# *Drop the Anchor*: Lightweight Memory Management for Non-Blocking Data Structures[*]

Anastasia Braginsky[†]
Computer Science
Technion
anastas@cs.technion.ac.il

Alex Kogan[‡]
Oracle Labs
alex.kogan@oracle.com

Erez Petrank
Computer Science
Technion
erez@cs.technion.ac.il

## ABSTRACT

Efficient memory management of dynamic non-blocking data structures remains an important open question. Existing methods either sacrifice the ability to deallocate objects or reduce performance notably. In this paper, we present a novel technique, called *Drop the Anchor*, which significantly reduces the overhead associated with the memory management while reclaiming objects even in the presence of thread failures. We demonstrate this memory management scheme on the common linked list data structure. Using extensive evaluation, we show that Drop the Anchor significantly outperforms Hazard Pointers, the widely used technique for non-blocking memory management.

## Categories and Subject Descriptors

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

## General Terms

Algorithms, Design, Theory

## Keywords

Concurrent data structures, progress guarantee, lock-freedom, parallel programming, linked list, memory management, hazard pointers, timestamps, freezing

## 1. INTRODUCTION

Non-blocking data structures [9, 11] are fast, scalable and widely used. In the last two decades, many efficient non-blocking implementations for almost any common data structure have been developed. However, when designing a dynamic non-blocking data structure, one must address the non-trivial issue of how to manage its memory. Specifically, one has to ensure that whenever a thread removes some internal node from the data structure, then (a) the memory occupied by this node will be eventually deallocated (i.e., returned to the memory manager for arbitrary reuse), and (b) no other concurrently running thread will access the deallocated memory, even though some threads might hold a reference to the node.

Previous attempts to tackle the memory management problem had limited success. Existing non-blocking algorithms usually take two standard approaches. The first approach is to rely on automatic garbage collection (GC), simply deferring the problem to the GC. By doing this, the designers hinder the algorithm from being ported to environments without GC [5]. Moreover, the implementations of these designs with currently available (blocking) GC's cannot be considered non-blocking.

The second approach taken by designers of concurrent data structures is to adopt one of the available non-blocking memory management schemes. The most common schemes are probably the Hazard Pointers technique by Michael [14] or the similar Pass the Buck method by Herlihy et al. [10]. In these schemes, each thread has a pool of global pointers, called *hazard pointers* in [14] or *guards* in [10], which are used to mark objects as "live" or ready for reclamation. When a thread $t$ reclaims a node, $t$ adds the node to a special local reclamation buffer. Once in a while, $t$ scans its buffer and for each node it checks whether some other thread has a hazard pointer[1] to the node. If not, that node can be safely deallocated. Special attention must be given to the time interval after a thread obtains a reference to an object and before it registers this object in a hazard pointer. During this time, the object may be reclaimed and reallocated. Thus, by the time it gets protected by a hazard pointer, it could have become a completely different entity. This delicate point enforces validation of the object's state after assigning it with a hazard pointer.

Although these techniques are not universal (i.e., there is no automatic way to incorporate them into a given algorithm), they are relatively simple. Moreover, a failure

[*] Supported by the Israeli Science Foundation (grant No. 283/10).

[†] Supported by the Ministry of Science and Technology, Israel.

[‡] The work on this paper was done while the author was with the Department of Computer Science, Technion.

[1] In this paper we will use the term "hazard pointers", but guard pointers are equally relevant.

*SPAA'13,* July 23–25, 2013, Montréal, Québec, Canada.
Copyright 2013 ACM 978-1-4503-1572-2/13/07 ...$15.00.

of one thread prevents only a small number of nodes (to which the failed thread has references in its hazard pointers) from being deallocated. The major drawback of these techniques, however, is their significant runtime overhead, caused mainly by the management and validation of the global pointers required before accessing *each* internal node for the first time [8]. Along with that, expensive instructions, such as memory fences or compare-and-swap (CAS) instructions [14, 10, 8], are required for correctness of those schemes. Moreover, if the validation fails, the thread must restart its operation on the data structure, harming the performance further.

Another known method for memory management uses per-thread timestamps, as in [7], which are incremented by threads before every access to the data structure. When a thread removes a node, it records the timestamps of other threads. Later, it can deallocate the node once all threads increase their timestamps beyond the recorded values. Although this method is very lightweight, it is vulnerable to thread delays and failures. In such cases, memory space of an unbounded size may become impossible to reclaim [14].

In this paper, we concentrate on the linked list, one of the most fundamental data structures, which is particularly prone to the shortcomings of previous approaches [8]. The presented technique eliminates the performance overhead associated with the memory management without sacrificing the ability to deallocate memory in case of thread failures. The good performance of our technique stems from the assumption that thread failures are typically very uncommon in real systems, and if they do occur, this is usually indicative of more serious problems than being unable to deallocate some small part of memory. Our approach provides a flexible tradeoff between the runtime overhead introduced by memory management and the size of memory that might be lost when some thread fails.

Our memory management technique builds on a combination of three ideas: timestamps, anchors and freezing. As in [7], we use per-thread timestamps to track the activity of each thread on the data structure. Similarly to [14], we use global pointers, which we call anchors. Unlike [14], however, a thread drops the anchor (i.e., records a reference in the anchor) every bunch of node accesses, e.g., every one thousand nodes it traverses. As a result, the amortized cost of anchor management is spread across multiple node accesses and is thus very low. To recover the data structure from a failure of a thread $t$, we apply freezing [3]. That is, using $t$'s anchors, other threads mark nodes that $t$ may hold a reference to, as frozen. Then they copy and replace the frozen part of the data structure, restoring the ability of all threads to deallocate memory. The recovery operation is relatively expensive, but it is required only in the uncommon case in which a thread fails to make progress for a long while. Thus, the overall cost of the memory management remains very low.

We have implemented our scheme in C and compared its performance to the widely used implementation of the linked list based on Hazard Pointers (HP) [14]. Our performance results show that the total running time, using the anchor-based memory management, is about 250–500% faster the one based on HP. We also discuss how to apply our technique on other data structures, where the use of other approaches for memory management is more expensive.

