

# First Experiences on PWCS synchronized Data Structures\*

**Benjamin Engel and Marcus Völz**

Technische Universität Dresden  
Operating-Systems Group

Nöthnitzer Strasse 46, Dresden, Germany  
{engel, voelp}@os.inf.tu-dresden.de

## Abstract

During last year’s RTLWS, Nicholas McGuire introduced a new mechanism for exploiting the low-level nondeterminism of modern many-core architectures to synchronize objects: probabilistic write copy select (PWCS). In this paper, we report on our first experiences in synchronizing four standard data structures with PWCS: static arrays, hash tables, dynamic lists and trees. Comparing fence-based and hash-based PWCS against more traditional locking schemes we found that, despite its current limitation to a single writer, PWCS is a promising candidate for further exploration in the vast design space of manycore synchronization mechanisms.

## 1 Introduction

In his talk during last year’s RT-Linux Workshop, Nicholas McGuire sketched a completely new synchronization primitive together with a revolutionary idea to cope with race conditions. Rather than mitigating races by all means, he proposed to exploit the increasing complexity of modern hardware and the implied decreasing likelihood of permanent races to avoid costly atomic operations in traditional synchronization primitives. The design principle behind Probabilistic Write Copy Select (PWCS) [6] is to make object inconsistencies detectable and to exploit the complexity-induced nondeterminism of modern hardware architectures to quickly leave situations where races render objects inconsistent. PWCS achieves this by first invalidating a consistency tag before the actual data is written. Once the modification is complete, a write to a second consistency tag re-validates the object. In the mean time, readers will find mismatching tags and retry the operation or skip to another replica of the object. Replacing

tags with hashes, the dependency on cache coherent hardware and memory fences can be relaxed even further, in particular, if hardware filters avoid updates with older data [7].

In this paper, we investigate the general feasibility of PWCS for synchronizing data structures such as arrays, hash tables, lists and trees. On the one hand, the links that connect the individual items of these data structures are small enough to allow for perfect hashes through pointer replication. On the other hand, extending PWCS to updates that are no longer in place aggravates the synchronization problem. We found that although PWCS is currently limited to a single writer (next to arbitrary many concurrent readers) synchronizing linked data structure reads with PWCS typically outperforms coarse and fine-grain lock-based approaches while maintaining high success rates for the first read attempt. At the same time, we saw difficulties in recovering from concurrent writes during long linked list traversals. An in depth discussion of traversal algorithms is left for future work.

---

\*This work is in part funded by the DFG through the projects “QuaOS”, the CRC 912 “HAEC” and through the excellence initiative “center for Advanced Electronics Dresden (cfAED)” and by the state Saxony and the European Union through the ESF young researcher grant IMData.

Next, we give a more detailed introduction into PWCS and clarify the assumptions on which this work builds. Sections 3ff. discuss our general handling of in-place modifications (i.e., data-updates in arrays), lists (Sec. 4), hash tables (Sec. 5) and binary trees (Sec. 6) and presents our experimental results. Section 7 relates our work to the works of others. Section 8 concludes.

## 2 PWCS

Classical synchronization primitives such as locks or transactions and operations on lock-free data structures seek to prevent observable inconsistencies at all costs. For instance, locks inhibit readers and other writers from accessing a lock-protected object while this object is under modification. Lock-free data structures and transactions operate on private copies (e.g., in the cache), which they hook in after all inconsistencies are resolved.

In contrast, PWCS explicitly allows objects to become inconsistent as long as this inconsistency can be detected by all concurrent readers. All elements of the array are protected by a single global pair of consistency tags. Readers of array elements copy first the begin tag and then the required data into a local buffer. After that, they compare the stored end tag with the local copy of the begin tag. A match of the version numbers of which these tags are comprised reveals whether the data is consistent or whether a writer has modified this data in the mean time. The latter is achieved by writers working in the opposite direction. That is, a writer first invalidates data consistency by increasing the end tag, then modifies the data and re-validates the object again by also increasing the begin tag. On processors with a weak load and store ordering, fences are required between the reads of the tags and the data and between updating the tags and modifying the data.

```
index = random % ARRAYSIZE;
Item item;
foreach (r : Replica) {
    tag_begin = r.tag_begin;
    l_fence();
    memcpy (item, r[index].item);
    l_fence();
    tag_end = r.tag_end;
    if (tag_begin == tag_end) break;
}
```

