Fast and Robust Memory Reclamation for Concurrent Data Structures

[Technical Report]

Oana Balmau
EPFL

Rachid Guerraoui
EPFL

Maurice Herlihy
Brown University

Igor Zablotchi
EPFL

ABSTRACT

In concurrent systems without automatic garbage collection, it is challenging to determine when it is safe to reclaim memory, especially for lock-free data structures. Existing concurrent memory reclamation schemes are either fast but do not tolerate process delays, robust to delays but with high overhead, or both robust and fast but narrowly applicable.

This paper proposes QSense, a novel concurrent memory reclamation technique. QSense is a hybrid technique with a fast path and a fallback path. In the common case (without process delays), a high-performing memory reclamation scheme is used (fast path). If process delays block memory reclamation through the fast path, a robust fallback path is used to guarantee progress. The fallback path uses hazard pointers, but avoids their notorious need for frequent and expensive memory fences.

QSense is widely applicable, as we illustrate through several lock-free data structure algorithms. Our experimental evaluation shows that QSense has an overhead comparable to the fastest memory reclamation techniques, while still tolerating prolonged process delays.

1. INTRODUCTION

1.1 The Problem

Any realistic application requires its data structures to grow and shrink dynamically and hence to reclaim memory that is no longer being used. For the foreseeable future, many high-performing applications, such as operating systems and databases, will be written in languages where programmers manage memory explicitly (such as C or C++). There is thus a clear need for concurrent data structures that scale *and* efficiently allocate/free memory. Designing such data structures is however challenging, as it is not clear when it is safe to free memory, especially when locks are prohibited (lock-free constructions [11, 18]).

To illustrate the difficulty, consider, as illustrated in Figure 1, two processes p_1 and p_2 concurrently accessing a linked list of several nodes. Process p_1 is reading node n_1, while p_2 is concurrently removing node n_1. Assume p_2 unlinks n_1 from the list. Then, p_2 needs to decide whether n_1's memory can be freed. If p_2 were allowed to block p_1 and inspect p_1's references, then it could easily determine whether freeing n_1 is safe. But in a lock-free context, p_2 has a priori no way of knowing if p_1 is still accessing n_1 or not. So, if p_2 goes ahead and frees node n_1, it triggers an illegal access next time p_1 tries to use n_1.

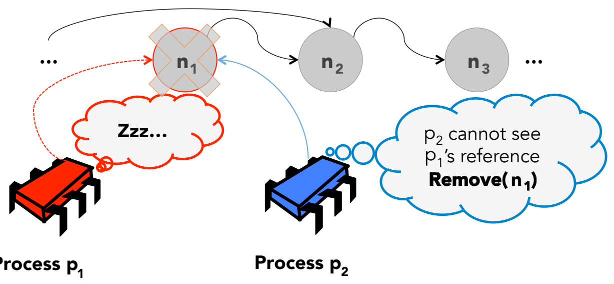

Figure 1: The concurrent memory reclamation problem.

1.2 The Trade-off

Various approaches have been proposed to address the issue of concurrent, programmer-controlled memory reclamation for lock-free data structures. *Hazard pointers* (HP) [25] (which we recall in more details in § 3) is perhaps the most widely-used method. Basically, the programmer publishes the addresses (hazard pointers) of nodes for as long as they cannot be safely reclaimed. A reclaiming process must ensure that a node it is about to free is not marked by any other process. The hazard pointer methodology holds two important advantages: (1) it is wait-free and (2) it is applicable to a wide range of data structures. However, hazard pointers also have a notable drawback: as we explain in § 3.2, they require a memory fence instruction for every node traversed in the data structure. This can decrease performance by up to 75% (as we will see in § 7).

Most memory reclamation techniques that seek to overcome the performance penalty of hazard pointers have been analyzed in terms of amortized overhead [1, 5, 6, 14]: the overhead of reclamation operations is spread across several node accesses or across several operations, thus considerably reducing their impact on performance. *Quiescent State Based Reclamation* (QSBR) (which we recall in § 3), is among the most popular schemes applying the amortized overhead principle [6, 14]. QSBR is fast and can be applied to virtually any data structure. However, QSBR is *blocking*: if a process is delayed for a long time (a process failure is a particular type of delay), an unbounded amount of memory might remain unreclaimed. As such, using QSBR with lock-free data structures would negate one of the main advantages of lock-freedom: *robustness* to process delays.

There have indeed been several proposals for achieving both lock-freedom and low amortized overhead [5, 6]. Yet, these are ad-hoc methods that apply only to certain well-chosen data structures. They require significant effort to be adapted to other data structures (as we discuss in § 8).

Overall, the current prevalent solutions for concurrent mem-

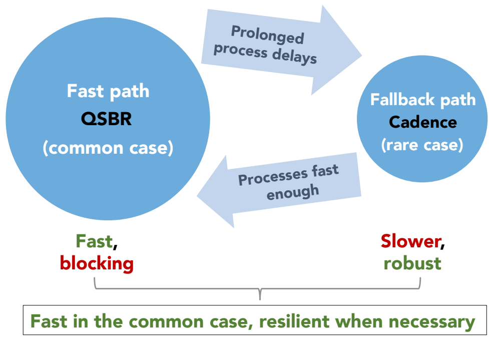

Figure 2: A high-level view of QSense.

ory reclamation are either *wait-free* but *with high overhead*, *fast* but *blocking*, or *ad-hoc*, lacking a clear and systematic methodology on how to apply them.

1.3 The Contributions

We design, implement and evaluate *QSense*, a novel technique for concurrent memory reclamation that is, at the same time, easily applicable to a wide range of concurrent data structures (including lock-free ones), robust, and fast.

QSense uses a hybrid approach to provide fast and robust memory reclamation. Figure 2 depicts a high-level view of QSense. In the common case (i.e. when processes do not undergo prolonged delays), the fast QSBR scheme is employed. A delay could be caused, for instance, by cache misses, application-related delays, or being descheduled by the operating system. By *prolonged delay*, we refer to a delay of a process p_1 that is long enough such that a large number of nodes (larger than a given configurable threshold) are removed but cannot be safely reclaimed by another concurrent process p_2. If prolonged process delays are detected, QSense automatically switches to a fall-back memory reclamation scheme that is robust. When all processes are active again (no more prolonged delays), the system automatically switches back to the fast path. By *robustness* we mean that any process performing operations on the data structure (called *worker process*) will finish any action related to memory reclamation within a bounded number of steps, regardless of the progress of other worker processes. To guarantee this progress, we require certain timing assumptions about a set of auxiliary background processes, that do not participate in the actual data structure operations (all assumptions are discussed in § 5).

The fall-back path consists of a subprotocol we call *Cadence*, a novel amortized variant of the widely-used hazard pointer mechanism. Cadence overcomes the necessity for per-node memory barriers during data structure traversal, significantly increasing performance. We achieve this through two new concepts. The first is *rooster processes*: background processes that periodically wake up and generate context switches, which act as memory barriers. The periodic context switches ensure that any hazard pointer becomes visible to other processes within a bounded time T. The second concept is *deferred reclamation*: a process p only reclaims a removed node n after n has been awaiting reclamation for longer than T. This ensures any hazard pointer potentially protecting n must be visible. Therefore, n's memory can be safely freed provided no hazard pointers are protecting n. Cadence can be used either as part of QSense or as a stand-alone memory reclamation scheme.

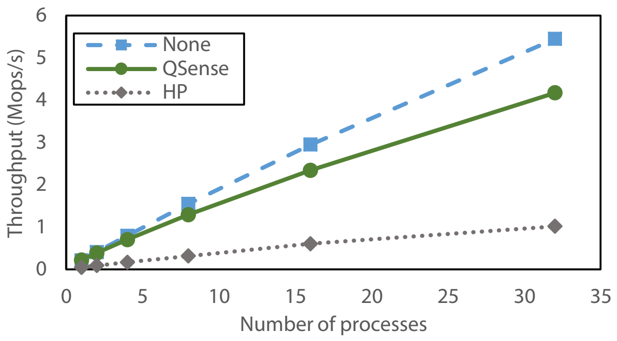

Figure 3: QSense, HP and no reclamation on a linked list of 2000 elements, with a 10% updates workload.