## 2. RELATED WORK

Memory management can be fairly considered as the Achilles heel of many dynamically sized non-blocking data structures. In addition to the techniques mentioned in the introduction (that use per-thread timestamps [7] or global pointers [14, 10]), one can also find an approach based on reference counting [16, 6, 4, 15]. There, the idea is to associate a counter with every node, which is updated atomically when a thread gains or drops a reference to the node. Such atomic updates are typically performed with a fetch-and-add instruction, and the node can be safely removed once its reference count drops to zero. This approach suffers from several drawbacks, such as requiring each node to keep the reference count field even after the node is reclaimed [16, 15] or using uncommon atomic primitives, such as double compare-and-swap (DCAS) [4]. The major problem, however, remains performance [14, 8], since even when applying a read-only operation on the data structure, this approach requires atomic reference counter updates on every node access.

In a related work, Hart et al. [8] compare several memory management techniques, including hazard pointers, reference counters, and so-called quiescent-state-based reclamation. In the latter, the memory can be reclaimed when each thread passes through at least one quiescent state [13], in which it does not hold any reference to shared nodes, and in particular, to nodes that have been removed from the data structure. In fact, the timestamp-based technique [7] discussed in the introduction can be seen as a special case of the quiescent-state approach. Hart et al. [8] find that when using hazard pointers or reference counters, expensive atomic instructions, such as fences and compare-and-swaps (CAS) executed for every traversed node, dominate the performance cost. Quiescent-state reclamation usually performs better, but it heavily depends on how often quiescent states occur. Moreover, if a thread fails before reaching the quiescent state, no memory can be safely reclaimed from that point.

Dragojevic et al. [5] consider how hardware transactional memory (HTM) can help to alleviate the performance and conceptual difficulties associated with memory management techniques. In contrast to [5], our algorithm does not rely on special hardware support, such as HTM.

The freezing idea was previously used in the context of concurrent data structures by Braginsky and Petrank in their recent work on chunk-based linked lists [3]. There, list nodes are grouped into chunks for better cache locality and list traversal performance. The freezing technique is used in [3] for list restructuring to notify threads that the part of the data structure they are currently using is obsolete. This is done by setting a special freeze-bit on pointers belonging to nodes in the obsolete part, making the pointers/nodes unsuitable for traversing. A thread that fails to use a frozen pointer realizes that this part of the data structure is obsolete and it restarts its operation, usually after helping to accomplish the list restructuring procedure that froze that part.

## 3. AN OVERVIEW OF DROP THE ANCHOR

As mentioned in the introduction, our technique relies on three building blocks, namely timestamps, anchors, and freezing. A thread $t$ manages a monotonically increasing

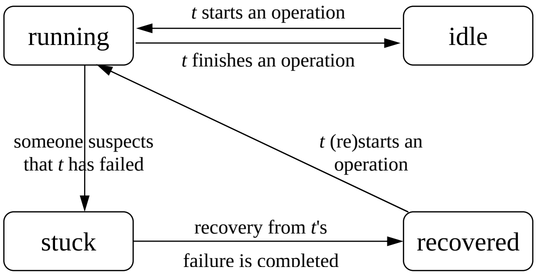

running
t starts an operation
idle
t finishes an operation
someone suspects that t has failed
t (re)starts an operation
stuck
recovery from t's failure is completed
recovered

**Figure 1: Transition diagram for possible states of the thread $t$**

timestamp in the following way. When $t$ starts its operation on a list data structure, it reads the timestamps of all threads and sets its timestamp to the maximal value it read plus one. When $t$ finishes its operation, it simply marks its timestamp as idle.

The timestamp of $t$ is associated with two flags, STUCK and IDLE. These flags specify one of the following states of the timestamp (and of the corresponding thread): *running* (both flags are turned off, meaning that $t$ has a pending operation on the data structure), *idle* (only the IDLE flag is on, meaning that $t$ does not have any pending operation on the data stucture), *running, but stuck*, which we call for brevity simply *stuck* (only the STUCK flag is on, meaning that $t$ with a pending operation is suspected by other thread(s) to be stuck) and *recovered* (both flags are turned on, meaning that other threads have frozen and copied the memory that might be accessed by $t$). The transition between these states is captured in Figure 1. Normally, $t$ moves between running and idle states. Once some thread suspects $t$ to be stuck, $t$'s timestamp is marked as stuck. The only way for $t$ to return to the running state is to go through the recovered state (cf. Figure 1).

The timestamps are also used to mark the insertion and deletion times of list nodes. That is, each node in the list has two additional fields, which are set as follows. When $t$ decides to insert (remove) a node into (from) the list, it sets the node's insertion (deletion, respectively) timestamp field to be higher by one from the maximal timestamp value that it observes among the timestamps of all threads.

The nodes deleted by $t$ are stored in $t$'s special reclamation buffer, which is scanned by $t$ once in a while (as in [14]). During each scan, and for each deleted node $n$, $t$ checks whether the deletion time of $n$ is smaller than the current timestamps of other threads (plus an additional condition described later), and if so, deallocates the node. This check ensures that all threads have started a new list operation since the time this node was removed from the list, and therefore, no thread can be viewing this node at this time.

If threads did not fail, this would be everything needed to manage the memory of non-blocking lists by a traditional epoch-based approach [7]. Unfortunately, thread failures may happen. In the design described so far, if a thread fails during its operation on the list, no additional node can be deallocated, since the timestamp of the failed thread would not advance.

To cope with the problem of thread failures, we use two additional concepts, namely anchors and freezing. Anchors are simply pointers used by threads to point to list nodes. In fact, hazard pointers[14] can be seen as a special case of anchors. The difference between the two is that the anchor is not dropped (set) before accessing every internal node in the list, but rather every ten, one hundred, or several thousands of node accesses (the frequency is controlled by the ANCHOR_THRESHOLD parameter). As a result, the amortized cost of anchor management is significantly reduced and spread across the traversal of (controllably) many nodes in the list data structure. The downside of our approach, however, is that when a thread $t$ is suspected of being stuck, other threads do not know for sure which object $t$ may access when (and if) it revives. They only know a range of nodes where $t$ might be, which includes the node pointed by $t$'s anchor plus additional nodes reachable from that anchored node. The suspecting threads use this range to recover the list from the failure of $t$. Specifically, they freeze all nodes in the range by setting the special *freeze-bit* of all pointers in these nodes[2]. Next, they copy all frozen nodes into a new sub-list. Finally, they replace the frozen nodes with the new sub-list and mark $t$'s timestamp as recovered. This mark tells other threads that the list was recovered from $t$'s failure. In other words, threads may again deallocate nodes they remove from the data structure, disregarding $t$'s timestamp.

