A Lock-Free Algorithm for Concurrent Bags

Håkan Sundell
School of Business and Informatics
University of Borås
501 90 Borås, Sweden
Hakan.Sundell@hb.se

Anders Gidenstam
School of Business and Informatics
University of Borås
501 90 Borås, Sweden
Anders.Gidenstam@hb.se

Marina Papatriantafilou
Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden
ptrianta@chalmers.se

Philippas Tsigas
Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden
tsigas@chalmers.se

ABSTRACT

A lock-free bag data structure supporting unordered buffering is presented in this paper. The algorithm supports multiple producers and multiple consumers, as well as dynamic collection sizes. To handle concurrency efficiently, the algorithm was designed to thrive for disjoint-access-parallelism for the supported semantics. Therefore, the algorithm exploits a distributed design combined with novel techniques for handling concurrent modifications of linked lists using double marks, detection of total emptiness, and efficient memory management with hazard pointer handover. Experiments on a 24-way multi-core platform show significantly better performance for the new algorithm compared to previous algorithms of relevance.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; E.1 [**Data Structures**]: Lists, stacks, and queues

General Terms

Algorithms, Performance, Reliability, Experimentation

Keywords

concurrent, data structure, non-blocking, shared memory

1. INTRODUCTION

Concurrent producer/consumer collections (e.g. set, bag, pool) are fundamental data structures that are key components in applications, algorithms, run-time and operating systems. This paper presents an efficient lock-free and linearizable bag data structure implementation for multiple producers and consumers, supporting the *Add* and the *TryRemoveAny* operations. The *Add* operation adds an item to the collection and the *TryRemoveAny* operation removes an item from the collection unless it is empty. The collection allows any number of occurrences of an item and keeps no information about the order in which items were inserted. A common use of this kind of collection is to communicate data items between producers and consumers in the task of parallelizing applications. Speedup is a major goal in these scenarios where both data and task parallelism are heavily exploited and the collection merely constitutes the "glue" in the application design. Because of this, it is essential that the collection implementation have as high scalability and low overhead as possible. The new algorithm for implementing a lock-free and linearizable bag is designed as a connected set of local data structures in order to maximize parallelism for disjoint accesses.

Non-blocking synchronization is often advocated for the emerging multicore architectures thanks to both its possible advantages in performance and its progress properties. With respect to the latter, the two most important non-blocking methods proposed in the literature are *lock-free* and *wait-free* [1]. Lock-free implementations of shared data structures guarantee that at any point in time in any possible execution some operation will complete in a finite number of steps. In cases with overlapping accesses by concurrent operations, some of them might have to repeat steps in order to complete the operation. However, real-time systems might have stronger requirements on progress, and thus in wait-free implementations each operation is guaranteed to complete in a bounded number of its own steps, regardless of possible concurrent overlaps of the individual steps and the relative execution speeds.

Producer/consumer collections are common means for implementing pipelined design patterns in parallel applications, whereas the proposed bag data structure fits directly when insertion-order are of no importance. Also directly applicable for this kind of parallel design are concurrent LIFO stack, FIFO queue, and "pool" data structures. Solutions for the load-balancing problem such as work-stealing deques [2] have many resemblances to the producer-consumer problem. However, due to different semantics, in order to use these deque data structures as a producer-consumer collection, a new overall connecting algorithm is needed, and the required and linearizable [3] overall semantics cannot be established without significant modifications to the deque algorithms.

Most contemporary programming language frameworks support multi-thread programming and their corresponding framework libraries include an increasing number of concurrent collection data

SPAA'11, June 4–6, 2011, San Jose, California, USA.

structures. For example, .NET 4.0 has defined the `IProducerConsumerCollection` interface with semantics that guarantees no insertion-order of items removed, as well as a concrete implementation with the `ConcurrentBag` class. These data structure implementations, in the .NET and other frameworks, are using various synchronization techniques that are primarily aiming for efficiency. In this paper though, we focus exclusively on strictly non-blocking collection algorithms as implementations being just "concurrent" (while possibly efficient as e.g. "lock-less") are still prone to problems as e.g. deadlocks. Absence of explicit locks does not imply any non-blocking properties, unless the latter are proven to be fulfilled.

A large number of lock-free stack and queue implementations have appeared in the literature, e.g. [4][5][6][7][8][9] being the most influential or most efficient results. These results all have a number of specialties or drawbacks as e.g. static in size, requiring atomic primitives not available on contemporary architectures, and having a high overhead. Moreover, all stacks and queues are inherently limiting the level of disjoint-access-parallelism [10] due to the strict LIFO or FIFO ordering. Afek et al. [11] recently presented a concurrent "pool" implementation with semantics similar to our proposed bag algorithm. Its design aims at allowing high scalability and is also apparently lock-free (although not explicitly stated in the paper), but not linearizable (e.g. it lacks a global state indicating emptiness).

This paper improves on previous results by combining the underlying approaches and designing the new algorithm to maximize efficiency on contemporary multi-core platforms. Also very importantly, the paper aims to provide semantics that conform to a natural intuition of programmers, namely that a concurrent execution's outcome is as if the operations executed sequentially and consistently with their actual time order. Consequently, the new lock-free algorithm is fully linearizable, has no limitations on concurrency, supports a high degree of disjoint-access-parallelism, has a cache-aware design, is fully dynamic in size with efficient memory utilization, and only requires atomic primitives available on contemporary platforms. Experiments on a 24-way multi-core platform shows significantly better performance for the new algorithm compared to all previous semantically compatible lock-free implementations.

The rest of the paper is organized as follows. Section 2 presents the new algorithm together with intuitive arguments for linearizability. In Section 3, related works with lock-free producer/consumer collections are discussed. In Section 4, some benchmark and application experiments are shown. Finally, Section 5 concludes this paper. The detailed implementation description are provided in Appendix A. Corresponding proofs are outlined in more detail in [12].

2. THE NEW LOCK-FREE ALGORITHM

A common and efficient [13, 4] approach for implementing non-blocking collection data structures is to use a continuous (e.g. cyclic) array, where each array element holds a pointer to the stored items. In a concurrent environment, modifications (i.e., insertions and deletions of items) to the array elements are handled by using the *CAS*[1] atomic synchronization primitive. In ordered (e.g. stack and queue) collections, concurrent insertions or deletions have to compete on modifying the same array element (e.g. at the head or tail array index) due to the required ordering semantics, thus limiting the parallelism[2]. However, in the case of the bag data structure there is no enforced ordering, and hence concurrent insertions and deletions can chose to modify any arbitrary array element, anytime. Therefore we propose to distribute the bag data structure by splitting the array, such that each thread normally only operates on its own array part. To enable dynamic growing and shrinking of the bag without any limitations except system memory, each thread maintains a linked list of fixed-size array blocks, where new array blocks are added or removed as needed. Thus, the bag data structure is constructed out of a shared array of linked lists of array blocks, as depicted in Figure 1.