QSense requires minimal additions to the data structure code, consisting only of a number of calls to functions provided by the QSense interface. We present a simple set of rules that determine where to place these calls. QSense communicates with data structures through three functions: `manage_qsense_state`, `assign_HP` and `free_node_later`. The rules concerning where to call these three functions are:

1. Call `manage_qsense_state` in states where no references to shared objects are held by the processes (usually in between data structure operations).
2. Call `assign_HP` before using a reference to a node so as to inform other processes that the node's memory should not be reclaimed yet.
3. Call `free_node_later` whenever a node is removed from the data structure (where `free` would be called in a sequential setting).

We show empirically that QSense achieves good performance in a wide range of scenarios, while providing the same progress guarantees as a hazard pointer based scheme. Our experiments in § 7 show that QSense achieves a 29% overhead on average over leaky implementations of a lock-free concurrent linked list, a skip list and a binary search tree. Moreover, QSense outperforms the popular hazard pointers technique by two to three times. To illustrate this point, Figure 3 shows sample results from our experiments, comparing QSense to hazard pointers and no memory reclamation (leaky implementation) on a concurrent linked list.

Roadmap. The rest of this paper is organized as follows. In § 2, we pose the problem. In § 3, we recall prior work that inspired QSense. In § 4, we give an overview of QSense. Then, in § 5, we dive into the details of Cadence and detail the assumptions needed for its correctness. In § 6, we prove safety and liveness properties of QSense. In § 7, we compare QSense's performance against that of popular memory reclamation schemes. Finally, in § 8, we describe related work and we conclude in § 9.

2. MODEL AND PROBLEM DEFINITION

We consider a set of n processes that communicate through a set of shared memory locations using primitive memory access operations. A *node* is a set of memory locations that can be viewed as a logical entity. A *data structure* consists of one or more fixed nodes that can always be accessed directly by the processes, called *roots*, and the set of nodes that are reachable by following pointers from the roots.

2.1 Node States

At any given time, a node can be in one of five states [25]: (1) *Allocated* — the node has been allocated by a process, but not yet inserted into the data structure. (2) *Reachable* — the node is reachable by following pointers from the roots of the data structure. (3) *Removed* — the node is no longer *reachable*, but may still be in use by some processes. (4) *Retired* — the node is removed and cannot be used by any process, but is not yet *free*. (5) *Free* — the node's memory is available for allocation.

2.2 The Memory Reclamation Problem

We can now state the *memory reclamation problem* as follows: given some *removed* nodes, make them available for re-allocation (i.e. change their state to *free*) after it is no longer possible for any process (except the reclaiming process) to access them (i.e. after they have become *retired*).

The problem is distinct from garbage collection. Here, we are only concerned with the reclamation of memory used by data structure nodes, and not the reclamation of arbitrary memory regions. Moreover, the nodes whose memory needs to be reclaimed are expressly marked by the programmer after they have been explicitly unlinked from the data structure and are no longer reachable.

2.3 Terminology

Safe: A node n is *safe* [25] for a process p if: (1) n is *allocated* and p is the process that allocated it, or (2) n is *reachable*, or (3) n is *removed* or *retired* and p is the process that removed n from the data structure.

Possibly unsafe: A node n is *possibly unsafe* [25] for a processes p if it is impossible, using only knowledge of p's private variables and the semantics of the data structure algorithm used, to positively establish that n is *safe* for p. Informally, a node is *possibly unsafe* if it is impossible to determine using only local process data and knowledge of the data structure algorithm whether accessing n will trigger an access violation (e.g. if n has been reclaimed in the meanwhile).

Access hazard: An *access hazard* [25] is a step in the algorithm that might result in accessing a *possibly unsafe* node for the process that is executing the algorithm.

Hazardous reference: A process p holds a *hazardous reference* [25] to a node n if one of p's private variables holds n's address and p will reach an *access hazard* that uses n's address. Informally, a *hazardous reference* is an address that will be used later in a hazardous manner (i.e. to access *possibly unsafe* memory) without further verification of safety.

3. BACKGROUND

In this section we recall quiescent state based reclamation and hazard pointers, the techniques QSense builds upon.

3.1 Quiescent State Based Reclamation

Quiescent State Based Reclamation (QSBR) is a technique which emerged in the context of memory management in operating system kernels [4]. Quiescence-based schemes are fast, outrunning popular pointer-based schemes under a variety of workloads [14]. Therefore, in QSense, we chose a quiescence technique for the fast path. Though not a contribution of this paper, QSBR is detailed below, for completeness. QSBR makes use of *quiescent states* (at process level) and *grace periods* (at system level).

A process is in a *quiescent state* if it does not hold references to any shared objects in the data structure. Quiescent states need to be specified at the application level. Typically, a process is in a quiescent state whenever it is in between operations (read/insert/delete). In practice, quiescent states are declared after processes have finished a larger number of operations — called the *quiescence threshold* — as batching operations in this way boosts performance.

A *grace period* is a time interval in the execution during which each worker process in the system goes through at least one quiescent state. If the time interval $[a, b]$ is a grace period, after time b no process holds hazardous references to nodes that were removed before time a. The occurrence of grace periods is managed through an epoch-based technique [6, 14]. At every step of the execution, every process is in one of three logical epochs. Each process has three lists in which removed nodes are stored, called *limbo lists* (one per epoch). If a node n has been removed when a process p was in epoch i, n will be added to p's i^{th} limbo list. Each process keeps track of its local epoch and all processes have access to a shared global epoch. When a process p declares a quiescent state, it does the following. If p's local epoch e_p is different than the global epoch e_G, then p updates e_p to e_G. Else, if all processes have their local epoch equal to e_G (including p), p increments e_G by 1.

The Problem of Robustness in QSBR. The main advantage of QSBR is the low overhead. Nevertheless, its lack of robustness to significant process delays makes its out of the box use unsuitable for some practical applications. As we show in § 7, if achieving a grace period is no longer possible or takes a significant amount of time, the system might run out of memory and eventually block. In § 5, we show how we address the problem of resilience to prolonged process delays in QSense.

3.2 Hazard Pointers (HP)

The main idea behind the hazard pointers scheme [25] is to associate with each process a number of single-writer multi-reader pointers. These pointers — called *hazard pointers* — are used by processes to indicate which nodes they might be about to access without further validation. The nodes marked by hazard pointers are unsafe to reclaim.

The hazard pointer scheme mainly consists of *node marking* and *node reclamation. Node marking* is the assignment of hazard pointers to nodes. This ensures that the reclaiming process can discern which of the nodes that were removed from the data structure are *retired* (and thus safe to reclaim). *Node reclamation* is a procedure that makes nodes available for re-allocation. Every time a process removes a node from a data structure, the process adds a reference to the node in a local list of removed nodes. After a given number of node removals, each process will go through its list of removed nodes and will free those nodes that are not protected by any hazard pointers. The programmer needs to ensure the following condition:

Condition 1. *At the time of any hazardous access by a process p to the memory location of a node n (access hazard), n has been continuously protected by one of p's hazard pointers since a time when n was definitely safe for p.*

Michael [25] provides a *methodology* for programmers to enforce this condition: (1) Identify all hazardous references and access hazards in the data structure code. (2) For each hazardous reference, determine the step when the reference is created and the last access hazard where the reference is used. This is the period when the reference needs to be protected by a hazard pointer. (3) Examine the overlap of the periods from step 2. The maximum number of hazard pointers is the maximum number of distinct hazardous references that exist simultaneously for the same process. (4) For each hazardous reference, assign a hazard pointer to it and immediately afterwards, verify if the reference (the node) is still safe. If the verification fails, follow the path of the original algorithm that corresponds to failure due to contention (e.g. try again, backoff etc.).