The recovery procedure is relatively heavy performance-wise and has certain technical issues, but in return, the common path, i.e., the traversal of the data structure, incurs virtually no additional operations related to memory management. Since the recovery is expected to be very infrequent, we believe (and show in our performance measurements) that the complication associated with the recovery procedure pays off by eliminating the overhead in the common path. In the next section, we provide technical details of the application of this general idea into the concrete non-blocking implementation of the linked list.

# 4. DETAILED DESCRIPTION

## 4.1 Auxiliary fields and records

We use the singly linked list of Harris [7] as a basis for our construction. To support our scheme, each thread maintains two records where it stores information related to the memory management. The first record is global, i.e., it can be read and written by any thread (not just the owner of the record), and used to manage the thread's timestamp and anchor. The second record is local, and is used during object reclamations and for deciding whether the recovery procedure is necessary.

The structure of the records is given in Listing 1. The global record contains two fields, `timestampAndAnchor` and `lowTimeStamp`. The `timeStampAndAnchor` field contains the timestamp, the anchor, and the IDLE and STUCK bits of the thread, combined into one word so that all can be modified atomically. The width and the actual internal structure of the field depends on the underlying machine. In certain settings of 64-bit Linux-based architectures, the virtual memory addressing requires 48 bits; the two least significant bits in a pointer are typically zeroed due to memory alignment. Moreover, most existing architectures support wide-CAS instruction, which operates atomically on two adjacent memory words (i.e., 128 bits). In such settings, we allocate 64

[2] The freeze-bit is one of the least significant bits of a pointer, which are normally zeroed due to memory alignment.

**Listing 1: Auxiliary records**

```
struct GlobalMemoryManagementRec{
    uint128_t timeStampAndAnchor;
    uint64_t lowTimeStamp;
};

struct LocalMemoryManagementRec{
    list_t reclamationBuffer;
    uint64_t minTimeStamp;
    uint32_t minTimeStampThreadID;
    uint32_t minTimeStampThreadCnt;
};
```

bits for the timestamp and 64 bits for the anchor pointer, including two bits for two flags, which specify the state of the thread (i.e., running, idle, stuck and recovered). In the settings where only 64 bits can be a target for a CAS instruction, one can use 48 bits or fewer for the anchor pointer and, respectively, 16 bits for timestamp. However, different allocation techniques can be used to require fewer bits for the pointer.

When a thread $t$ accesses the list, it reads the timestamps of all threads in the system and sets its own timestamp to one plus the maximum among all the timestamp values that were read. It writes its new timestamp in the `timeStampAndAnchor` field, simultaneously setting the IDLE bit to zero. When $t$ completes its operation, it simply turns the IDLE bit on (leaving the same timestamp value). The exact details of the manipulation of this field are provided in subsequent sections.

In addition to the `timeStampAndAnchor` field, the global record contains a field called `lowTimeStamp`. This field is set by $t$ to the minimal timestamp observed by $t$ when it starts an operation on the list. As described in Section 4.4, the `lowTimeStamp` field is used by other threads when they try to recover the list from the failure of $t$ (to identify nodes that were inserted into the list before $t$ started its current operation).

The local record has four fields. The description of their role is given in Section 4.3.

Along with adding auxiliary records for each thread, we also augment each node in the linked list with two fields having self-explanatory names, `insertTS` and `retireTS`. These fields are set to the current maximal timestamp plus one when a node is inserted into or deleted from the list, respectively.

## 4.2 Anchor maintenance

Anchor maintenance is carried out when threads traverse the list, looking for a particular key. The simplified pseudocode for this traversal composes the `find` method given in the full version of this paper [2]. Recall that this method is used by all list operations in [7].

A thread counts the number of list nodes it has passed through and updates its anchor every ANCHOR_THRESHOLD nodes (where ANCHOR_THRESHOLD is some preset number). The anchor points to the first node in the list that can be accessed by the thread (which is the node pointed by `prev` in the `find` method). Anchor updates are made in the auxiliary `setAnchor` function also shown in [2]. An anchor update may fail for thread $t_i$ if some other thread $t_j$ has marked the `timeStampAndAnchor` field of $t_i$ as stuck, as explained in Section 4.4.

It is important to note that the actual update of the anchor is done with CAS (and not with a simple write operation) to avoid races with concurrently running threads that might suspect $t_i$ being stuck and try to set the STUCK bit in $t_i$'s `timeStampAndAnchor`. From a performance standpoint, however, the write operation of a hazard pointer, made on accessing every node, requires an expensive memory fence right after it [14, 8]. In contrast, the CAS in our approach is performed only every ANCHOR_THRESHOLD node accesses, and its amortized cost is negligible.

We note that the `find` function is allowed to traverse the frozen nodes of the list. A node is frozen if the second least significant bit in its `next` pointer is turned on (while the first least significant bit is used to mark the node as deleted [7]). If there is a need to update the `next` pointer of the frozen node, the update operation fails (as in [3]) and retries after invoking the `helpRecovery` method (pseudo-code can be found in [2]). As its name suggests, the latter method is used to help the recovery process of some stuck thread. This method is also called when a thread fails to update its anchor in `setAnchor`.

Finally, we note that at any time instant, list operations have references to at most two adjacent list nodes. (Recall that for the linked list data structure two hazard pointers are required [14]). As we require that a stuck thread will be able to access nodes only between its current anchor and (but not including) the next potential anchor, the ANCHOR_THRESHOLD parameter for the linked lists has to be at least 2.

## 4.3 Node reclamation

When a thread $t_i$ removes a node from the list, it calls the `retireNode` method, which sets the deletion timestamp of the node (i.e., the `retireTS` field) to the current maximal timestamp plus one. Then, similarly to [14], the `retireNode` method adds the deleted node to a reclamation buffer. The latter is simply a local linked list (cf. Listing 1) where $t_i$ stores nodes deleted from the list data structure, but not deallocated yet. When the size of the buffer reaches a predefined bound (controlled by the RETIRE_THRESHOLD parameter), $t_i$ runs through the buffer and deallocates all nodes with the retire timestamp smaller than the current minimal timestamp (plus an additional condition elaborated in Section 4.4). Note that if the deletion time of a node $n$ is smaller than the timestamp of a thread $t_j$, $t_j$ started its last operation on the list after $n$ was removed from the list; thus, $t_j$ will never access $n$. Obviously, if this holds for any $t_j$, it is safe to deallocate $n$.