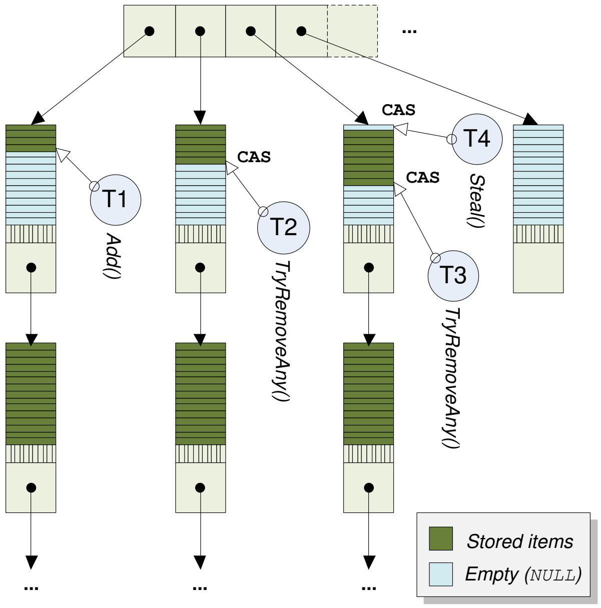

Figure 1: A lock-free bag implemented using a linked list of arrays, where each thread is normally working on its own array block.

The main algorithm steps of the *Add* and *TryRemoveAny* operations are shown in Algorithms 1 and 2 respectively. Each thread is using thread-local storage (TLS), for storing the current active position (threadHead) for adding or removing items and a pointer to the first block (threadBlock) in the thread's linked list. In case this list should be empty, the *TryRemoveAny* operation resorts to trying to steal any item from any other linked list in the bag, as shown in Algorithm 3. Each thread stores in TLS the last tried position (stealHead) and the last tried block (stealBlock).

However, as presented, this simple algorithm has a number of issues that need to be addressed:

- **Dynamic capacity.** As new array blocks are dynamically allocated, old and empty blocks also need to be freed. See section 2.1.

- **Stealing should stop when bag is empty.** As the bag is using a distributed design there is no single shared variable indicating emptiness. See section 2.2.

[1] The Compare-And-Swap (CAS) atomic primitive will update a given memory word, if and only if the word still matches a given value (e.g. the one previously read). CAS is generally available in contemporary systems with shared memory, supported mostly directly by hardware and in other cases in combination with system software.

[2] Some optimizations are possible thanks to the definition of the linearizability property, although the parallelism is only achieved after observing contention on a certain array element.

- **Memory.** Lock-free and dynamic algorithms need lock-free and efficient memory management to avoid dangling pointers. See section 2.3.

Algorithm 1 $Add(item)$

```
1: if $threadHead$ has reached end of array then
2:     Allocate new block and add it in the linked list before $threadBlock$ and set $threadBlock$ to it
3:     $threadHead \leftarrow 0$
4: end if
5: $threadBlock[threadHead] \leftarrow item$
6: $threadHead \leftarrow threadHead + 1$
```

Algorithm 2 $TryRemoveAny()$

```
1: loop
2:     if $threadHead < 0$ then
3:         if $threadBlock$ is last block in linked list then
4:             return $Steal()$
5:         end if
6:         $threadBlock \leftarrow$ next array block in list
7:         $threadHead \leftarrow$ last array position
8:     end if
9:     $item \leftarrow threadBlock[threadHead]$
10:    if $item \neq NULL$ and $CAS(threadBlock[threadHead], item, NULL)$ then
11:        return $item$
12:    else
13:        $threadHead \leftarrow threadHead - 1$
14:    end if
15: end loop
```

Algorithm 3 $Steal()$

```
1: loop
2:     if $stealHead$ has reached end of array then
3:         $stealBlock \leftarrow$ next block in linked list or next list
4:         $stealHead \leftarrow 0$
5:     end if
6:     $item \leftarrow stealBlock[stealHead]$
7:     if $item \neq NULL$ and $CAS(stealBlock[stealHead], item, NULL)$ then
8:         return $item$
9:     else
10:        $stealHead \leftarrow stealHead + 1$
11:    end if
12: end loop
```

2.1 Linked list handling

Whenever any of the array blocks becomes empty (due to own *TryRemoveAny* or concurrent *Steal* operations), its memory should be freed. To handle concurrent deletions of linked list, the same main strategy as used by Harris [14] could be used where next pointers are augmented with a special mark (mark1) that indicates logical deletion. The logical deletion is done using *CAS* and any operation (current or concurrent) that observes this mark should try to perform the actual removal from the linked list of the block using *CAS*.

The *Steal* operations cannot safely (i.e., ensuring emptiness) delete the array blocks on the first position in a linked list, as concurrent *Add* operations would invalidate the performed scan of the array elements. Therefore, the first block should only be deleted by the owning *TryRemoveAny* operation. However, even blocks on the second or further position might be unsafe for *Steal* operations to delete, as these blocks might have become the first block due to concurrent deletions of previous blocks. Our solution is to add an additional mark (mark2) that is set on the next pointer of the previous block using *CAS*, then indicating that the referenced block is logically deleted. Any operation (current or concurrent) that observes this mark should try to set mark1 on the referenced block using *CAS*, as seen in Figure 2. Note that between steps II and III a concurrent operation might succeed with its own steps I and II, and therefore causes an extra step between step III and IV where the current operation needs to propagate the observed mark further on to the next block before proceeding with step IV.

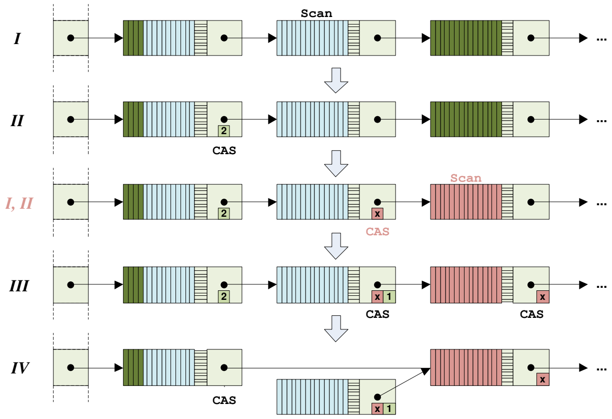

Figure 2: Multiple-step process for marking and deleting blocks (e.g. the middle block) noted to be empty. Another block (in red) might also concurrently be going through the same procedure.