The Problem of Instruction Reordering in HP. An important practical consideration when applying the above methodology is instruction reordering [18, 21]. In most modern processors, instructions may be executed out of order for performance considerations. In particular, in the widespread x86/AMD 64/SPARC TSO memory models, stores may be executed after loads, even if the stores occur before loads in the program code [21, 28]. Instruction reordering is relevant in the case of hazard pointers due to step 4 in the methodology above. We assign a hazard pointer and then verify that the node is still safe, thus ensuring that the hazard pointer starts protecting that node from a time when the node is definitely safe. However, if assigning the hazard pointer (a store) is reordered after the validation (a load), then we can no longer be certain that the node is still safe when the hazard pointer becomes visible to other processes. Therefore, it is necessary for the programmer to insert a *memory barrier* between the hazard pointer assignment and the validation. Algorithm 1 shows the high-level instructions that need to be added to the data structure code when accessing a node, if hazard pointers are used (lines 2–4).

```
1  Read reference to node $n$
2  Assign a hazard pointer to $n$
3  Perform a memory barrier
4  Check if node $n$ is still valid
5  Access $n$'s memory
6  Release reference to $n$ (e.g. move to successor node)
```

Algorithm 1: High-level steps taken when accessing a node in a data structure using *hazard pointers*.

A *memory barrier* [18] (or fence) is an instruction that ensures no memory operation is reordered around it. In particular, all stores that occur before a memory barrier in program order will be visible to other processes before any loads appearing after the barrier is executed. Therefore, when validating that a node is still safe (as per step 4 in the methodology), we can be certain that the hazard pointer is already visible.

Memory barriers are expensive instructions. They can take hundreds of processor cycles. This cost results in a significant performance overhead for hazard pointer implementations, especially in read-only data structure operations (update operations typically use other expensive synchronization primitives such as compare-and-swap, so the marginal cost of memory barriers due to hazard pointers is much lower than for read-only operations). Moreover, these memory barriers must be performed on a per-element basis, which causes the performance of hazard pointers to scale poorly with data structure size.

4. AN OVERVIEW OF QSENSE

Combining QSBR's high-performance with hazard pointers' robustness in a hybrid memory reclamation scheme is appealing. In this section, we first argue why merging QSBR with the original hazard pointers technique is also however a challenge. Then, we give a high-level view of QSense.

4.1 Rationale

One could imagine a hybrid scheme where QSBR and hazard pointers are two separate entities, with the switch between the two schemes triggered by a *signal* or *flag*. QSBR would run in the common case (when no process delays are observed) and hazard pointers would be employed when a long process delay makes quiescence impossible. However, after a switch to hazard pointers based reclamation, hazardous references from when the system was running in QSBR mode would need to be protected as well. So, hazard pointers should be protecting nodes during the entire execution of the system, regardless of the mode of operation the system is currently in. As discussed above, the original hazard pointers algorithm requires a memory barrier call after every hazard pointer update, to ensure correctness. However, ideally, when the system operates in QSBR mode, the per-node memory barriers required by hazard pointers should be eliminated. Per-node memory barriers should be placed as specified in the HP algorithm only when the system goes into fallback mode. Nonetheless, such an approach is not correct. The scenario in Algorithm 2 illustrates why.

```
1  $R_1$. Read a pointer to a node $n$ (Load)
2  $R_2$. Assign a hazard pointer to $n$ (Store)
3  $R_3$. If fallback mode is active (Load) execute a memory
         barrier (here, suppose fallback mode is
         inactive and the memory barrier is not
         executed)
4  $R_4$. Recheck $n$ (Load)
5  $R_5$. Use $n$ (Loads and Stores)
6
7  $D_1$. Remove $n$
8  $D_2$. Check fallback−flag (here, suppose fallback
         mode was activated)
9  $D_3$. Scan hazard pointers
10 $D_4$. Free $n$ (assuming no hazard pointer protects it)
```

Algorithm 2: Example of illegal operation interleaving

Consider two processes, P_R and P_D, at a time t when the system makes the switch from the fast path to the fallback path. P_R is a reader process performing steps R_1 to R_5 and P_D is a deleting process performing steps D_1 to D_4, during which it detects that it must switch to the fallback scheme.

Assume that P_D's steps are not reordered (we can use memory barriers to ensure this, since we are mainly concerned with removing memory barriers from read-only operations, but not necessarily from deletion operations). However, P_R's steps can be reordered; more precisely, in the TSO model, the R_2 store can be delayed past all subsequent reads [21, 28] if P_R does not detect that the fallback-flag is turned on and does not perform a memory barrier.

Next, consider the following interleaving of P_R's and P_D's steps. Initially the fallback-flag is off (the system is running in the fast path). The reader will read a reference to n (R_1) and assign a hazard pointer to n (R_2). At R_3 the fallback-flag is off, so the memory barrier is not executed. Thus, the store of the hazard pointer (R_2) can be delayed past all subsequent reads. Then, P_R rechecks n (R_4) and finds that it is still a valid node. Now, suppose that another process P activates the fallback path. This is where P_D steps in and executes D_1 through D_4. Since P_R's store of the hazard pointer was delayed, P_D is free to reclaim the node in question. Finally, in step R_5, P_R will try to use the reference to n that it had acquired without publishing the hazard pointer via a memory barrier and thus attempt to access a reclaimed node, which is incorrect.

If a memory barrier was called after the update of each hazard pointer in both the fast and fallback paths, the QSBR/HP hybrid would function correctly. However, adding per-node memory barriers when running the fast path means re-introducing the main performance bottleneck of the fallback scheme into the fast path. Consequently, the performance of the hybrid in QSBR mode would be similar to its performance in HP mode, what we initially set out to avoid.

The challenge is to eliminate the traversal memory barriers when the system operates in the fast path (QSBR), while optimistically updating the hazard pointer values. We address this challenge by designing Cadence, a hazard pointer inspired memory reclamation scheme which does not require per-node memory barriers upon traversal. Cadence is presented in § 5. Then, in § 6 we show that Cadence is a good candidate for the fallback scheme in QSense, preserving the safety properties of the algorithm, while not hindering the performance of QSBR in the fast path.

4.2 QSense in a Nutshell

QSense is a hybrid scheme, unifying the high-performing approach provided by QSBR and the robustness provided by hazard pointers. QSense is an adaptive scheme, using two paths (a fast path and a fallback path) and automatically deciding when it should switch paths. QSBR constitutes the fast path and is used in the common case, when all processes are active in the system. As QSBR was presented in prior work [6, 14], we omit the implementation details.

When one or more processes experience prolonged delays (e.g. blocked in I/O), there is a risk of exhausting the available memory, because quiescence is not possible. In this case, Cadence serves as a safety net. Cadence guarantees that QSense continues to function within a bounded amount of memory. Cadence eliminates the expensive per-node memory fences needed for data structure traversal, the main drawback of the original hazard pointer scheme. Instead of using memory barriers, Cadence forces periodic context switches in order to make outstanding hazard pointer writes visible. Since the cost of expensive operations (in our case context switches) is spread across a large number of operations, Cadence achieves scalability that is up to three times as good as the original hazard pointer scheme, while maintaining the same safety guarantees. Moreover, using Cadence as the fallback path allows the elimination of memory barriers when running in the fast path. QSense automatically detects the need to switch to the fallback scheme and triggers the switch through a shared flag. Similarly, when all the processes become active again in the system (e.g. return from a routine incurring a long delay), QSense re-establishes the quiescence-based reclamation mechanism automatically.