**FIGURE 1:** *PWCS Protected Array : Reader*

```
index = random % ARRAYSIZE;
Item item;
foreach (r : Replica) {
    r.tag_end++;
    s_fence();
    memcpy (r[index].item, item);
    s_fence();
    r.tag_begin++;
}
```

**FIGURE 2:** *PWCS Protected Array : Writer*

Adding consistency tags locally to each array element results in a more fine-grain synchronization pattern. However, like with locks, this fine-grain pattern comes at the cost of larger data structures and more synchronization operations. In the case of tag-protected PWCS structures, these are the fences before and after accessing the data. In addition, we have to require that the underlying communication medium respects tags in that the begin-tag modification will be visible before data modifications and that data modifications complete before the modified end-tag becomes visible. Figure 1 and Figure 2 show the pseudo code of a PWCS protected array with fine grained object tags.

A second dimension in the design space of PWCS is the use of tags versus hashes. Hash-based consistency tags lift the remaining dependencies on the order in which data and tags are written. However, hashes are in general prone to collisions and expensive to calculate when only small parts of large objects change.

A third dimension in the PWCS design space is the number of replica of the protected object. Increasing this number reduces the likelihood of finding all replica in an inconsistent state, in particular if readers and writers work in opposite directions or if replica are chosen randomly for reading. Alternatively readers may repeatedly try to obtain a consistent version of the very same object, thereby reducing the numbers of replica to one, but decreasing performance as well (higher reader-writer-contention on one object).

In this paper we report on our experiences in synchronizing four data structures with PWCS: arrays, lists, hash tables and binary trees. We compare PWCS synchronization against reader-writer locks, both node granular and at the coarser granularity of one lock per data structure. We discuss the data structures in the above order and present experimental results. Measurements were taken on a 4 socket 16 core Bulldozer system (AMD Opteron 6274) running at 2.2 GHz and on a Core i7/920 (Nehalem, 4 cores, 2 hyperthreads) at 2.66 GHz.

Our ultimate goal is to prepare PWCS for non-coherent architectures with weak memory models. We will therefore make the following assumptions on the available hardware. Implementations for concrete architectures with stronger memory ordering may omit some of the fences and other protecting measures.

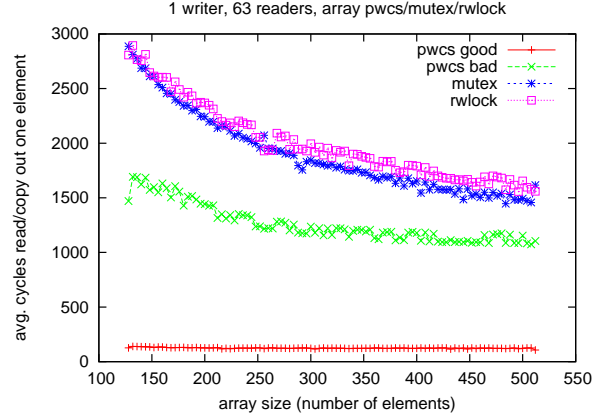
1. Reads and writes of small amounts of data (up to the size of a pointer) are atomic and complete. That is, reads of such a part of the object may return any value that was written but no other values.
2. Writes will eventually propagate to all readers. The order in which writes may arrive at readers may differ. In particular, it may happen that one reader receives the first part of an object whereas a second reader receives the last part and that then the respective missing parts arrive in reverse order or only after a second update was received.
3. Fences have the usual semantics on loads and stores. That is, stores separated by a store fence (`sfence`) are read in such a way that stores before the fence become visible before stores after the fence. A possible implementation of such a fence is to wait for the maximum transmission time before issuing the subsequent stores.

Assumption 1 is required to ensure that pointers and tags always refer to one of the addresses that have been written. In particular, if writes are constrained to objects of a certain type, no pointer will refer to the middle of such an object. Assumption 2 allows for arbitrary communication networks between the individual cores of the system. In particular, it allows updates between two cores to be reordered during the transmission, lending more flexibility for on-chip or off-chip routing.