Algorithm 4 $DeleteBlock()$

```
1: if $stealPrev \neq NULL$ then
2:     if $CAS(stealPrev.next, stealBlock, stealBlock + mark2)$ then
3:         Set $mark1$ on $stealBlock.next$ using CAS
4:         if $stealBlock.next$ has $mark2$ then
5:             Set $mark1$ on $stealBlock.next.next$ using CAS
6:         end if
7:         repeat
8:             if $stealPrev.next$ is not referencing $stealBlock$ then
9:                 $UpdateStealPrev()$
10:            end if
11:        until $stealPrev = NULL$ or $CAS(stealPrev.next, stealBlock + mark2, stealBlock.next - mark1)$
12:        $stealBlock \leftarrow$ next block in linked list or next list
13:    end if
14:    $UpdateStealPrev()$
15: else
16:    $stealPrev \leftarrow stealBlock$
17:    $stealBlock \leftarrow$ next block in linked list or next list
18: end if
```

To keep track of the previous block, the additional TLS-variable stealPrev is used and should be continuously set to reference the block preceding stealBlock. Whenever stealBlock is on a new list, stealPrev should be reset to `NULL`. The steps for conditionally deleting a block whenever a *Steal* operation has scanned all array elements to be empty, is described in Algorithm 4. Line 3 of Algorithm 3 should be replaced to a call to *DeleteBlock*. The pur-

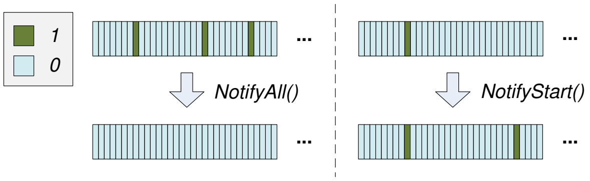

Figure 3: An atomic notification structure for registration of concurrent Add operations.

pose of the *UpdateStealPrev* procedure is to update stealPrev such that it references the block preceding stealBlock. To achieve this, it might have to search from the start of the linked list, and also apply helping of steps III-IV of concurrent operations whenever mark1 or mark2 is noticed on the traversed blocks. If the search fails, stealPrev is set to NULL. Similar helping should also be performed during the deletion (using only mark1) that should take place in Line 6 of Algorithm 2.

2.2 Linearizability and global emptiness detection

To be correct, the algorithm should implement full semantics and for the caller, concurrent operations should appear to take effect atomically, i.e., the algorithm should be linearizable [3]. If the bag is totally empty, the *TryRemoveAny* should terminate and return failure (e.g. NULL). For ensuring total emptiness, the *Steal* operation needs to continuously scan the whole data structure, in combination with some ability to detect concurrent changes due to *Add* operations. Figure 3 illustrates how each individual thread can "subscribe" on pending *Add* operations on a particular block[3] by setting the corresponding bit in a bit array using *CAS*. If an *Add* operation is initiated on this block, it clears the whole bit array by simply writing zero to the memory words, before line 5 of Algorithm 1. By doing so, it notifies all subscribed threads on the pending insert (which will be done in line 6). However, there can be pending inserts that actually have taken place during the scanning, but were initiated before the scanning thread started subscribing. Therefore, we can only ensure total emptiness after having scanned the whole bag (while continuously ensuring no pending inserts and that all array elements are NULL) for as many repetitions as there are threads. The intuition behind this is that each thread can only have up to one pending insert, and thus the bag must have been empty during one of the repetitions. The extension steps to the *Steal* operation are shown in Algorithm 5.

We can now define the specific statements in the algorithm where the operations appear to take effect, i.e., the linearizability points:

- The *Add* operation, takes effect at the write statement in line 6 of Algorithm 1. This follows by the definition how array elements represent items belonging to the bag.

- The *TryRemoveAny* operation returning an item, takes effect at the successful *CAS* statement in line 10 of Algorithm 2 or line 7 of Algorithm 3. This follows by the definition how array elements represent items belonging to the bag.

Algorithm 5 $Steal() - Extension$

```
1:  for $i = 0$ to $NR\_THREADS + 1$ do
2:      repeat
3:          Perform the steps of the $Steal$ algorithm for one block at a time
4:          if $i = 1$ then
5:              Set the corresponding bit for this thread in the notification array of the block using CAS
6:          else if $i > 1$ then
7:              if The bit for this thread in the notification array is clear or a non-empty array element has been found then
8:                  $i \leftarrow 0$
9:              end if
10:         end if
11:     until All lists in the bag have been scanned
12: end for
13: return NULL
```

- The *TryRemoveAny* operation returning NULL, takes effect at the read statement of the set notification bit in line 7 of Algorithm 5 during one of the repetitions. This follows by the previous reasoning about detecting total emptiness.

2.3 Memory management

Whenever an empty array block has been fully removed (e.g. after the successful *CAS* in line 11 of Algorithm 4) from the linked list, its memory should be freed and be made available for system allocation. However, other threads are concurrently traversing the linked list, and might consequently have TLS or local variables that reference the deleted block. For handling this problem, we build on the efficient and lock-free memory management scheme proposed by Michael [9] which makes use of shared "hazard" pointers. Throughout the whole bag algorithm, whenever a new block is visited and its memory address stored in a variable, a corresponding hazard pointer must be set. The memory management system will then make sure that a block of memory is not reused until no hazard pointers are referencing it. Note that memory of a deleted block might be reused even though there are next pointers from other blocks pointing to it.

However, this becomes a problem when trying to dereference a next pointer with mark1. As the next pointer belongs to a block that is being deleted, it is not possible to acquire a safe pointer to the block referenced by the next pointer. On the other hand, this is something that the bag algorithm needs to do, e.g. line 5 of Algorithm 4. To meet this requirement, our solution is to extend the memory management scheme [9] such that it can handle a "handover" of a hazard pointer between two or more threads. This works in the way that before removing the block from the linked list with *CAS*, the current thread sets a hazard pointer to the block referenced by the next pointer. After having performed the full removal of the block containing the marked next pointer, the next pointer is set to NULL. Before clearing the hazard pointer (holding a reference to the block after the deleted one), it signals to the memory management system to perform another scan of hazard pointers, as another thread might now have set a hazard pointer to the same block. In this way, any other thread may now safely dereference also next pointers with mark1.