When $t_i$ finds that some thread $t_j$ exists such that the timestamp of $t_j$ is smaller than or equal to the timestamp of one of the nodes in $t_i$'s reclamation buffer, $t_i$ stores the ID of that thread (i.e., $j$) in the `minTimeStampThreadID` field of its local memory management record (cf. Listing 1) and $t_j$'s timestamp in the `minTimeStamp` field of that record. It also sets the `minTimeStampThreadCnt` field to 1. It is important to note that if several threads have the same minimal timestamp, $t_i$ will store the smallest ID in `minTimeStampThreadID`. This will ensure that even if several threads are stuck with the same timestamp, all threads will consider the same thread in the recovery procedure.

On later scans of the reclamation buffer, if $t_i$ finds that the thread $t_j$ (whose ID is stored in $t_i$'s `minTimeStampThreadID`) still has the same timestamp, $t_i$ will increase the `minTimeStampThreadCnt` counter. Once the counter reaches the predefined RECOVERY_THRESHOLD parameter, $t_i$ will suspect that $t_j$ has failed and will start the recovery procedure described in Section 4.4.

## 4.4 Recovery procedure

The recovery procedure is invoked in one of the following three cases. First, it is invoked by a thread $t_i$ that tries to deallocate an object $n$ from its reclamation buffer, but repeatedly finds a running thread $t_j$ whose timestamp remains smaller than or equal to the timestamp of $n$ (cf. Section 4.3). The second case is when a thread $t_i$ tries to modify the `next` pointer of one of the nodes in the list, but finds that this node is frozen (cf. Section 4.2). Finally, the third case happens when a thread tries to update its anchor by modifying its `timeStampAndAnchor` field, but finds that some other thread turned the STUCK bit on in this field (cf. Section 4.2). In two last cases $t_i$ invokes the `helpRecovery` method. There, $t_i$ scans through global records of the threads, looking for a thread $t_j$ with the STUCK bit in $t_j$'s `timeStampAndAnchor` field turned on.

The recovery procedure consists of four phases (the code can be found in [2]). We explain these phases using the example in Figure 2. Assume that at some point in time the list data structure is in the state depicted in Figure 2(a), and thread $t_0$ decides to recover the list from the failure of thread $t_1$. Before invoking the first phase of the recovery procedure, $t_0$ stores locally the current value of $t_1$'s `timeStampAndAnchor` field. Then, in the first phase of the recovery procedure, $t_0$ attempts to modify $t_1$'s `timeStampAndAnchor` field by turning the STUCK bit on using CAS operation (cf. Figure 2(b)). If this operation fails, $t_0$ rereads $t_1$'s `timeStampAndAnchor` field and checks whether it was marked as stuck by some other thread. If not, it aborts the recovery procedure (since either $t_1$ is actually alive and has modified its `timeStampAndAnchor` field, or some other thread, i.e., $t_2$, has finished the recovery of $t_1$ and, as we will see later, turned both STUCK and IDLE bits on). Otherwise, if the CAS operation that turns the STUCK bit on succeeds, or if it fails, but $t_1$ is marked as stuck by another thread, $t_0$ proceeds to the second phase.

In the second phase of the recovery procedure, $t_0$ freezes and copies all nodes that $t_1$ might access if $t_1$ revived and traversed the list until realizing at the next anchor update that its anchor is marked as stuck. To identify such nodes, $t_0$ extracts $t_1$'s anchor pointer out of the value stored in $t_1$'s `timeStampAndAnchor` field (which points to node 25 in our example in Figure 2(a)). Then, $t_0$ starts setting the freeze-bit in the `next` pointers of reachable nodes, starting from node 25. It copies the frozen nodes (with the freeze-bit set off) into a new list. Note that some of the nodes may already be deleted from the list (e.g., node 25, 27 and 42 in Figure 2), but not disconnected or reclaimed yet. Such nodes are frozen, but they do not enter the new copied part of the list. The thread $t_0$ keeps freezing and copying until it passes through ANCHOR_THRESHOLD nodes having an insertion timestamp smaller than the value of $t_1$'s `lowTimeStamp` field. In our example in Figure 2, let us assume that these are nodes 25, 27, 40, 41 and 42. Note that for traversing those nodes, $t_0$ had to update its own anchor to be the same as $t_1$'s anchor in order to handle $t_0$'s failure during the recovery procedure. At the end of the second phase the list looks as depicted in Figure 2(c). The pseudo-code for how we freeze the nodes and create the copies can be found in [2].