Applicability. QSense can be used with any data structure for which both the fast path and the slow path are applicable. Since Cadence does not introduce any additional usability constraints compared to hazard pointers, and QSBR can be applied to virtually any data structure, this means that QSense can be used with any data structure for which hazard pointers are applicable. Applying QSense to a data structure is done in three steps: (1) Call `manage_qsense_state` to declare a quiescent state (e.g. between every two data structure operations). The function automatically handles amortizing overhead by executing the memory reclamation code only once every Q calls to `manage_qsense_state` (where Q is the quiescence threshold introduced in § 3.1). (2) Following the methodology from § 3.2, protect hazardous references by calling `assign_HP`. (3) To reclaim memory, call `free_node_later` when `free` would be called in a sequential setting. An example of how to apply QSense to a concurrent linked list can be found in the appendix.

5. CADENCE

In this section, we present Cadence, our fallback path for QSense. We first describe Cadence as a stand-alone memory reclamation technique and then show how we integrate Cadence with QSBR in our QSense scheme.

5.1 The Fallback Path

Cadence builds upon hazard pointers, eliminating the need to use per-node memory barriers when traversing the data structure. Note that while Cadence is the fallback path in QSense, it could also be used as a stand alone memory reclamation scheme. Algorithm 3 presents the pseudocode of the main functions of Cadence. This scheme is based on two new concepts: *rooster processes* and *deferred reclamation.*

1. **Rooster processes** are a mechanism used to ensure that all new hazard pointer writes become globally visible after at most a period of time T (the sleep interval, a configurable parameter). For every core or hardware context on which a worker process is running, we create a rooster process and pin it to run on that core. Every rooster process has the task of sleeping for a predetermined amount of time T, waking up, and going back to sleep again, in an infinite loop. In this way, every worker process is guaranteed to be switched out periodically, and thus the worker processes' outstanding writes, *including hazard pointers*, will become visible to the other processes (as detailed in *Note on assumptions* below). Therefore, for every time t, all hazard pointers that were published before time $t - T$ are visible to all processes.

2. **Deferred reclamation.** Figure 4 illustrates the main idea of how rooster processes and deferred reclamation work together. By verifying that a node n is *old enough* (pseudocode shown in Algorithm 3, lines 35–39), the reclaiming process makes sure that at least one rooster process wake-up has occurred since n was removed from the data structure and thus that any hazard pointers protecting n have become visible. If n is removed by a process at time t_0, then at time $t > t_0 + T$, any potential hazard pointers protecting n are visible. This is because these hazard pointers were written before t_0 (by the methodology in § 3.2) and at least one rooster process wake-up has

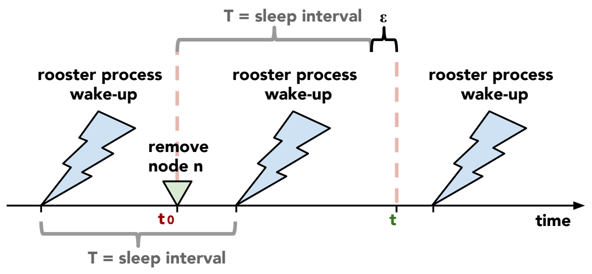

Figure 4: Rooster processes and deferred reclamation

occurred between t_0 and t. Therefore, at t, the reclaiming process can safely free the node's memory provided that no hazard pointers are protecting it.

```
 1  // *** Cadence ***
 2  timestamped_node {
 3   node* actual_node;
 4   timestamp time_created;
 5   timestamped_node* next;};
 6  HP_array HP; // Shared HP array
 7
 8  void assign_HP(node* target, int HP_index) {
 9    // Assign hazard pointer to target node.
10    HP[HP_index] = target;
11    // No need for a memory barrier here.
12  }
13
14  void scan(timestamped_node* removed_nodes_list)
15  {
16    // Insert non-null values in HP_in_use.
17    HP_array HP_in_use = get_protected_nodes(HP);
18    // Free non-hazardous nodes
19    timestamped_node* tmplist;
20    tmplist = removed_nodes_list;
21    removed_nodes_list = NULL;
22    timestamped_node* cur_node;
23    while (tmplist != NULL) {
24     cur_node = tmplist;
25     tmplist = tmplist->next;
26     // Deferred reclamation
27     if (!is_old_enough(cur_node) ||
28     HP_in_use.find(cur_node->actual_node)) {
29      cur_node->next = removed_nodes_list;
30      removed_nodes_list = cur_node;
31     } else {
32      free(cur_node->actual_node);
33      free(cur_node);
34  } } }
35
36  boolean is_old_enough(timestamped_node*
        wrapper_node) {
37    current_time = get_current_time();
38    age = current_time - wrapper_node->
        time_created;
39    return (age >= (ROOSTER_SLEEP_INTERVAL +
        EPSILON));
40  }
```

Algorithm 3: Main functions of Cadence

When a node n is removed from the data structure (when the `free` function would be called in a sequential setting), n is *timestamped* and placed inside the removing process' local list of nodes awaiting reclamation. Once every R such node removals, a `scan` of the list is performed (see Algorithm 3, lines 14–33). A `scan` inspects the process' removed nodes list and frees those nodes that are safe to reclaim (retired). Nodes that are *old enough and are not protected by any hazard pointers* are freed; hazard pointers of all the worker processes are checked, not only the reclaiming process'. The rest of the nodes — which are either not old enough, or are protected — are left in the removed nodes list, to be reclaimed at a later time (Algorithm 3, lines 26–30).

From the programmer's perspective, using Cadence is essentially identical to using hazard pointers. The difference is that no memory barriers are needed after publishing a new hazard pointer, thus making Cadence up to three times as fast than the original hazard pointers (as we will see in § 7). Algorithm 4 shows the steps taken when accessing a node, when Cadence is used.

```
Read reference to node $n$
Assign a hazard pointer to $n$
~~Perform a memory barrier~~
Check if $n$ is still valid
Access $n$'s memory
Release reference to $n$ (e.g. move to successor node)
```

Algorithm 4: High-level steps taken when accessing a node in a data structure using *Cadence*.

Note on assumptions. For correctness, Cadence relies on the following assumptions. First, we assume that the only instruction reorderings possible are the ones between loads and subsequent reads, as in the TSO model. We assume this in § 3.2 to determine the memory barrier placement and in § 5.1 to justify that the memory barrier is no longer needed.

Second, we require that a context switch implies a memory barrier for the process being switched out. This is necessary to guarantee that hazard pointers published by a process p become visible at the latest right after p is switched out by a rooster process. The low-level locking required to perform a context switch automatically provides a memory barrier for the process being switched out. Although this property is architecture dependent and might not be generally true if architectures change in the future, it does hold for most modern architectures [19, 21].

Third, we assume that rooster processes never fail. This is a reasonable assumption, considering that rooster processes do not take any steps that could produce exceptions: their only actions are going to sleep and waking up. However, small timing inconsistencies might appear, in the form of (1) rooster processes possibly taking slightly longer than T between wake-ups, i.e. "oversleeping" (it is reasonable to assume that this difference is small, since modern operating systems use fair schedulers [20]) and (2) different cores seeing slightly different times when creating and comparing timestamps. We make the assumption that these inconsistencies are bounded and introduce a tolerance ϵ to account for them. We use ϵ explicitly, as shown in Figure 4: to verify if a removed node n is old enough, we compute the difference between the current value of the system clock and n's timestamp. If the time difference is larger than $T + \epsilon$, then n is old enough.

Note that we are making no assumptions about the *worker processes* (i.e. the processes performing read or write operations on the data structure). In particular, they may be delayed for an arbitrary amount of time. Therefore, the model under which our construction is correct and wait-free is partly asynchronous (the worker processes) and partly synchronous (the rooster processes).