3. RELATED WORK DISCUSSION

Treiber presented a lock-free stack (a.k.a. IBM Freelist), which was later efficiently fixed from the ABA[4] problem by Michael [9].

[3] We chose to place this notification structure per block instead of per linked list, in order to reduce the number of cache lines invalidated by an *Add* operation. As a consequence of this choice, the last block in a linked list is never deleted, and notification state is propagated downwards whenever blocks are removed.

[4] The ABA problem is due to the inability of *CAS* to detect concurrent changes of a memory word from a value (A) to something else (B) and then again back to the first value (A).

Hendler et al. [8] presented an extension where randomized elimination[5] is used as a back-off strategy and for increasing scalability when contention on the stack's head is noticed via failed *CAS* attempts.

Tsigas and Zhang [4] presented a lock-free queue that is an extension of [13] for multiple producers and consumers, where synchronization is done both directly on the array elements and the shared head and tail indices using *CAS*. In order to avoid the ABA problem when updating the array elements, the algorithm exploits using two (or more) null values. Moreover, for lowering the memory contention the algorithm alternates every other operation between scanning and updating the shared head and tail indices. Michael and Scott [5] presented a lock-free queue based on a linked list. Synchronization is done via shared pointers indicating the current head and tail node as well via the next pointer of the last node, all updated using *CAS*. The queue is fully dynamic as more nodes are allocated as needed when new items are added. The original presentation used unbounded version counters, and therefore required double-width *CAS* which is not supported on all contemporary platforms. The problem with the version counters can easily be avoided by using some memory management scheme as e.g. [9]. Moir et al. [6] presented an extension where randomized elimination is used as a back-off strategy and for increasing scalability when contention on the queue's head or tail is noticed via failed *CAS* attempts. However, elimination is only possible when the queue is close to be empty during the operation's invocation. Hoffman et al. [7] takes another approach to increase scalability by allowing concurrent *Enqueue* operations to insert the new node at adjacent positions in the linked list if contention is noticed during the attempted insert at the very end of the linked list. To enable these "baskets" of concurrently inserted nodes, removed nodes are logically deleted before the actual removal from the linked list, and as the algorithm traverses through the linked list it requires stronger memory management than [9] and a strategy to avoid long chains of logically deleted nodes.

In resemblance to [4] the new algorithm uses *CAS* and arrays to store (pointers to) the items. Moreover, shared indices are avoided and scanning [4] is always used for finding empty or occupied array elements. In contrast to [4] the array is not static or cyclic, but instead more arrays are dynamically allocated as needed when new items are added, making our bag fully dynamic. In resemblance to [5][6][7] the new algorithm is dynamic, and in resemblance to [7] removed blocks are logically deleted and blocks are being traversed. In contrast to [7], the new algorithm suffice with a slightly modified version of [9] for memory management purposes.

Afek et al. [11] presented a pool data structure, where the collection is consisting of several lock-free queues on which the load is distributed. The pool is using randomized elimination combined with diffraction in a tree-like manner, where elimination is used for increasing disjoint-access-parallelism. In resemblance to [11] the new algorithm uses several underlying data structures, but instead of randomization it is improving disjoint-access-parallelism inherently by its algorithmic design and avoiding global synchronization. In addition, the new algorithm is linearizable.

Fomitchev and Ruppert [15] extended the linked list structure by Harris [14] with having an additional mark on pointers, where the new mark was used on the node preceding the node to be marked ordinarily, in order to earlier inform concurrent operations traversing the linked list about the ongoing node removal. In contrast to [15], the new algorithm are using the additional mark in order to distinguish between logical deletions done by the owner thread and the stealing threads respectively.

Arora et al. [2] presented an efficient and lock-free work-stealing deque based on arrays. It was later improved to handle dynamic sizes by Hendler et al. [16] that used a doubly-linked list of arrays, where each array can be dynamically allocated. In resemblance to [2][16], the new algorithm assign each thread its own data structure, uses arrays as the basic representation for storing items, and *CAS* is only used for updates that can be concurrently updated by several threads. In resemblance to [16], additional array blocks are dynamically allocated when needed, although the new algorithm only maintains a single-linked list. In contrast to [2][16], the new algorithm provides linearizable bag semantics; e.g. provides a global state indicating emptiness.

Cache-aware algorithms for non-blocking data structures have attended an increasing interest, with several recent results in the literature, e.g. [17].

4. EXPERIMENTAL STUDY

We have evaluated the performance of our new lock-free bag implementation by the means of some custom micro-benchmarks. The purpose is to estimate how well the implementation compares with other known lock-free compatible implementations under high contention and increasing concurrency. The benchmarks are the following:

1. Random 50%/50%. Each thread is randomly (the sequence is decided beforehand) executing either an *Add* or *TryRemoveAny* operation.

2. 1 Producer / N-1 Consumers. Each thread (out of N) is either a producer or consumer, throughout the whole experiment. The producer is repeatedly executing *Add* operations, whereas the consumers are executing *TryRemoveAny*.

3. N-1 Producers / 1 Consumer. Performed as the previous benchmark, except with another distribution of producers and consumers.

4. N/2 Producers / N/2 Consumers. Performed as the previous benchmark, except with another distribution of producers and consumers.

We have also evaluated the performance of our new implementation in the scope of real applications. For this purpose we chose to compute and render an image of the Mandelbrot set [18] in parallel using the producer/consumer pattern. The program uses a shared collection data structure that is used for communication between the program's two major phases:

- Phase 1 consists of computing the number (with a maximum of 255) of iterations for a given set of points within a chosen region of the image. The results for each region together with its coordinates are then put in the collection data structure.

- Phase 2 consists of, for each computed region stored in the collection, computing the RGB values for each contained point and draw these pixels to the resulting image. The colors for the corresponding number of iterations are chosen according to a rainbow scheme, where low numbers are rendered within the red and high numbers are rendered within the violet spectrum.

Phase 1 is performed in parallel with phase 2, i.e., like a pipeline. Half of the threads perform phase 1 and the rest perform phase

[5] If not conflicting with linearizability, two concurrent and matching operations might be eliminated by exchanging data directly without passing through the main data structure.