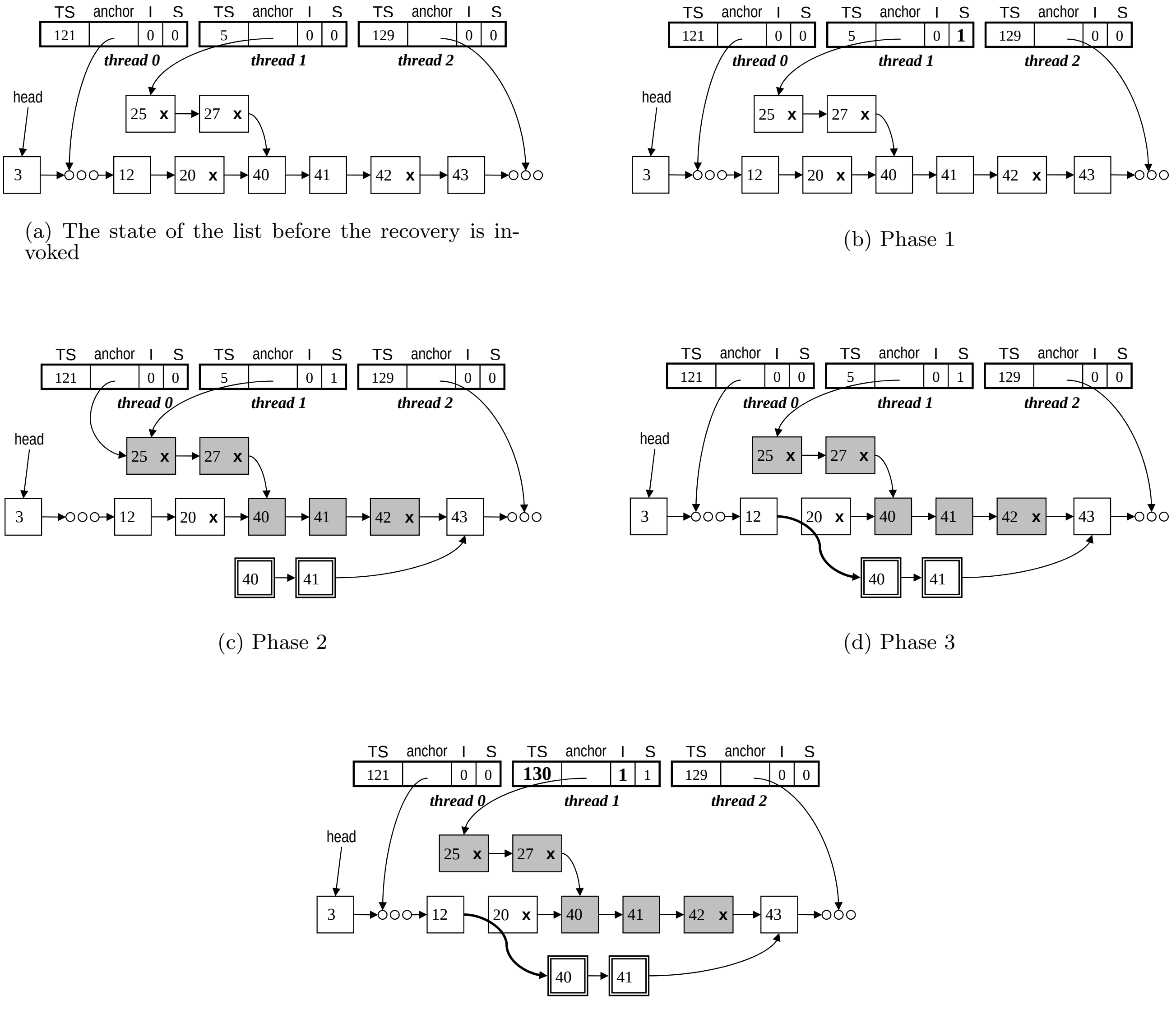
In the third phase, $t_0$ attempts to replace the frozen nodes with a locally copied part of the list. To this end, it runs from the beginning of the list data structure and looks for the first (not-deleted) node $m$ whose `next` pointer either points to the not-deleted frozen nodes or it is followed by a sequence of one or more deleted nodes such that the `next` pointer of the last node in the sequence points to a not-deleted frozen node (in Figure 2(c), $m$ is the node 12). If such $m$ were not found by reaching the end of the list data structure, $t_0$ would finish this phase, as it would assume that some other thread has replaced the frozen part of the list with the new list created by that thread. Otherwise, $t_0$ attempts to update $m$'s `next` pointer to point to the corresponding copied node in the new list. If it fails, it restarts this phase from the beginning. Otherwise, $t_0$ inserts all nodes between $m$ and the first frozen node (i.e., node 20 in Figure 2(c)) into its reclamation buffer in order to deallocate them later, bringing the list to the state exhibited in Figure 2(d). The code of the procedure for replacing frozen nodes can also be found in [2].

One subtlety that is left out of the code for lack of space, is the verification that the new local list indeed matches the frozen nodes being replaced. It is crucial to ensure that if a thread running the recovery procedure gets delayed, it does not replace another frozen part of the list when it resumes. To this end, we record the sources of the new nodes, when they are copied, and CAS the new list into the data structure only if it replaces the adequate original nodes that can be found in the recorded sources.

In the final, fourth phase, $t_0$ sets the IDLE bit in the $t_1$'s `timeStampAndAnchor` field, marking $t_1$ as recovered. Additionally, $t_0$ promotes $t_1$'s timestamp, recording the (logical) time when $t_1$ was recovered (cf. Figure 2(e)). Note that $t_0$ does not need to check whether its CAS has succeeded, since if it hasn't, some other thread has performed this operation. We denote a timestamp of a thread with IDLE and STUCK flags turned on as *recovery timestamp*.

## 4.5 The refined reclamation procedure

Thread $t_1$, considered stuck, might actually have a pointer to a node $n$ which is already not a part of the list. For instance, in the state of the list shown in Figure 2(a), $t_1$ might be stopped while inspecting node 25 (or 27). If this node is currently in the reclamation buffer of some other thread $t_k$ (i.e., $t_0$ or $t_2$), and if $t_k$ does not consider $t_1$ after the recovery is done (i.e., $t_k$ only checks that node 25's `retireTS` is smaller than the timestamp of any *running* thread), $t_k$ might deallocate node 25 and $t_1$ might erroneously access this memory if and when it revives. Note that node 27 may already be unreachable from the node pointed by $t_1$'s anchor by the time of $t_1$'s recovery, if, e.g., the next pointer of node 25 was updated while $t_1$ was inspecting node 27. In this case, node 27 will not be frozen and copied at all. In order to cope with such situations, before deallocating a node we require its retire timestamp (`retireTS`) to be larger than the timestamp of any thread in the *recovered* state (in addition to being smaller than the timestamp of any running thread). This way we prevent nodes removed from the list



(a) The state of the list before the recovery is invoked

(b) Phase 1

(c) Phase 2

(d) Phase 3

(e) Phase 4

**Figure 2: Recovery phases. Nodes marked with 'x' are deleted, i.e., the delete-bit of their `next` pointer is turned on [7]. Shaded nodes are frozen, i.e., the freeze-bit of their `next` pointer is turned on.**

before some thread got recovered from being deallocated, as the thread being recovered might hold a pointer to such node. When (and if) the recovered thread becomes running again, it will be possible to reclaim those nodes. To summarize, when a thread wants to deallocate a node $n$, it checks that $n$ is not frozen (i.e., the freeze-bit in its `next` pointer is not set) and that the following condition holds:

$$\text{MAX}(\{\text{timestamp of } t_x \mid t_x \text{ is recovered}\}) < \text{n.retireTS} < \text{MIN}(\{\text{timestamp of } t_x \mid t_x \text{ is running}\})$$

It should be noted that when calculations of the retire timestamp and the recovery timestamp are done simultaneously (by different threads), the retire timestamp can erroneously be higher than the recovery timestamp, and wrong reclamation can happen. Therefore, when calculating the retire timestamp for a node, we require a thread to pass twice over the timestamps of the threads verifying that no thread was marked stuck or recovered concurrently. If such thread(s) is found, the node is inserted into the reclamation stack as frozen.

For simplicity of presentation, in the algorithm described above frozen nodes are not reclaimed. Such nodes can only appear if threads fail and such a solution may be acceptable. However, frozen nodes can be easily reclaimed for recovered threads that have resumed operation. A recovered thread can reclaim nodes according to the recovery timestamps. Also, a frozen node that appears in a reclamation stack can be reclaimed using its `retireTS` field and the `lowTimeStamp` field of all stuck threads. Details are omitted.