### 3 Arrays

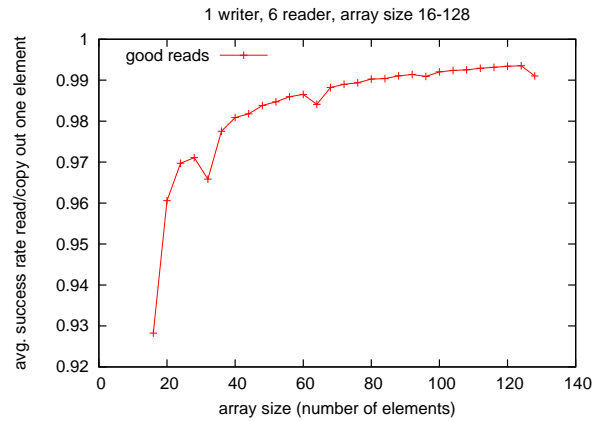
Our array experiments evaluate PWCS' capabilities to protect data against in-place updates. The results here are primarily to confirm McGuire's findings in our setup. For global consistency tags, we augment the entire array with a header. Object granular consistency tags augment all elements with such a header. The payload is 64 bytes, the size of the array ranges from 128 to 512 elements. In this exper-

iment one writer modifies the data structure and 1–63 readers read parts of it concurrently. Depending on the evaluated protection scheme, the information stored in the header is one of the following: a mutex, a rwlock<sup>1</sup>, a pair of consistency tags, or a hash value.



**FIGURE 3:** Read performance of up to 63 readers copying 64 byte payload, protected by PWCS tags or locks

Figure 3 shows the average read performance of a PWCS-protected array, compared to mutex or rwlock synchronization. In the implementation based on mutexes and rwlocks, the readers have to write the lock variable atomically in order to acquire a lock, thus the smaller the array is, the more expensive this operation will be, since contention increases with decreasing number of objects to lock. In case of PWCS-based locking, only writers have to modify the header, while readers might detect inconsistent versions and have to retry, either using another replica or the very same object.



**FIGURE 4:** Chance reading a single object returns a consistent copy

<sup>1</sup>We use the standard Linux `pthread_mutex_t` or `pthread_rwlock_t`.

Figure 4 shows the average success rate of reading a single object, depending on the size of the array. With a relatively small array containing 16 objects, readers have a chance above 90% to get a consistent copy. In Figure 3 two cases are plotted, the 'good' one, where the first read results in a consistent copy and the 'bad' one, where the reader has to retry until a consistent version is found. As one can see, the former case costs about 120 cycles, whereas the latter results in approximately 1400 cycles. It occurs with decreasing probability, which explains why in average the read performance approaches the 'good' case.

## 4 Lists

The two fundamental differences between PWCS-protected arrays and linked data structures are:

1. individual links are typically small enough to be read or written atomically, and
2. the elements of the data structure are no longer modified in place but allocated, de-allocated or re-linked to a different data structure of the same type.

Clearly, due to the implied traversal of the entire list, global hashes are inconvenient for protecting linked data structures. Otherwise, global consistency tags for lists and trees match the construction for arrays. That is, writers first invalidate the global end tag, modify the link structure and re-validate the data structure by writing the begin tag. Since operations on the payload are often much slower than modifications on the links of a single element, it is convenient to protect links and payload data separately. For our tag-based list versions, we have therefore chosen list element headers with separate begin and end tags for the prev and next pointers and for the payload stored in the element. Likewise, we use a hash for the pointers that does not cover the separately protected payload. It suggests itself to use a replica of the pointers as a perfect, i.e., collision-free hash. In addition, we need some means to indicate whether the element still belongs to the same list and that it is still located at the same position in the list. This is because writers may dequeue the very list element a reader is currently working on. In particular, no writer must de-allocate a list element that is currently used by readers. In this respect, PWCS protected linked data structures require the same type-safe memories as lock-free algorithms. RCU [3] is one version to establish this type safety guarantee. Notice however, in comparison to RCU protected lists,

we need to defer only the garbage collection of typed list elements into the pool of untyped memory. As long as list elements preserve their type, no operation has to be deferred to the end of the next grace period.