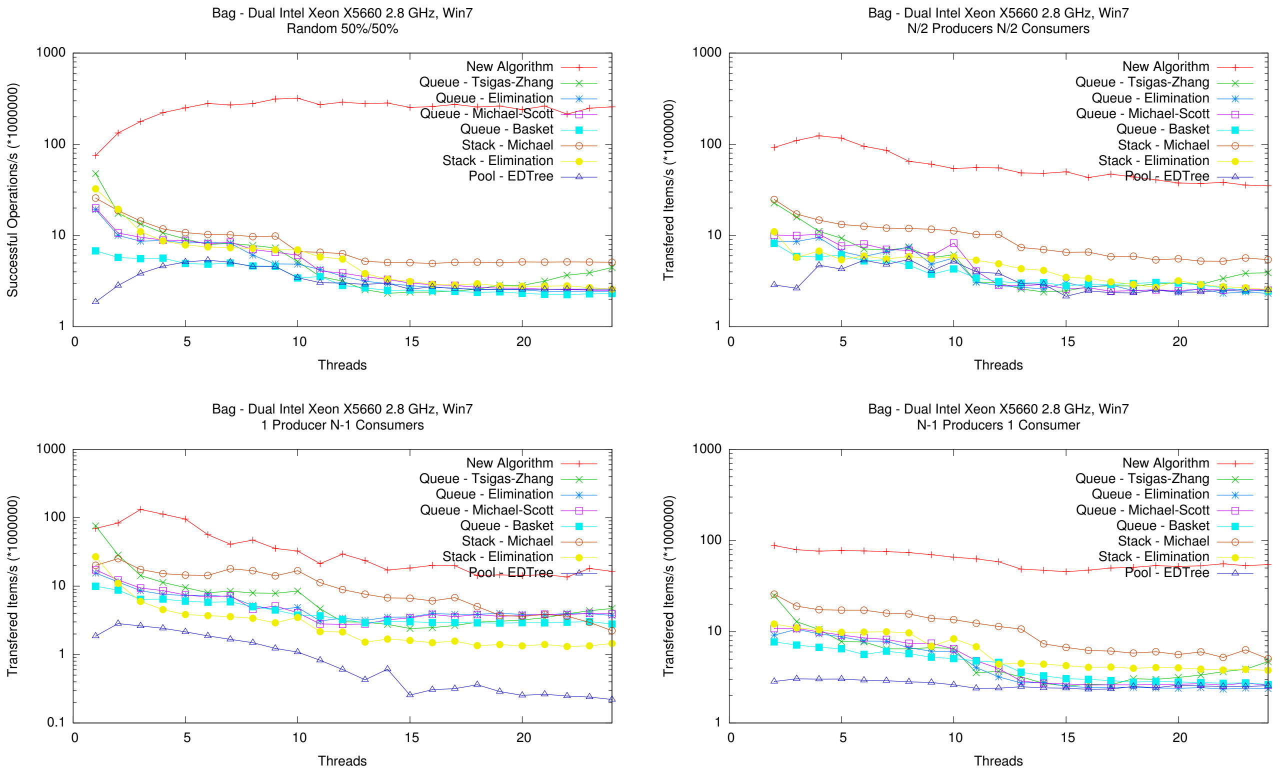

Figure 4: Benchmark experiments on a 24-way Dual Intel Xeon processor system.

2. We have implemented this application in C, for the purpose of rendering a 32-bit color image of 2048 times 2048 pixels. The size of each square-shaped region is chosen to be one of 16x16 (i.e., 16 by 16), 8x8, 4x4, or 2x2 pixels. The whole image is divided into a number (equal to the number of threads) of larger parts[6], where each producer thread (i.e., phase 1) work sequentially on the regions contained within its own part. The consumer threads (i.e., phase 2) render the regions got from the collection in the order that they were obtained, until the producer threads have finished and the collection is empty.

For comparison we have implemented the dynamic lock-free queue by Michael and Scott [5], the same with elimination [6], the baskets queue [7], and the static cyclic array lock-free queue presented in [4]. For comparison, we also implemented the lock-free stack by Michael [9], and the same with elimination by Hendler et al. [8]. Moreover, we implemented the ED-tree based lock-free (although not linearizable) pool by Afek et al. [11].

All dynamic stacks, queues and pools (as well as the new bag algorithm) have been implemented to support collection sizes only limited by the system's memory, i.e., using lock-free management schemes [9] or [19] and lock-free free-lists where appropriate. For the new implementation, the size of the array block (`BLOCK_SIZE`) is chosen to fit within one cache line. All implementations are written in C and compiled with the highest optimization level. For all implementations, synchronization hotspots were padded with dummy bytes in order to avoid false sharing. In our benchmark experiments, each concurrent thread is started at the very same time and each benchmark runs for one second for each implementation. Exactly the same sequence of operations was performed per thread for all different implementations compared. A clean-cache operation was also performed just before each run, and final results are taken as an average over 10 runs. All benchmark and application experiments have been executed on a dual Intel Xeon X5660 2.8 GHz with 12 GB DDR3 1333 MHz system running Windows 7 64-bit. Each of the two processors has 6 cores, each core being capable of executing 2 threads each, making up to 24 hardware threads in total.

The results from the benchmark experiments with up to 24 threads are shown in Figure 4. Note that the results are shown in logarithmic scale, due to the very large difference in performance for the different implementations. The results of benchmarks 1 show the number of successful (failed *TryRemoveAny*s, *Pop*s or *Dequeue*s are not counted) operations executed per second in the system in total. The results of benchmarks 2-4 show the number of items per second that have passed through the collection (i.e., the number of successful *TryRemoveAny*, *Pop* or *Dequeue* operations). Interestingly, the new algorithm performs significantly, often with a magnitude, better than all the other implementations. The superior performance and scalability can be explained by the distributed design and the low synchronization overhead. The declining scalability with increasing number of threads (a behavior shared with the pool implementation) can be explained being in major due to memory bandwidth saturation.

The results from the application experiments with up to 24 threads are shown in Figure 5. As the Mandelbrot set is known as being an

[6] Due to the nature of the Mandelbrot set, this way of deciding each part might not be fair in respect of workload per thread. As can be seen in the experimental results, this partition pattern causes that 3 parts take longer time than 2 parts in parallel, because the total execution time depends on the slowest part.