5.2 Merging the Fast and Fallback Paths

In QSense, from a high-level design point of view, the fast path (QSBR) and the fallback path (Cadence) can be viewed as two separate entities, functioning independently. The switch between the two modes is triggered via a shared flag, called the *fallback-flag*. However, even if the two modes of operation are logically distinct, there are elements of the two schemes that have been merged or that are continuously active. Even if QSense is running in the fast path, hazard pointers still have to be set. As explained in § 4.1, this is necessary because in the case of a switch to the fallback path, hazardous references need to be protected. Furthermore, for similar reasons, timestamps need to be recorded when a node is removed, regardless of the mode of operation of QSense. Moreover, when running in fallback mode and performing a `scan`, QSBR's `limbo_list` (with all three epochs) becomes the `removed_nodes_list` scanned by Cadence. Pseudocode for the main functions used by QSense is shown in Algorithm 5 (unless stated otherwise, all pseudocode references in this section refer to Algorithm 5).

Any of the worker processes can trigger the switch between the fast and fallback paths. The switch can be split into the following sub-problems:

1. **Detecting the need to switch to the fallback path.** QSense triggers the switch from QSBR to Cadence when a process detects that its removed (but not freed) nodes list reached a size C, where C is a parameter of the system (lines 53–60). Reaching a large removed nodes list size for one process indicates that quiescence was not possible for an extended period.
2. **Switching from the fast path to the fallback path.** QSense signals the switch from QSBR to Cadence through the *fallback-flag*. The process that has detected the need for the switch sets the shared *fallback-flag*. The *fallback-flag* is checked by all processes when performing node reclamation (i.e. calling the `free_node_later` function, shown in lines 35–60), so the path switch will eventually be detected by all active processes (line 41). If the flag is set to *fallback mode*, a hazard pointer style `scan` as described in § 5.1 is immediately performed to reclaim nodes (lines 42–47; `scan` shown in Algorithm 3).
3. **Detecting when it is safe to switch back to the fast path.** To determine when to switch from the fallback path to the fast path we need to verify if all processes have once again become active in the system. While the system operates in fallback mode, there is at least one process which cannot participate, because using the fallback path implies that one of the processes was delayed and quiescence was not possible. To assess whether all the processes have become active in the meantime, we keep an array of *presence-flags* (one flag per worker process), which is reset periodically. After each operation (or batch of operations) on the data structure, processes set their corresponding *presence-flags* to true, to signal that they are active (line 18). Then, processes scan the *presence-flag* array (line 26). If one of the processes sees all of the presence flags set to true, it infers that all processes might be active again in the system and a switch from fallback path to fast path is attempted.
4. **Switching from the fallback path to the fast path.**

```
1  //*** QSENSE interface ***
2  void manage_qsense_state();
3  void assign_HP(node* target, int HP_index);
4  void free_node_later(node* n);
5
6  // One limbo list per epoch
7  timestamped_node* limbo_list[3];
8  int call_count = 0;
9  int free_node_later_call_count = 0;
10
11 // *** QSENSE main functions***
12 void manage_qsense_state(){
13  // Batch operations
14  call_count += 1;
15  if (call_count % QUIESCENCE_THRESHOLD != 0) {
16   return;
17  }
18  // Signal that the process is active
19  is_active(process_id);
20  seen_fallback_flag = fallback_flag;
21  if (seen_fallback_flag == FAST_PATH) {
22   // Common case: run the fast path
23   quiescent_state();
24   prev_seen_fallback_flag = FAST_PATH;
25  } else if (seen_fallback_flag == FALLBACK_PATH
       ) {
26   // Try to switch to fast path
27   if ( all_processes_active() ) {
28    // Trigger switch to the fast path
29    fallback_flag = FAST_PATH;
30    prev_seen_fallback_flag = FAST_PATH;
31    quiescent_state();
32   }
33   prev_seen_fallback_flag = FALLBACK_PATH;
34 } }
35
36 void free_node_later (node* n) {
37  // Create timestamped wrapper node
38  timestamped_node* wrapper_node = alloc(size(
       timestamped_node));
39  wrapper_node->actual_node = n;
40  wrapper_node->time_created = get_current_time
       ();
41  limbo_list[my_current_epoch].add(wrapper_node)
       ;
42  seen_fallback_flag = fallback_flag;
43  if (seen_fallback_flag == FALLBACK_PATH &&
44  ++free_node_later_call_count % R == 0) {
45   // Running in fallback mode. All three epochs
        in limbo list are scanned.
46   scan(limbo_list[0]); scan(limbo_list[1]);
47   scan(limbo_list[2]);
48   prev_seen_fallback_flag = FALLBACK_PATH;
49  } else if ( prev_seen_fallback_flag ==
       FALLBACK_PATH &&
50    seen_fallback_flag == FAST_PATH) {
51   // QSBR mode switch triggered by another
        process.
52   quiescent_state();
53   prev_seen_fallback_flag = FAST_PATH;
54  } else if (size(limbo_list) >= C &&
55    prev_seen_fallback_flag == FAST_PATH) {
56   // Trigger switch to fallback mode:
57   fallback_flag = FALLBACK_PATH;
58   prev_seen_fallback_flag = FALLBACK_PATH;
59   scan(limbo_list[0]); scan(limbo_list[1]);
60   scan(limbo_list[2]);
61 } }
```

Algorithm 5: Main QSense functions

If QSense runs in fallback mode, but all processes have become present in the meantime, the possibility to switch from Cadence back to QSBR is detected, as described above. Similarly to switching from the fast path to the fallback path, the switch in the opposite direction is signaled through setting the value of the shared *fallback-flag* and immediately declaring a quiescent state (lines 27–30). If the switch to the fast path was already triggered by another process, the new value of the *fallback-flag* will be seen upon retiring a node (in the `free_node_later` function, lines 48–52).

The current version of QSense does not support dynamic membership: processes cannot join or leave the system as an algorithm is running. Also, if a process crashes and never recovers, QSense will switch to fallback mode and stay there forever. Both of these issues can be addressed by adding mechanisms for processes to announce entering or leaving the system and for evicting participating processes that have not quiesced in a long time. We leave these extensions for future work.

6. CORRECTNESS & COMPLEXITY

In this section we argue for the safety and liveness of Cadence and QSense. For completeness, safety and liveness proofs for QSBR can be found in the appendix.

6.1 Cadence

PROPERTY 1. (SAFETY) *If at time t, a node n is identified in the `scan` function as eligible to be reused by process p, then no process $q \neq p$ holds a hazardous reference to n at time t.*

PROOF. Assume by contradiction that (1) n is identified as eligible for reuse by p at time t and (2) there exists another process q that holds a hazardous reference to n at t. Then by (1) and the `scan` algorithm, at time t, p inspects n's timestamp and finds that n is old enough, meaning that n has been removed from the data structure at a time $t' \leq t - T$ (where T is the rooster process sleep interval, including the tolerance ϵ). Therefore, by Condition 1 in § 3.2, q has had a hazard pointer hp dedicated to n since a time $t'' \leq t'$. Since $t - t'' \geq T$, the write by q to hp at t'' is visible to p at time t. Therefore, by the `scan` algorithm, p will not identify n as eligible for reuse (since it is protected by a hazard pointer). □

LEMMA 1. *For any process p, at the end of a call to `scan` by p, p can have at most $NK+T$ retired nodes in its removed nodes list, where N is the number of processes, K is the number of hazard pointers per process and T is the rooster process sleep interval.*

PROOF. At the time of the `scan`, there can be at most NK nodes protected by hazard pointers (since NK is the total number of hazard pointers), and there can be at most T nodes that are not yet old enough (for clarity, we assume that p can remove at most one node per time unit). □

PROPERTY 2. (LIVENESS) *At any time, there are at most $N(K + T + R)$ retired nodes in the system, where R is the number of nodes a process can remove before it invokes `scan`.*

PROOF. Fix a process p and examine how many nodes can be removed by p but not yet reclaimed. Using Lemma 1, it follows that between two `scan` calls, the number of nodes in p's removed nodes list can grow up to $NK+T+R$, before the next `scan` call is triggered, lowering the size of the removed nodes list to $NK + T$ again. So the maximum size of a process' removed nodes list is $NK + T + R$, where the NK term comes from the nodes that are protected by hazard pointers. When considering the entire system, we can have at most NK HP-protected nodes, and at most $N(T + R)$ non-HP-protected nodes, so the maximum number of retired nodes is $N(K + T + R)$. □

6.2 QSense

PROPERTY 3. (SAFETY) *If a node n is identified at time t by process p as eligible for reuse, then no process $q \neq p$ holds a hazardous reference to n.*

PROOF. Since all processes keep track of both hazard pointers (exactly as in Cadence) and of epochs (exactly as in QSBR), regardless of whether the system is in the fast path or in the fallback path, the safety guarantees of both methods are maintained. □

For the next property, we define a *legal* value of C, the threshold introduced in step 1 of § 5.2. C refers to the size of the removed nodes list and is used for determining when to switch to the fallback path. We say that C is legal if $C > \max(mQ, NK + T, (K + T + R)/2)$, where Q is the quiescence threshold introduced in § 3, m is the maximum number of nodes that can be removed by a single operation, and N, K and T are as in § 6.1. Picking a legal value for C is always possible since C is a configurable parameter.