## 4.6 Correctness argument

In this section we give high-level arguments behind the proofs of correctness. We start by outlining the assumed memory model and defining linearization points for the modified list operations. Then we argue that any internal node deleted after the last recovery was finished (or deleted any time if no thread has been suspected being stuck) will be eventually reclaimed (we call this property *eventual conditional reclamation*). Next, we argue that our technique guarantees the safety of memory references. In other words, no thread $t$ accesses the memory that has been reclaimed since the time $t$ obtained a reference to it. Finally, we argue that our technique is non-blocking, meaning that whenever a thread $t$ starts the recovery procedure, then after a finite number of $t$'s steps either $t$ completes the recovery, or some other thread completes an operation on the list. In addition, we show that the system-wide progress with respect to the list operations is preserved, that is after a finite number of completed recovery procedures there is at least one completed list operation.

Due to space limitation, the proof sketch of all lemmas appears in the full version of this paper [2].

### 4.6.1 Model and linearizability

Our model for concurrent multi-threaded computation follows the linearizability model of [12]. In particular, we assume an asynchronous shared memory system where $n$ deterministic threads communicate by executing atomic operations on some finite number of shared variables. Each thread performs a sequence of steps, where in each step the tread may perform some local computation or invoke a single atomic operation on a shared variable. The atomic operations allowed in our model are reads, writes, or compare-and-swaps (CAS). The latter receives a memory address of a shared variable $v$ and two values, *old* and *new*. It sets the value of $v$ to *new* only if the value of $v$ right before CAS is applied is *old*; in this case CAS returns *true*. Otherwise, the value of $v$ does to change and CAS returns *false*. We assume that each thread has an ID, denoted as tid, which is a value between 0 and $n-1$. In systems where tid may have values from arbitrary range, known non-blocking renaming algorithms can be applied (e.g., [1]). In addition, we assume each thread can access its tid and $n$.

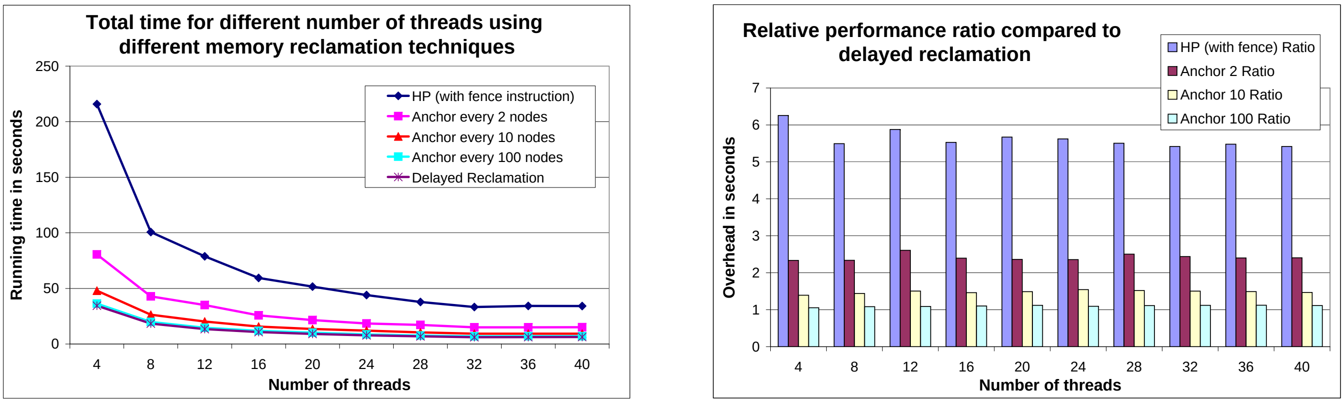
The original implementation of all operations of the non-blocking linked list by Harris [7] is linearizable [12]. We argue that after applying Drop the Anchor memory management technique, all list operations remain linearizable. Recall that all list operations invoke the `find` method, which returns pointers to two adjacent nodes, one of which holds the value smaller than the given key. For further details, see [7]). Denote this node as *prev*. Furthermore, recall that list operations may invoke `find` several times. For instance, `insert` will invoke `find` again if the `next` pointer of *prev* has being concurrently modified (in particular, in our case, frozen). Thus, we define the linearization points for a list operation *op* with respect to the *prev* returned from the last invocation of `find` by *op*. If this *prev* node is not frozen (i.e., the freeze-bit of its `next` pointer is not set), the linearization point of *op* is exactly as in [7]. However, if this *prev* node is frozen, we set the linearization point of *op* at the time instance defined as following. Consider the sequence of frozen nodes read by the corresponding `find` operation starting from a frozen node $m$ and including the (frozen) node *prev* (where $m$ and *prev* might be the same node). The linearization point of *op* is defined at the latest of the two events: (a) the corresponding `find` traversed $m$ (i.e., read the `next` pointer of the node previous to $m$ in the list) and (b) the latest time at which some node between (and including) $m$ and *prev* was inserted or marked as deleted. The intuition is that when `find` returns a result from a frozen part of the list, this part no longer reflects the actual state of the list at the moment *prev* node is read. Thus, we have to linearize the corresponding operation at some earlier time instance, at which the nodes read by `find` are still consistent with the actual keys stored in the list.

### 4.6.2 Eventual conditional reclamation

LEMMA 4.1. *Let $T_s$ and $T_f$ be the time when a thread $t$ starts and finishes, respectively, the call to `retireNode(node)` and $T_r > T_f$ is the time when $t$ finishes to scan its reclamation buffer. Then at least one of the following events occurs in the time interval $[T_s, T_r]$:*

1. *Some thread remains running throughout $[T_f, T_r]$, and its timestamp changes at most once in $[T_f, T_r]$.*
2. *Some thread becomes recovered at some point in time in $[T_s, T_r]$.*
3. *The memory allocated to `node` is reclaimed by the time $T_r$.*

Based on the lemma above, we prove that when a thread removes a node from the list, as long as that tread keeps applying (delete) operations on the list and particularly "bad" things do not happen to other threads (e.g., they are not suspected to be failed), the memory of that node will be eventually reclaimed.



(a) The total running time comparison for searches only.

(b) Memory management overhead referred to delayed reclamation for searches only.