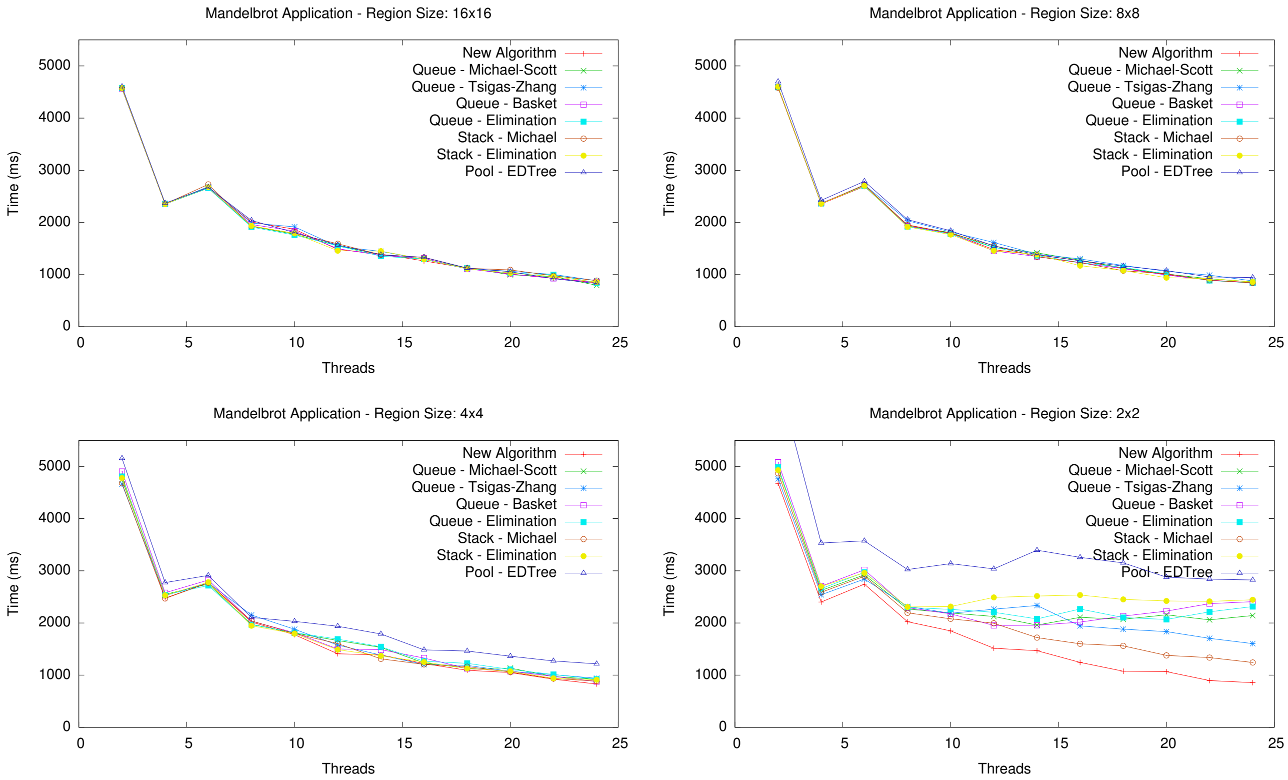

Figure 5: Application experiments on a 24-way Dual Intel Xeon processor system.

embarrassingly parallel problem, the amount of communication is relatively low, and therefore the impact of the choice of collection implementation is expected to be of minor importance. For example, this behavior is apparent with the experiment using 16x16-sized regions. However, as regions get smaller the amount of communication increases, and the performance of the collection will have a bigger impact on the overall performance. Consequently, as can be started to be seen with 4x4 and then with the 2x2 region size, the new algorithm is fast even in cases where there is no significant contention, and as contention becomes higher in more communication intensive instances the new algorithm shows its good parallel behavior.

5. CONCLUSIONS

We have presented a new algorithm for implementing a lock-free producer/consumer collection data structure. To the best of our knowledge, this is the first lock-free bag algorithm with all of the following properties:

- Distributed design, allowing disjoint-access-parallelism.
- Exploiting thread-local static storage.
- Dynamic in size via lock-free memory management.
- Only requires atomic primitives available in contemporary systems.

The algorithm has been shown to be lock-free and linearizable. Experiments on a contemporary multi-core platform shows significantly better performance for the new algorithm compared to previous state-of-the-art lock-free implementations of data structures that can serve as producer/consumer platforms. We believe that our implementation should be of highly practical interest to contemporary and emerging multi-core and multi-processor system thanks to both its high performance and its strong progress and consistency guarantees.

6. REFERENCES

[1] M. Herlihy, “Wait-free synchronization,” *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 124–149, Jan. 1991.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, “Thread scheduling for multiprogrammed multiprocessors,” in *ACM Symposium on Parallel Algorithms and Architectures*, 1998, pp. 119–129.

[3] M. Herlihy and J. Wing, “Linearizability: a correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[4] P. Tsigas and Y. Zhang, “A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems,” in *Proceedings of the 13th annual ACM Symposium on Parallel Algorithms and Architectures*, 2001, pp. 134–143.

[5] M. M. Michael and M. L. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,” in *Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing*, 1996, pp. 267–275.

[6] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, “Using elimination to implement scalable and lock-free fifo queues,”

in *Proceedings of the 17th annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2005, pp. 253–262.

[7] M. Hoffman, O. Shalev, and N. Shavit, “The baskets queue,” in *Proceedings of the 11th International Conference on Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, vol. 4878. Springer, 2007, pp. 401–414.

[8] D. Hendler, N. Shavit, and L. Yerushalmi, “A scalable lock-free stack algorithm,” *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp. 1–12, 2010.

[9] M. M. Michael, “Hazard pointers: Safe memory reclamation for lock-free objects,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 8, Aug. 2004.

[10] A. Israeli and L. Rappoport, “Disjoint-access-parallel implementations of strong shared memory primitives,” in *Proceedings of the 13th annual ACM symposium on Principles of Distributed Computing*, Aug. 1994.

[11] Y. Afek, G. Korland, M. Natanzon, and N. Shavit, “Scalable producer-consumer pools based on elimination-diffraction trees,” in *Euro-Par 2010*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6272, pp. 151–162.

[12] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas, “A lock-free algorithm for concurrent bags,” Department of Computer Science and Engineering, Chalmers University of Technology, Tech. Rep. 2011:01, January 2011.

[13] L. Lamport, “Specifying concurrent program modules,” *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2, pp. 190–222, 1983.

[14] T. L. Harris, “A pragmatic implementation of non-blocking linked lists,” in *Proceedings of the 15th International Symposium of Distributed Computing*, Oct. 2001, pp. 300–314.

[15] M. Fomitchev and E. Ruppert, “Lock-free linked lists and skip lists,” in *Proceedings of the 23rd annual symposium on Principles of Distributed Computing*, Jul. 2004, pp. 50–59.

[16] D. Hendler, Y. Lev, M. Moir, and N. Shavit, “A dynamic-sized nonblocking work stealing deque,” *Distributed Computing*, vol. 18, no. 3, pp. 189–207, 2006.