PROPERTY 4. (LIVENESS) *If C has a legal value then, at any time, there can be no more than $2NC$ retired nodes in the system.*

PROOF. Assume by contradiction that there is a time t when there are $U > 2NC$ retired nodes in the system. Then there exists a process p such that at time t, p has more than $2C$ retired nodes.

Let $t_1 < t$ be the time of the last quiescent state called by p before t. Since t_1, p has completed at most Q operations (otherwise p would have gone through another quiescent state before t) and has therefore removed at most mQ nodes. Therefore, at t_1, p has at least $2C - mQ > C$ (using the fact that $C > mQ$) retired nodes and therefore triggers a switch to the fallback path. This means that p will call a hazard pointers `scan` before starting any other operation, by construction of the QSense algorithm (step 2 in § 5.2). Let t_2 be the time of this first `scan` after t_1. After this scan is complete, p can have at most $NK + T$ retired nodes, by Lemma 1. If $t > t_2$, then at t, p can have at most $NK + T + mQ < 2C$ retired nodes ($NK + T$ at most at the end of the `scan` plus mQ because p has completed at most Q operations since t_2), a contradiction. Therefore it must be the case that $t_1 < t < t_2$. Since the `scan` at t_2 is called immediately after the quiescent state at t_1, without other operations being started by p (by step 2 in § 5.2), the number of retired nodes does not increase between t_1 and t. So at t_1, p has more than $2C$ retired nodes. We now show this to be impossible.

Let t_3 be the time of the last quiescent state called by p before t_1. Since p performed Q operations between t_3 and t_1, it follows that at t_3, p had at least $2C - mQ > C$ retired nodes, thus triggering a switch to the fallback state and a `scan` at time t_4, $t_3 < t_4 < t_1$. After this `scan`, p had at most $NK + T$ retired nodes and therefore, at t_1, p had at most $NK + T + mQ < 2C$ retired nodes, a contradiction because we had shown that p had more than $2C$ retired nodes at t_1. This completes the proof. $\square$

7. EXPERIMENTAL EVALUATION

We first describe the experimental setup of our evaluation. We proceed with presenting the methodology of our experiments and we finally discuss our evaluation results.

7.1 Experimental Setting

We apply QSense to a lock-free linked list [24], a lock-free skip list [11] and a binary search tree [27]. The base implementations of these data structures are taken from ASCYLIB [8]. The code to reproduce our experiments is available at `https://github.com/zablotchi/qsense`. We compare the performance of QSense to that of QSBR, HP and a leaky implementation. Our evaluation was performed on a 48 core AMD Opteron with four 12-core 2.1 GHz Processors and 128 GB of RAM, running Ubuntu 14.04. Our code was compiled with GCC 4.8.4 and the `-O3` optimization level.

Each experiment consists of a number of processes concurrently performing data structure operations (searching, inserting or deleting keys) for a predefined amount of time. Each operation is chosen at random, according to a given probability distribution, with a randomly chosen key. An initialization is performed before each experiment, where one process fills the data structure up to half the key-range.

7.2 Methodology

The purpose of our evaluation is two-fold. First, we aim to determine whether QSense performs similarly to QSBR in the base case. It is important to highlight the base case behavior of QSense (i.e. when no processes undergo prolonged delays), since this is the expected execution path in most scenarios. To this end, we run a set of tests emphasizing the scalability with respect to the number of cores. For this first category of tests, the system throughput is recorded as a function of the number of cores (each process is pinned to a different core). The number of cores is varied from 1 to 32, the operation distribution is fixed at 50% reads and 50% updates (i.e. 25% inserts, 25% deletes) and the key range sizes are fixed at 2000 for the linked list, 20000 for the skip list and 2000000 for the binary search tree.

The second goal of the evaluation is to examine the behavior of QSense in case of prolonged process delays. To this end, we run a set of tests that include periodic process disruptions and measure the system throughput as a function of *time*. We observe the switch from QSBR to Cadence when the system senses that one of the processes was delayed and the switch back to QSBR when all the processes became active in the system again. We seek to trigger a switch between the paths (QSBR to Cadence or oppositely) every 10 seconds. To induce the system switch, every 20 seconds one of the processes is delayed for a period of 10 seconds. The operation distribution and key range sizes are fixed as above and the number of processes is fixed at 8. As before, the data structure is filled up to half of its capacity, processes start simultaneously, then run for 100 seconds. The throughput is recorded as a function of time and is measured every second.

7.3 Results

The top row of Figure 5 shows the behavior of QSBR, QSense, HP and None (the leaky implementation) on the linked list, on the skip list and on the binary search tree in the common case, when no process delays occur. The throughput of the algorithms is plotted as a function of number of cores (higher is better). Due to its amortized overhead, QSBR maintains a 2.3% overhead on average compared to None. As expected, HP achieves the lowest overall throughput, with an average overhead of 80% over the leaky implementation. This is due to the expensive per-node memory barriers used upon data structure traversal. QSense achieves two to three times better throughput than HP and has an average overhead of 29% over None. While close to QSBR in all scenarios, QSense does not match its performance. Even if QSense follows a quiescence-based reclamation scheme in the base case, it still has to maintain the hazard pointer and timestamp values updated. This induces increased overhead, compared to QSBR. Despite the fact that no memory barriers are used upon updating the hazard pointers, the process-local variable updates alone add sequential complexity. This explains why the performance gap between QSBR and QSense is larger for the skip list than for the linked list, or the tree: whereas the linked list only uses two hazard pointers per process and the tree uses six, the skip list can use up to 35 hazard pointers per process.

QSense does not completely match the performance of QSBR but, unlike QSBR, it can recover from long process delays. The bottom row of Figure 5 illustrates this. The throughput of QSense is shown alongside QSBR and HP. In this experiment, one of the processes is delayed in the 10-20, 30-40, 50-60, 70-80 and 90-100 time intervals. A similar pattern can be observed for all three data structures. Note that the first time a process is delayed (after 10 seconds), QSBR (shown in orange) can no longer quiesce. Consequently, the system runs out of memory and eventually fails. In contrast, QSense (shown in green) continues to run. When quiescence is no longer possible, QSense switches to Cadence and then switches back to QSBR as soon as the delayed process becomes active. The sequence of fallbacks and recoveries is continued throughout the entire experiment run. As intended, the throughput achieved by QSense is similar to QSBR during the fast path. When QSense is running in the fallback path, it can be seen that Cadence outperforms the original hazard pointer scheme by a factor of 3x on average, because it avoids per-node memory barriers upon data structure traversal.

8. RELATED WORK

Reference counting (RC) [9, 12, 15, 30] assigns shared counters to each node in the data structure, representing the number of references to the nodes held at every given time. When a node's counter reaches zero, the node is safe to reclaim. Though easy to implement, RC requires expensive atomic operations on every access to maintain consistent counters.

