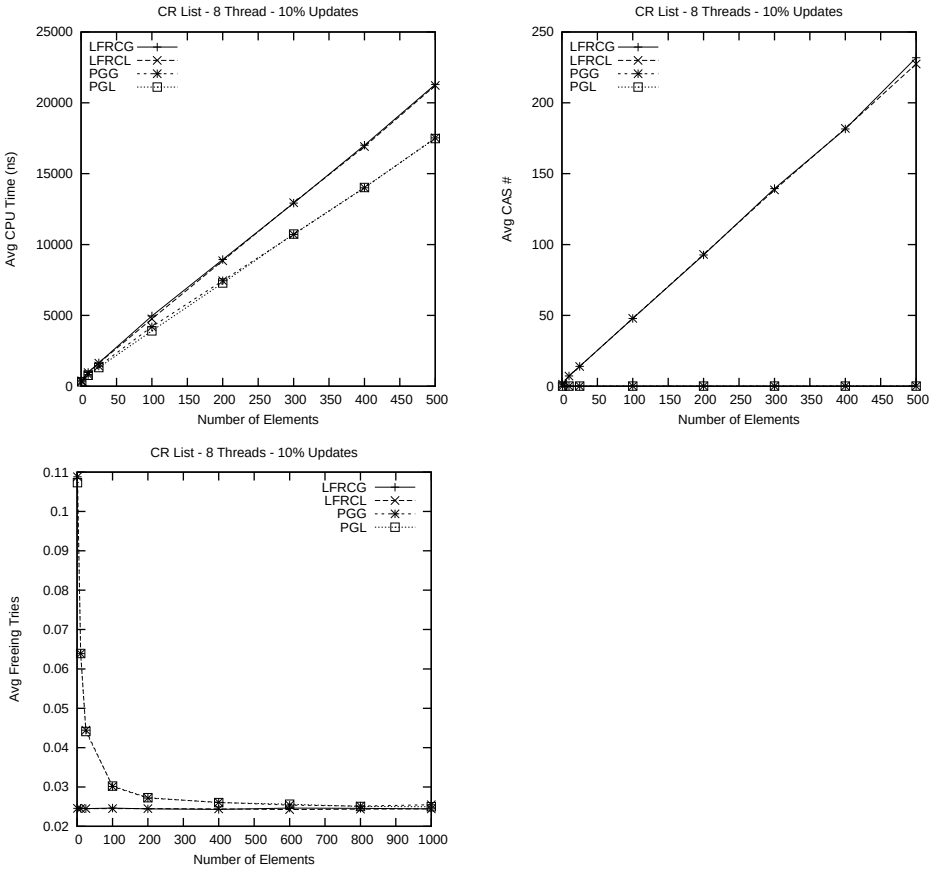

Fig. 17. Average time vs. average CAS on various elements number.

6. Detlefs, D.L., Martin, P.A., Moir, M., Steele, Jr., G.L.: Lock-free reference counting. In: PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (2001) 190–199
7. Gidenstam, A., Papatriantafilou, M., Sundell, H., Tsigas, P.: Efficient and reliable lock-free memory reclamation based on reference counting. IEEE Trans. Parallel Distrib. Syst. **20**(8) (2009) 1173–1187
8. Lameter, C.: Effective synchronization on linux/numa systems. In: Gelato Federation Meeting. (2005)
9. Dice, D., Shavit, N.: Tlrw: return of the read-write lock. In: In Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing. (2009)