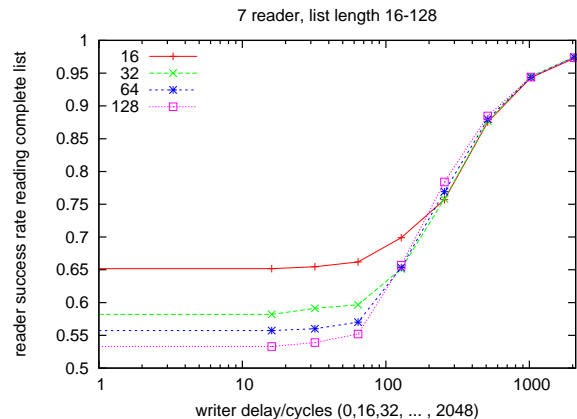
For our per-node hash-based lists, we exploit the fact that normal user-level pointers should not refer into kernel space and vice versa and that all our elements are 64-byte aligned to store an 8-bit version number in the two most and six least significant bits of the pointer replica.

```
Object * obj = list_head;
tag_begin = obj->tag_begin;
while (!found) {
    Object * next = object->next;
    tab_begin_next = next->tag_begin;
    tag_end = object->tag_end;
    if (tag_begin == tag_end) {
        object = next;
        tag_begin = tab_begin_next;
    } else {
        // error
    }
}
```

**FIGURE 5:** *PWCS Protected List*

Figure 5 shows the pseudo code for traversing and modifying a single-linked list. Double-linked lists work accordingly.

For our experimental setup, we have pre-allocated pointer-to-list elements in an array to allow random access by the writer. The nodes are then connected in a random fashion. Operating concurrently with the 1 – 7 readers, which traverse the list from the beginning to the end, a writer picks one item randomly, dequeues it and enqueues a new one at another position in the list.

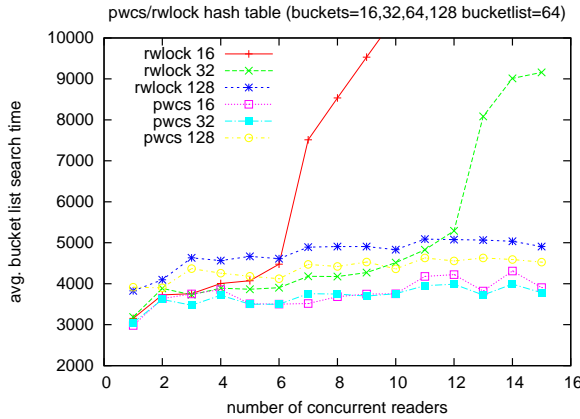


**FIGURE 6:** *Success rate traversing the complete list without encountering an inconsistent node*

Figure 6 shows the average chance a single reader has to traverse the complete list without hitting a node the writer is concurrently en- or dequeuing. In contrast to reading from an array, where the position of an object is fixed in memory, a reader cannot easily recover when reading a broken node, since the previous node might have been changed in the mean time too. The length of the list and the number of concurrent readers has no measurable influence on the traversal probability. To increase the chance a reader has we artificially reduced the write speed by adding a delay after every write operation. This reflects the observation that the higher the modification rate is, the smaller is the success of reading the whole list.

## 5 Hash Table

Our hash table implementation combines in-place modifications of the array and the modification of dynamically allocated elements in its collision chain. To mimic a balanced hash table, writers remove an element from the collision chain of one randomly picked bucket and insert a new element into the same bucket. Readers traverse through the chain of one randomly picked bucket to search for a specific element. The hash table varies in size (between 16 and 128 buckets), whereas the collision chain length is constantly at 64 nodes per bucket. To prevent readers from hitting an invalid node, we use list-granular locks instead of node locks, the writer locks the head of the bucket list, while PWCS-readers check the consistency tags before and after traversing the complete list.

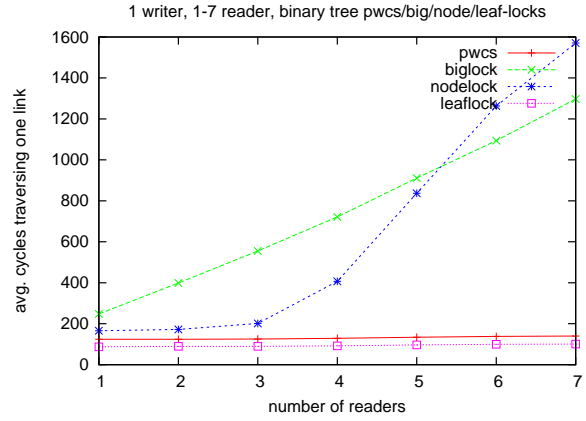


**FIGURE 7:** Hash table performance, comparing PWCS- and rwlock-protected bucket lists

Figure 7 shows the performance results of our

hash implementation. Depending on the number of buckets, the contention quickly increases in the rwlock-based hash table, resulting in longer searches when the number of readers gets close to half the number of buckets. PWCS shows a slight increase, but stays below its rwlock-based node lock counterpart and does not suffer from such an impact.