Pointer-based techniques. Hazard pointers (HP) [25], or Pass-the-Buck [16] rely on the programmer marking nodes that are unsafe to reclaim before performing memory accesses to their locations. A reclaiming process must ensure that a node it is about to deallocate is not marked. An advantage of these schemes is that they maintain the lock-

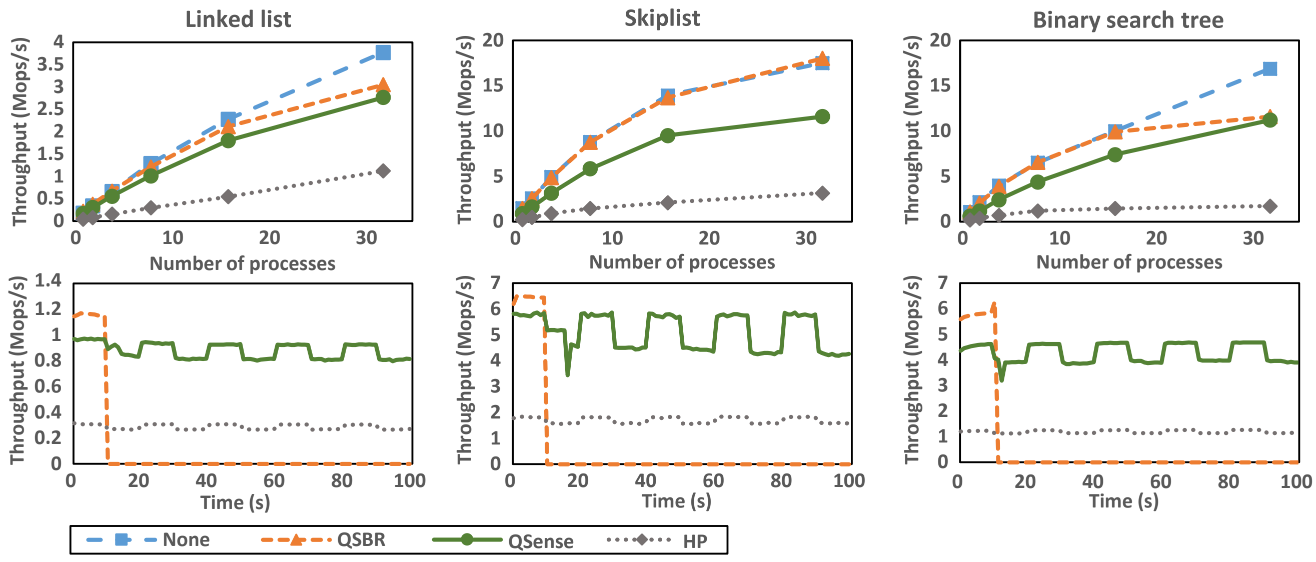

Figure 5: Scalability of memory reclamation on a linked list (2000 elements), a skip list (20000 elements) and a BST (2000000 elements) with 50% updates (top row); Path switching with process delays (8 processes, 50% updates) (bottom row).

free property of data structures. Yet, their implementation requires a memory fence instruction to be issued for every node traversed, which significantly hinders the performance of read-only operations [14].

Improvements on HP. Morrison et al. [26] introduce a new, strengthened version of the Total Store Ordering (TSO) memory model [28] in which there is a known bound on the time it takes for writes in the store buffer to become visible in main memory. A variant of HP which does not need memory barriers is proposed. While this solution is similar to Cadence, it relies on hardware guarantees that do not exist yet in practice. McKenney et al. [22] describe a method to force quiescent states by creating a high-priority daemon process that executes on each CPU in turn, but this method has never been applied to HP. Finally, Aghazadeh et al. [1] provide an improvement on HP by reducing the number of hazard-pointer-to-node comparisons per scan call to one, at the cost of increasing the amount of time between node removal and node reclamation.

Epoch-based techniques. These techniques [13, 14, 23] rely on the assumption that live processes will eventually drop references they hold on a node (i.e. if a reclaiming process waits long enough, all other processes will eventually stop holding references to the deleted nodes, which will thus become safe to reclaim). Though epoch-based techniques have good performance [14], their main drawback is that they are blocking.

Ad-hoc techniques. Drop the Anchor (DTA) [5] combines timestamping with a HP-inspired technique. Processes use timestamps to track their progress and they place anchor pointers in the data structure upon traversal. When a process delay occurs, the other processes work together to reconstruct the data structure using the anchors. Similarly to QSense, DTA spreads the cost of expensive operations across a large number of operations. However, the applicability of DTA to other data structures, besides the linked list implementation provided by the authors, is unclear. In contrast, QSense is as easy to apply as HP, for which a well established methodology exists.

DEBRA+ [6] uses a variant of QSBR when processes are making progress in the common case and has a slower mechanism for treating delays. When a delayed process is preventing other processes from quiescing for too long, the slow process is *neutralized*, using an OS signal. Upon resuming execution, a neutralized process runs special *recovery code* to clean up any inconsistencies it might have left in the data structure before it was neutralized. Like QSense, DEBRA+ has a fast path and a recovery path. Unlike QSense, DEBRA+ relies on OS-specific instructions. Moreover, DEBRA+ is only easy to apply to lock-free data structures that have (1) an explicit help function and an explicit descriptor record containing all the information required by the help function and (2) operation code that follows a certain pattern (quiescent preamble – non-quiescent body – quiescent postamble). It is unclear how to extend DEBRA+ to state-of-the-art algorithms not satisfying the above properties.

Automatic memory reclamation. ThreadScan [3] is an automatic technique for concurrent memory reclamation. Processes add references to removed nodes to a shared *delete buffer*. Periodically, a scan is initiated by sending a signal to all processes. Each process examines its stack and registers and marks the corresponding entry in the delete buffer if there is a match, to indicate that the node is in use. After all processes complete the scan, unmarked nodes can be freed. ThreadScan amortizes the overhead of memory reclamation, similarly to QSense. Unlike QSense, ThreadScan has the advantage of being completely automatic. However, ThreadScan makes critical assumptions about the synchrony of its *worker processes* and about the layout of process stacks in memory. Cohen and Petrank [7] present an automatic memory reclamation scheme for lock-free data structures, inspired by mark-and-sweep garbage collection. This technique relies on the data structure operations being in normalized form [29]. While such implementations of lock-free data structures exist, there is no clear methodology on how to normalize lock-free data structure implementations.

HTM techniques. Another direction for concurrent memory reclamation is the use of Hardware Transactional Memory (HTM) [17]. Dragojevic et al. [10] use HTM to produce faster and simpler solutions to a common subproblem in prior memory reclamation techniques. StackTrack [2] uses the bookkeeping facilities of HTM directly to track node references. The downside of this class of solutions is that they rely on the presence of HTM on the target machine, which is rare in practice.

9. CONCLUSION

QSense offers a fast, robust and highly applicable solution to the concurrent memory reclamation problem. Whenever possible, the fast QSBR technique is used. In case of prolonged process delays, QSense switches to Cadence, a novel hazard pointer inspired scheme, that is robust to process delays. QSense can be integrated in data structures making use of the popular hazard pointer based schemes almost effortlessly, a significant advantage from the applicability standpoint. We show experimentally that QSense achieves performance similar to the highest-performing techniques in the common case (i.e. when all worker processes are active in the system), while tolerating process delays.

10. REFERENCES

[1] Z. Aghazadeh, W. Golab, and P. Woelfel. Making objects writable. PODC 2014.

[2] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. StackTrack: An automated transactional approach to concurrent memory reclamation. EuroSys 2014.

[3] D. Alistarh, W. M. Leiserson, A. Matveev, and N. Shavit. Threadscan: Automatic and scalable memory reclamation. SPAA 2015.

[4] A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma. Using read-copy-update techniques for System V IPC in the Linux 2.5 kernel. USENIX Annual Technical Conference 2003.

[5] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. SPAA 2013.

[6] T. A. Brown. Reclaiming memory for lock-free data structures: There has to be a better way. PODC 2015.