**Figure 3: Drop the Anchor vs. Hazard Pointers for lists with the initial size of 100k keys, the read-only workload results.**

Lemma 4.2. *Let $T_s$ be the time when a thread $t$ starts the call to* `retireNode(node)`*. Then* `node` *will be eventually reclaimed as long as $t$ keeps removing nodes from the list and there is no thread that is stuck or recovered at or after $T_s$.*

Note that even if some thread $t_x$ gets stuck or recovered after $T_s$ as above, it may have impact only on nodes being removed *before* (or concurrently to) $t_x$'s recovery.

### 4.6.3 Safety of memory references

First, we prove that access to any node that can be reached from $t$'s anchor (for any thread $t$) is safe, i.e., such node cannot be reclaimed.

Lemma 4.3. *No node reachable from an anchor of some thread can be reclaimed.*

Using the lemma above, we show that with the Drop the Anchor memory management, no thread will access a reclaimed memory.

Lemma 4.4. *No thread $t$ accesses the memory that has been reclaimed since the time $t$ obtained a reference to it.*

### 4.6.4 Progress guarantees

The original implementation of all operations of the non-blocking linked list by Harris [7] is lock-free [12] . We argue that after applying Drop the Anchor memory management technique, all list operations remain lock-free. We say that a thread $t_i$ *starts the recovery* of a thread $t_j$ when $t_i$ sets the stuck bit on in $t_j$'s `timestampAndAnchor` field. Similarly, we say a thread $t_i$ *completes the recovery* of a stuck thread $t_j$ when $t_i$ sets the idle bit on in $t_j$'s `timestampAndAnchor` field.

Lemma 4.5. *If a thread $t_i$ starts the recovery of $t_j$ at $T_s$, then the recovery of $t_j$ will be completed at $T_f > T_s$ (by possibly another thread $t_k$) and/or infinitely many list operations will be linearized after $T_s$.*

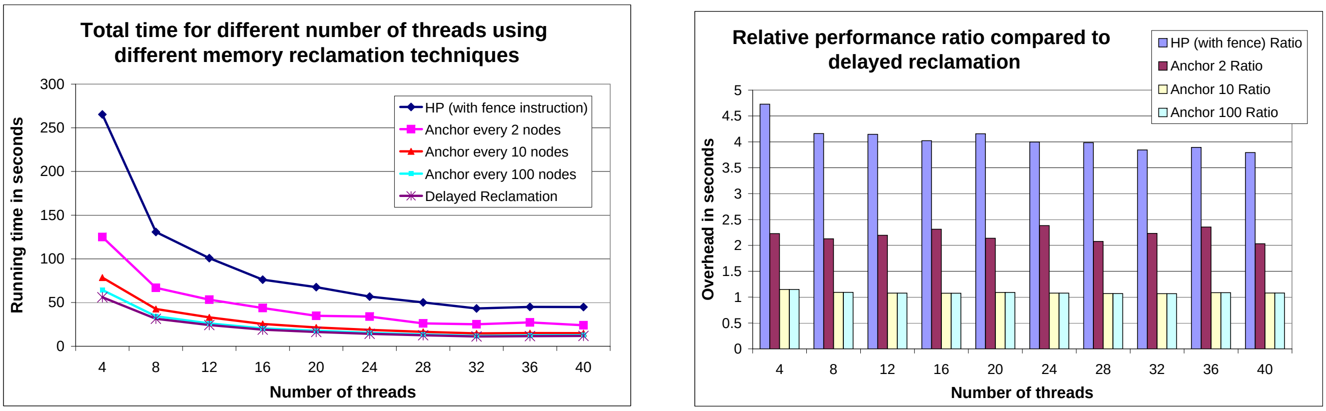
Next, we show that despite recovery operations, the system-wide progress is preserved, i.e., threads never keep recovering one another forever without completing list operations.

Lemma 4.6. *Consider $n+1$ recovery operations completed at times $T_1 < T_2 < ... < T_{n+1}$. Then there must be at least one list (delete) operation linearized in the time interval $[T_1, T_{n+1}]$.*

## 5. PERFORMANCE EVALUATION

We have implemented the non-blocking linked list data structure of Harris [7] with several memory management techniques. First, we have implemented the Hazard Pointers technique following the pseudo-code presented in [14], but with the additional memory fence instruction added just after the write of a new value to the hazard pointer of a thread [8]. Second, we have implemented our new Drop the Anchor technique presented in this paper. Finally, we have also implemented a simple technique, where nodes removed by a thread $t$ from the list are added to $t$'s reclamation stack and reclaimed later once 64 nodes are collected in the reclamation stack. We refer to this implementation as *delayed reclamation*. We note that this scheme is incorrect in a sense that it allows threads to access deallocated memory, but we used this implementation to represent a memory management scheme with a minimal performance impact. All our implementations were coded in C and compiled with -O3 optimization level.

We have run our experiments on the machine with two AMD Opteron(TM) 6272 16-core processors, operated by Linux OS (Ubuntu 12.04). We have varied the number of threads between 1 and 40, slightly above the number of threads that can run concurrently on this machine (32). If not stated otherwise, each test starts by building an initial list with 100k random keys. After that, we measure the total time of 320k operations divided equally between all threads. The keys for searches and insertions are randomly chosen 20-bit sized keys. For deletion operations, we ensure that randomly chosen keys actually exist in the list in order to make the reclamation process substantial. The values of recovery_threshold and retire_threshold were always 64. All threads are synchronized to start their operations immediately after the initial list is built and we



(a) The total running time comparison for 20% inserts, 20% deletes and 60% searches.

(b) Memory management overhead referred to delayed reclamation for 20% inserts, 20% deletes and 60% searches.

**Figure 4: Drop the Anchor vs. Hazard Pointers for lists with the initial size of 100k keys, the mixed workload results.**

measure the time it takes to complete all operations by all threads. We run each test 10 times and present the average results. The variance of all reported results is below 1.5% of the average.

Figures 3 and 4 show the measurements of the total time required to complete our benchmark using the HP memory management, the delayed reclamation and the Drop the Anchor method. For the latter, we have used three versions with different values for the ANCHOR_THRESHOLD value. Specifically, in the first version the anchor is dropped every second node (which is the lowest legitimate value for ANCHOR_THRESHOLD in the case of the linked list), in the second version the anchor is dropped every 10 nodes, and in the third – every 100 nodes. We show the results for read-only workload where all operations are searches (Figure 3(a)) and for the mixed workload, where 20% of all operations are insertions, 20% are deletions, and the remaining 60% are searches (Figure 4(a)).