[17] A. Braginsky and E. Petrank, “Locality-conscious lock-free linked lists,” in *ICDCN*, ser. Lecture Notes in Computer Science, M. K. Aguilera, H. Yu, N. H. Vaidya, V. Srinivasan, and R. R. Choudhury, Eds., vol. 6522. Springer, 2011, pp. 107–118.

[18] B. B. Mandelbrot, “Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z,” *Annals of the New York Academy of Sciences*, vol. 357, pp. 249–259, 1980.

[19] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas, “Efficient and reliable lock-free memory reclamation based on reference counting,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 8, pp. 1173–1187, 2009.

APPENDIX

A. ALGORITHM DETAILS

The specific fields of each array block are described in Program 1 as it is used in this implementation.

In the algorithm, there is no shared information about which array index that is currently being the next active for the *Add* or *TryRemoveAny* operation. Instead each thread is storing related information in separate memory using thread-local storage (TLS), its own index (threadID) into the global shared array, and pointers indicating the last known (by this thread) first block (threadBlock) as well as active indices (threadHead) for inserting and removing items. In addition, each thread also keeps track of the last block (stealBlock), the corresponding index (stealHead), and its preceding block (stealPrev) used for "stealing" items from other thread’s (i.e., the thread with global index stealIndex) linked lists.

Program 1 The block structure and auxiliary functions.

```
struct blockp_t {
    block_t * p; bool mark2 :1; bool mark1 :1;
};
struct block_t : public node_t {
    void * nodes[BLOCK_SIZE];
    long notifyAdd[NR_THREADS/WORD_SIZE];
    blockp_t next;
};
void Mark1Block(block_t *block) {
    for(;;) {
        blockp_t next = block->next;
        if(next.p == NULL || next.mark1 || CAS(&block->
             next,next,{next.p,next.mark2,true})) break;
    }
}
block_t * NewBlock() {
    block_t * block = NewNode(sizeof(block_t));
    block->next = NULL;
    NotifyAll(block);
    for(int i=0;i<BLOCK_SIZE;i++) block->nodes[i]=NULL;
    return block;
}
void NotifyAll(block_t *block) {
    for(int i=0;i<NR_THREADS/WORD_SIZE;i++)
        block->notifyAdd[i] = 0;
}
void NotifyStart(block_t *block, int Id) {
    do {
        long old = block->notifyAdd[Id/WORD_SIZE];
    } while(!CAS(&block->notifyAdd[Id/WORD_SIZE],old,old
         |(1<<(Id%WORD_SIZE))));
}
bool NotifyCheck(block_t *block, int Id) {
    return (block->notifyAdd[Id/WORD_SIZE]&(1<<(Id%
         WORD_SIZE)))==0;
}
void InitBag() {
    for(int i=0;i<NR_THREADS;i++) globalHeadBlock[i]=NULL
         ;
}
void InitThread() {
    threadBlock = globalHeadBlock[threadID];
    threadHead = BLOCK_SIZE;
    stealIndex = 0;
    stealBlock = NULL; stealPrev = NULL;
    stealHead = BLOCK_SIZE;
}
// Shared variables
block_t * globalHeadBlock[NR_THREADS];
// Thread-local storage
block_t * threadBlock, stealBlock, stealPrev;
bool foundAdd;
int threadHead, stealHead, stealIndex;
int threadID; // Unique number between 0 ... NR_THREADS
```

When an array block gets fully empty, the block itself should be removed from the corresponding linked list. As this can happen concurrently with insertions and removals of other blocks, we apply the method introduced in [14] where blocks are first marked (using bit 0 of the next pointer) and then removed from the linked list using *CAS*. Concurrent operations that observe the mark are obliged to "help" by also trying to fulfill the full removal, and thus supporting the lock-free property. However, detection of a block to be fully empty is not straight-forward, as the first block of each linked list can have concurrent *Add* operations, adding new items directly after a stealing *TryRemoveAny* operation have found the array elements at the corresponding array indices to be empty during its

Program 2 The new Add operation.

```
void Add(void *item) {
    int head = threadHead;
    block_t * block = threadBlock;
    for(;;) {
        if(head==BLOCK_SIZE) {
            block_t *oldblock = block;
            block = NewBlock();
            block->next=oldblock;
            globalHeadBlock[threadID]=block;
            threadBlock = block;
            head = 0;
        }
        else if(block->nodes[head]==NULL) {
            NotifyAll(block);
            block->nodes[head]=item;
            threadHead = head+1;
            return;
        }
        else head++;
    }
}
```

Program 3 The new TryRemoveAny operation.

```
void * TryRemoveAny() {
    int head = threadHead-1;
    block_t * block = threadBlock;
    int round = 0;
    for(;;) {
        if(block == NULL || (head<0 && block->next.p ==
             NULL)) {
            do {
                int i=0;
                do {
                    void *result = TryStealBlock(round);
                    if(result!=NULL) return result;
                    if(foundAdd) {
                        round=0; i=0;
                    }
                    else if(stealBlock==NULL) i++;
                } while(i<NR_THREADS);
            } while(++round<=NR_THREADS);
            return NULL;
        }
        if(head<0) {
            Mark1Block(block);
            for(;;) {
                blockp_t next=DeRefLink(&block->next);
                if(next.mark2) Mark1Block(next.p);
                if(next.mark1) {
                    if(next.p) NotifyAll(next.p);
                    if(CAS(&globalHeadBlock[threadId],
                         block,next.p)) {
                        block->next = {NULL,false,true};
                        DeleteNode(block); ReScan(next);
                        block=next.p;
                    }
                    else block=DeRefLink(&globalHeadBlock
                         [threadId]);
                }
                else break;
            }
            threadBlock = block;
            threadHead = BLOCK_SIZE;
            head = BLOCK_SIZE-1;
        }
        else {
            void *data = block->nodes[head];
            if(data==NULL) head--;
            else if(CAS(&block->nodes[head],data,NULL)) {
                threadHead = head;
                return data;
            }
        }
    }
}
```

scan. Moreover, even though blocks on second or further position in a linked list cannot have concurrent *Add* operations, the block itself might concurrently change position in the linked list, and actually become the first block. Therefore, detection of emptiness and marking for removal must be done in one atomic step that also includes the verification of the block not being the first in the linked list. To enable this, stealPrev is set to reference the previous block in the linked list, before starting the scan of the array elements of the current block. If all array elements were empty, we now need to verify that the next pointer of stealPrev is still pointing to the current block (if there exists a previous block, the current block cannot be the first) and at the same time marking the next pointer of the current block. As this cannot be done using the available single-word *CAS*, we instead introduce a second mark (bit 1 of the next pointer) indicating that the next block is marked for removal, and thus we can atomically verify the next pointer of stealPrev referencing the current block and also setting the second mark of it by using *CAS*. The further steps are then to mark the actual block of interest for removal and then proceed with the removal from the linked list, something also required to be done by any concurrent operation that have observed the second mark to be set.