[7] N. Cohen and E. Petrank. Automatic memory reclamation for lock-free data structures. OOPSLA 2015.

[8] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. ASPLOS 2015.

[9] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. PODC 2001.

[10] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. PODC 2011.

[11] K. Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge Computer Laboratory, 2004.

[12] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans. Parallel Distrib. Syst.*, 20(8), 2008.

[13] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. DISC 2001.

[14] 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), 2007.

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

[16] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. DISC 2002.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. ISCA 1993.

[18] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* 1st edition, 2012.

[19] E. Ladan-Mozes, I.-T. A. Lee, and D. Vyukov. Location-based memory fences. SPAA 2011.

[20] R. Love. *Linux System Programming: Talking Directly to the Kernel and C Library, 2nd Edition.* O'Reilly Media, Inc.

[21] P. E. McKenney. Memory barriers: a hardware view for software hackers. 2010.

[22] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read copy update. Ottawa Linux Symposium 2001.

[23] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. PDCS 1998.

[24] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. SPAA 2002.

[25] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6), 2004.

[26] A. Morrison and Y. Afek. Temporally bounding TSO for fence-free asymmetric synchronization. ASPLOS 2015.

[27] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. PPoPP 2014.

[28] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: X86-TSO. TPHOLs 2009.

[29] S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. PPoPP 2014.

[30] J. D. Valois. Lock-free linked lists using compare-and-swap. PODC 1995.

APPENDIX

A. QSBR CORRECTNESS AND COMPLEXITY

Lemma 2. *If the time interval $[a, b]$ is a grace period, after time b no process holds hazardous references to nodes that were removed before time a.*

Proof. Let $[a, b]$ be a grace period and consider a node n that was removed at time $t < a$. By the definition of the removed state, this means that no process can obtain a reference to n after a, but it is possible for processes to still hold references to n that they obtained before t. By the definition of a grace period, all processes will pass through a quiescent state between a and b. Therefore, for each process p there exists a time t_j, $a \leq t_j \leq b$, when p does not hold any reference to n. Thus, at b, none of the processes hold any references (and in particular, hazardous references) to n. □

Lemma 3. *For any process p, at the time when p updates its local epoch from e_j to e_G, no process holds any hazardous references to the nodes already present in p's ${e_G}^{th}$ limbo list.*

Proof. Note that all epoch updates are done modulo three (because there are three logical epochs). Without loss of generality, suppose process p passes through a local epoch cycle $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$. We want to show that when p reaches epoch 0 for the second time, no processes hold any hazardous references to the nodes in p's limbo bag 0.

We claim that the system goes through a grace period $[a, b]$ starting just before the quiescent state during which p updates its local epoch from 0 to 1 and ending just after the transition by p of its local epoch from 2 to 0. Note that this claim implies, using Lemma 2, that after p's local epoch becomes 0 again, no processes hold hazardous references to the nodes in p's limbo bag 0, as required to complete the proof of Lemma 3.

We now proceed to prove the claim. Assume that $[a, b]$ is not a grace period. It follows that there exists a process q that does not go through a quiescent state during $[a, b]$. Therefore, we know that the local epoch of q stays the same during the time interval $[a, b]$. Since during $[a, b]$, p updates its local epoch from 0 to 1, then from 1 to 2 and then from 2 to 0, there exist times $t_1 < t_2 < t_3$, $t_1, t_2, t_3 \in [a, b]$ such that $e_G = 1$ at t_1, $e_G = 2$ at t_2 and $e_G = 0$ at t_3. Since some process transitions the global epoch from 1 to 2 between t_1 and t_2, it must be the case the the epoch of q is equal to 1 (otherwise the update cannot be completed). But this means that later during $[a, b]$ the global epoch cannot be advanced from 2 to 0, because there exists at least one process (q) whose local epoch is not equal to 2. We have reached a contradiction. This completes the proof of the claim and of Lemma 3. □

Property 5. (Safety) *If at time t, node n is identified by process p as eligible for reuse, then no process holds any hazardous references to n at time t.*

Proof. This follows from Lemma 3 and from the fact that a process p will identify a node n as eligible for reuse if and only if p has just updated its local epoch from e_j to e_G and n is in p's ${e_G}^{th}$ limbo list. □

B. QSENSE ON A LINKED LIST

Algorithm 6 and Algorithm 7 show an example of how QSense can be applied to a lock-free concurrent linked-list [13]. The lines of code needed to use QSense are highlighted (in blue). First, in the beginning of each list operation, the `manage_qsense_state` function is called. This function takes care of switching between QSBR and Cadence, if necessary, and invoking a quiescent state for every batch of performed operations. Second, hazard pointers are assigned to protect nodes when the list is traversed, in the same way one would use the original hazard pointer technique. The only difference is that the memory barrier between the hazard pointer assignment and the verification step is no longer needed. Finally, `free_node_later` should be called instead of `free`, when a node is removed.

```
Node* search(Node* set_head, Key key) {
  manage_qsense_state();
  Node *left_node, *right_node;
retry_search:
  left_node = set_head;
  right_node = set_head->next;
  while (True) {
    //Protect node by hazard pointer and
        perform verification, without the
        memory barrier
    assign_HP(left_node, 0); assign_HP(
        right_node, 1);
    if (right_node != left_node->next) {
      goto retry_search;
    }
    if (right_node->key >= key) {
      break;
    }
    left_node = right_node;
    right_node = unmarked(right_node->next);
  }
  return right_node;
}

Boolean insert(Node *list_head, Key key) {
  manage_qsense_state();
  do {
    Node* left_node;
    Node* right_node = search_and_cleanup(
        list_head, key, &left_node);
    if (right_node->key == key) {
      return False;
    }
    //Allocate a node with the allocator of
        your choice
    Node* node_to_add = new_node(key,
        right_node);
    if (CAS(&left_node->next, right_node,
        node_to_add) == right_node) {
      return True;
    }
    //Node was not inserted; free the node
        directly.
    free(node_to_add);
  } while (True);
}
```

Algorithm 6: QSense on a concurrent linked-list (I)

```
Boolean delete(Node *list_head, Key key) {
  manage_qsense_state();

  Node* cas_result;
  Node* unmarked_node;
  Node* left_node;
  Node* right_node;
  do {
    right_node = search_and_cleanup(list_head,
        key, &left_node);
    if (right_node->key != key) {
      return False;
    }
    //Try to mark right_node as logically
        deleted
    unmarked_node = unmarked(right_node->next);
    Node* marked_node = marked(unmarked_node);
    cas_result = CAS(&right_node->next,
        unmarked_node, marked_node);
  } while (cas_result != unmarked_node);

  if (!unlink_right(left_node, right_node)) {
    search_and_cleanup(list_head, key, &
        left_node);
  }
  return True;
}

Boolean unlink_right(Node* left_node, Node*
    right_node) {
  Node* new_next = unmarked(right_node->next);
  Node* old_right_node = CAS(&left_node->next,
      right_node, new_next);
  Boolean removed = (old_right_node ==
      right_node);
  if (removed){
    //call instead of free
    free_node_later(old_right_node);
  }
  return removed;
}

Node* search_and_cleanup(Node* set_head, Key
    key, Node** left_node_ref) {
  Node *left_node, *right_node;
retry_search_cleanup:
  left_node = set_head;
  right_node = set_head->next;
  while (True) {
    //Protect node by hazard pointer and
        perform verification, without the
        memory barrier
    assign_HP(left_node, 0); assign_HP(
        right_node, 1);
    if (right_node != left_node->next) {
      goto retry_search_cleanup;
    }
    if (!is_marked(right_node->next)) {
      if (right_node->key >= key) {
        break;
      }
      left_node = right_node;
    } else {
    //Perform cleanup of marked node
      unlink_right(left_node, right_node);
    }
    right_node = unmarked(right_node->next);
  }
  *left_node_ref = left_node;
  return right_node;
}
```

Algorithm 7: QSense on a concurrent linked-list (II)