## 6 Binary Tree



**FIGURE 8:** Link traversal times

Figures 8 show our final results on PWCS protected binary trees. Readers traverse the binary tree from the root to the leaves. Writers are confined to modify leaf nodes only by inserting new nodes at the leaves, tree rotations are left for future work. We compare PWCS locks to a big tree lock, node and leaf locks. The implementation which takes only a lock on the leaf nodes is slightly faster, but not perfectly comparable, since modifications on the path are not detected, although they do not occur in this experiment, whereas the PWCS variant checks all the tags from the root node down to the leaf nodes, resulting in a higher read side overhead. Taking a lock on the whole tree as well as taking node locks all the way down from the root to the leaves expectedly shows high performance impacts.

## 7 Related Work

Naturally, this work stands in a strong relationship with classical and modern synchronization primitives. Classical lock-based approaches inherently need atomic operations to modify a lock variable after invalidating all copies in the local caches. Usually, this requires broadcasting an invalidate message or complex snoop filters to track the location

of active copies plus a read to attain consensus on the next lock holder. Queue locks such as MCS [2] try to reach this consensus upfront by ordering requesting threads in a list. However, the list enqueue still requires an atomic operation with the invalidate broadcast that it implies and withdrawing from the list further complicates its implementation [1]. Lock-free data structures (c.f. [4, 5]) avoid synchronization at the cost of also requiring type stable memory and complex atomic operations to atomically switch between consistent states. Dedicated hardware, such as transactional memory or dedicated lock networks, help reduce the programmer burden. However, they are either based on coherent caches or require a significant amount of space in addition to the on-chip or off-chip connections between cores.

Probabilistic algorithms and methods such as Monte Carlo and Las Vegas are well known, PWCS has similarities depending how it is implemented (multiple replica vs. multiple retries). However, with the exception of McGuire’s pioneering work on PWCS [6], the authors are not aware of attempts to apply these schemes to low-level synchronization primitives.

## 8 Conclusions

In this work, we have reported on our first experiences with PWCS-protected linked data structures. Although our implementations support only one concurrent writer (that is, writers have to synchronize themselves to modify the link structure), our performance measurements indicate that PWCS is an interesting road to follow. We have presented first results on arrays, lists, hash tables and binary trees. Comparing to RCU-based approaches will be one of our next steps, since we assume a higher read side overhead for PWCS, but do not necessarily need grace period detection for object destruction, as long as type stable memory is used. However, a more in-depth evaluation and more complex operations such

as recovery during traversal remain as future work.

The design principles behind PWCS open up a new interesting direction of research: to allow races to occur but make inconsistencies detectable and to construct algorithms such that the likelihood of permanent races diminishes over time. A particular interesting future project would be the combination of such a design method with probabilistic analysis methods such as probabilistic model checking.

## References

- [1] *O. Krieger, M. Stumm, R. Unrau and J. Hanna* “A Fair Fast Scalable Reader-Writer Lock” *PROC. OF THE IEEE INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING*, 1993
- [2] *J. Mellor-Crummey and M. Scott* “Scalable reader-writer synchronization for shared-memory multiprocessors” *3RD ACM SYMP. ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING*, 106-113, April 1991
- [3] *P. McKenney* “Read-Copy Update” <http://www.rdrop.com/users/paulmck/RCU/>
- [4] *J. Valois* “Lock-free linked lists using compare-and-swap” *14TH ANNUAL ACM SYMP. ON PRINCIPLES OF DISTRIBUTED COMPUTING*, 214-222, Aug 1995
- [5] *M. Herlihy* “Wait-free synchronization” *ACM TRANS. ON PROGRAMMING LANGUAGES AND SYSTEMS*, 13(1):124-149, Jan 1991
- [6] *N. McGuire* “Probabilistic Write Copy Select” *REAL-TIME LINUX WORKSHOP*, Oct. 2011
- [7] *B. Engel, M. Völz* “Verfahren und Einrichtung um eine eventuelle Schreibinkonsistenz in nicht kohärenten Architekturen abzusichern.” *PATENT APPLICATION*, 5.1.1 4337.20/A 3332, submitted July 2012