Our measurements show that in the mixed workload the Drop the Anchor-based implementation is faster in about 150–250% than the HP-based one, even if the anchor is dropped every second node. When increasing the ANCHOR_THRESHOLD parameter from 2 to 100, we get even higher improvement of 300–450% over the performance of HPs.

In read-only workload we can see even better performance improvement (400% on average) due to anchors usage compared to HP usage (cf. Figure 3(a)). Finally, we can see that for substantial amount of threads, the Drop the Anchor-based linked list performance is very close to the linked list implementation based on the simple delayed reclamation. This suggests that the amortized cost of the memory management in the Drop the Anchor technique is very small.

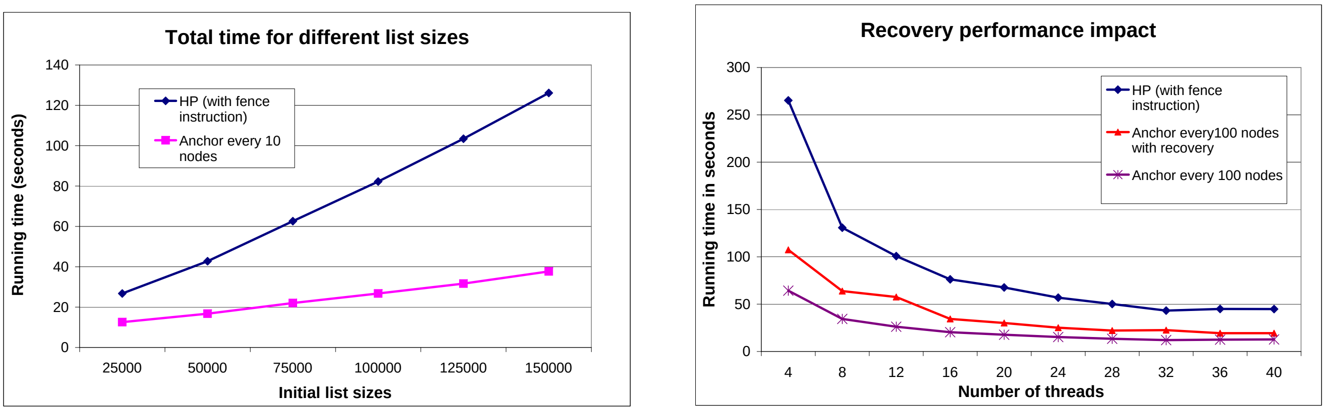
Additionally, Figures 3(b), 4(b) present the relative performance ratio of each memory management technique, explained above, compared to delayed reclamation. When the ratio is close to 1 it means that the memory management technique adds almost no overhead over the delayed reclamation. The HP memory management shows 400–550% slowdown, where Anchor-based implementation shows 7–10% slowdown for anchors dropped every 100 nodes, and 200–250% slowdown for anchors dropped every 2 nodes, all compared to delayed reclamation results.

In another set of experiments, we measure the impact of the initial size of the list on the performance of the HP-based and Drop the Anchor-based implementations, while the number of threads is constant (16) and the workload is mixed. The results are depicted in Figure 5(a). It can be seen that the running time of both implementations increases linearly with the size of the list as threads need to traverse more nodes per operation on average. The slope of the HP-based implementation is much steeper, however, suggesting that the overhead introduced by fences is much more significant than the cost of the anchor management.

In Figure 5(b) we can see the performance impact of the recovery procedure in the Drop the Anchor technique. We use the version of the technique with the ANCHOR_THRESHOLD value equals 100 for more significant impact. We explicitly delay one of the threads, thus causing this thread to be considered as stuck and recovered by other threads. The stuck thread returns to run after 2 seconds and the presented total time is measured until all threads finish their runs. The results show that the recovery procedure has 15–50% impact on the performance, even when the ANCHOR_THRESHOLD value is high. In any case, the Anchors-based implementation's performance (with the delay and recovery) is much better than the HP-based one.

## 6. DISCUSSION

We presented a new method for memory management of non-blocking data structures called *Drop the Anchor*. Drop the Anchor is a novel combination of the time-stamping method (which cannot handle thread failures) with the anchors and freezing techniques that provide a fallback allowing reclamation even when threads fail. Non-blocking algorithms must be robust to thread failures and so coping with thread failures in the memory manager is crucial. We have applied Drop the Anchor for the common non-blocking linked list implementation and compared it with



(a) Drop the Anchor vs. Hazard Pointers when 16 threads are run on lists with different initial sizes. In the Anchor-based implementation, the anchor is dropped every 10 nodes.

(b) The impact of the recovery on the performance of lists with the initial size of 100k keys.

**Figure 5: Drop the Anchor vs. Hazard Pointers for lists with the different initial sizes and the recovery performance impact.**

the standard Hazard Pointers method. Measurements show that Drop the Anchor drastically reduces the memory management overhead, while robustly reclaiming objects in all executions.

We believe our technique can be applied for other non-blocking data structures. Specifically, assume a data structure represented by a directed graph, where vertices correspond to internal nodes and edges correspond to pointers between these nodes. When recovering a thread $t$, we need to freeze and copy the sub-graph containing all internal nodes at the distance that depends on the ANCHOR_THRESHOLD parameter, from the node pointed by $t$'s anchor. Essentially, although the copying operation might be expensive and even involve the whole data structure, the scalability bottleneck associated with the memory fences will be removed from the common node access step.

## 7. ACKNOWLEDGMENTS

We want to thank Tim Harris for his helpful comments on the earlier version of this paper.

## 8. REFERENCES

[1] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.

[2] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures (full version). http://www.cs.technion.ac.il/ erez/Papers/-DropTheAnchorFull.pdf.

[3] A. Braginsky and E. Petrank. Locality-conscious lock-free linked lists. In *ICDCN*, pages 107–118, 2011.

[4] D. Detlefs, P. A. Martin, M. Moir, and G. L. Steele. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[5] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In *PODC*, pages 99–108, 2011.

[6] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *TPDS*, 20(8):1173–1187, 2009.

[7] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.

[8] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[9] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[10] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[12] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[13] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *PDCS*, pages 509–518, 1998.

[14] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *TPDS*, 15:491–504, 2004.

[15] H. Sundell. Wait-free reference counting and memory management. In *IPDPS*, 2005.

[16] J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, pages 214–222, 1995.