As the data structure is based on array blocks where each array element can contain an item or not, these blocks are not always fully utilized. From a system perspective it is therefore necessary to maximize the overall utilization and as well provide a lower bound on minimal utilization of memory. Therefore, an essential rule for *TryRemoveAny* operations that need to "steal" items from other thread's linked lists, is to never leave a visited array block until all items (including the array block itself) have been removed, whenever this rule is possible to fulfill.

If the data structure is totally empty, the *TryRemoveAny* should terminate and return failure (e.g. NULL). However, as the data structure is highly distributed, there is no single variable for identification of global emptiness. Instead, the operation needs to scan the whole data structure step by step, in combination with some ability to detect if something has changed (i.e., concurrent *Add* operations) since it started the global scanning. Figure 3 illustrates how each individual thread can "subscribe" on pending *Add* operations on a particular block by setting the corresponding bit in a bit array. If an *Add* operation is initiated on this block, it clears the whole bit array, thus "notifying" all subscribed threads on the pending insert. However, as there can be pending inserts that actually have taken place during the scanning, but were initiated before the scanning thread started subscribing, the whole scanning procedure has to repeated globally a certain number of times to ensure global emptiness. Note that even though this implies a relatively long time for detecting global emptiness, it does not affect the time for a *TryRemoveAny* operation to find and remove items added by concurrent *Add* operations during the scanning.

For our implementation of the new lock-free bag algorithm, we have selected a slightly modified version of the lock-free memory management scheme proposed by Michael [9] which makes use of shared "hazard" pointers. The interface defined by the memory management scheme is listed in Program 4. Using this scheme we can assure that an array block can only be reclaimed when there are no local references to it from pending concurrent operations or from pointers in thread-local storage. The *ReScan* operation is an extension to the original scheme by [9]. The purpose of this operation is to force a re-scan of a deleted node, in order to avoid the problem of a normal scan possibly missing hazard pointers "moving" from one thread to another (i.e., avoiding the scenario when

Program 4 The functionality supported by the memory management scheme.

```
node_t * NewNode(int size);
void DeleteNode(node_t *node);
node_t * DeRefLink(node_t **link);
void ReleaseRef(node_t *node);
void ReScan(node_t *node);
```

Program 5 The auxiliary TryStealBlock function.

```
void * TryStealBlock(int round) {
    int head = stealHead;
    block_t * block = stealBlock;
    foundAdd = false;
    if(block==NULL) {
        block = DeRefLink(&globalHeadBlock[stealIndex]);
        stealBlock = block;
        stealHead = head = 0;
    }
    if(head==BLOCK_SIZE) {
        stealBlock = block = NextStealBlock(block);
        head = 0;
    }
    if(block==NULL) {
        stealIndex = (stealIndex+1)%NR_THREADS;
        stealHead = 0;
        stealBlock = NULL; stealPrev = NULL;
        return NULL;
    }
    if(round==1) NotifyStart(block,threadId);
    else if(round>1 && NotifyCheck(block,threadId))
     foundAdd = true;
    for(;;) {
        if(head==BLOCK_SIZE) {
            stealHead = head;
            return NULL;
        }
        else {
            void *data = block->nodes[head];
            if(data==NULL) head++;
            else if(CAS(&block->nodes[head],data,NULL)) {
                stealHead = head;
                return data;
            }
        }
    }
}
```

there are first one active hazard pointer by thread i and then one by thread j with a small overlap in time where both are active, although both are missed during the concurrent scan and consequently the node is wrongly reclaimed).

In order to simplify the description of our new algorithm, we have omitted some of the details of applying the operations of the memory management [9]. In actual implementations, *ReleaseRef* calls should be inserted at appropriate places whenever a variable holding a safe pointer goes out of scope or is reassigned. The detailed descriptions of the *Add* and *TryRemoveAny* operations are listed in Programs 2 and 3 respectively, with the auxiliary functions *TryStealBlock* and *NextStealBlock* used by *TryRemoveAny* listed in Programs 5 and 6. The purpose of the *TryStealBlock* function is to continue to try stealing from the last tried index (stealHead) in the current block of stealing (stealBlock), and either return the removed item or `NULL` in case the current block was found to be empty. The purpose of the *NextStealBlock* function is to try removing the current block, unless it should not do so (e.g., if it was the first or last block in the list), and then advance stealBlock to the next block. If the current block could not be removed due to concurrent changes in the linked list, it returns the current block without advancing.

Program 6 The auxiliary NextStealBlock function.

```
block_t * NextStealBlock(block_t *block) {
    blockp_t next;
    for(;;) {
        if(block==NULL) {
            block = DeRefLink(&globalHeadBlock[stealIndex
                ]);
            break;
        }
        next = DeRefLink(&block->next);
        if(next.mark2) Mark1Block(next.p);
        if(stealPrev == NULL || next.p == NULL) {
            if(next.mark1) {
                if(next.p) NotifyAll(next.p);
                if(CAS(&globalHeadBlock[stealIndex],block
                    ,next.p)) {
                    block->next = {NULL,false,true};
                    DeleteNode(block); ReScan(next);
                }
                else {
                    stealPrev = NULL;
                    block = DeRefLink(&globalHeadBlock[
                        stealIndex]);
                    continue;
                }
            }
            else stealPrev = block;
        }
        else {
            if(next.mark1) {
                blockp_t prevnext = {block,stealPrev->
                    next.mark2,false};
                if(CAS(&stealPrev->next,prevnext,next.p))
                     {
                    block->next = {NULL,false,true};
                    DeleteNode(block); ReScan(next);
                }
                else {
                    stealPrev = NULL;
                    block = DeRefLink(&globalHeadBlock[
                        stealIndex]);
                    continue;
                }
            }
            else if(block==stealBlock) {
                if(CAS(&stealPrev->next,block,{block,true
                    ,false})) {
                    Mark1Block(block);
                    continue;
                }
                else {
                    stealPrev = NULL;
                    block = DeRefLink(&globalHeadBlock[
                        stealIndex]);
                    continue;
                }
            }
            else stealPrev = block;
        }
        if(block == stealBlock || next.p == stealBlock) {
            block=next.p;
            break;
        }
        block=next.p;
    }
    return block;
}
```