Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way

Trevor Brown
Department of Computer Science
University of Toronto
tabrown@cs.toronto.edu

ABSTRACT

Memory reclamation for sequential or lock-based data structures is typically easy. However, memory reclamation for lock-free data structures is a significant challenge. Automatic techniques such as garbage collection are inefficient or use locks, and non-automatic techniques either have high overhead, or do not work for many reasonably simple data structures. For example, subtle problems can arise when hazard pointers, one of the most common non-automatic techniques, are applied to many natural lock-free data structures. Epoch based reclamation (EBR), which is by far the most efficient non-automatic technique, allows the number of unreclaimed objects to grow without bound, because one slow or crashed process can prevent all other processes from reclaiming memory.

We develop a more efficient, distributed variant of EBR that solves this problem. It is based on signaling, which is provided by many operating systems, such as Linux and UNIX. Our new scheme takes O(1) amortized steps per high-level operation on the lock-free data structure and O(1) steps in the worst case each time an object is removed from the data structure. At any point, $O(mn^2)$ objects are waiting to be freed, where n is the number of processes and m is a small constant for most data structures. Experiments show that our scheme has very low overhead: on average 10%, and at worst 28%, for a balanced binary search tree over many thread counts, operation mixes and contention levels. Our scheme also outperforms a highly efficient implementation of hazard pointers by an average of 75%.

Typically, memory reclamation code is tightly woven into lock-free data structure code. To improve modularity and facilitate the comparison of different memory reclamation schemes, we also introduce a highly flexible abstraction. It allows a programmer to easily interchange schemes for reclamation, object pooling, allocation and deallocation with virtually no overhead, by changing a single line of code.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.4.2 [**Operating Systems**]: Storage Management—*Allocation/deallocation strategies*

PODC'15, July 21–23, 2015, Donostia-San Sebastián, Spain.

DOI: http://dx.doi.org/10.1145/2767386.2767436.

General Terms

Algorithms, Performance, Reliability

Keywords

Memory reclamation; lock-free; non-blocking; epoch-based reclamation; hazard pointers; signals; fault-tolerance

1. INTRODUCTION

In concurrent data structures that use locks, it is typically straightforward to free memory to the operating system after a node is removed from the data structure. For example, consider a singly-linked list implementation of the set abstract data type using hand-over-hand locking. Hand-over-hand locking allows a process to lock a node (other than the head node) only if it holds a lock on the node's predecessor. To traverse from a locked node u to its successor v, the process first locks v, then it unlocks u. To delete a node u, a process first locks the head node, then performs hand-over-hand locking until it reaches u's predecessor and locks it. The process then locks u (to ensure no other process holds a lock on u), removes it from the list, frees it to the operating system, and finally unlocks u's predecessor. It is easy to argue that no other process has a pointer to u when u is freed.

In contrast, memory reclamation is one of the most challenging aspects of lock-free data structure design. Lock-free algorithms (also called non-blocking algorithms) guarantee that as long as some process continues to take steps, eventually some process will complete an operation. The main difficulty in performing memory reclamation for a lock-free data structure is that a process can be sleeping while holding a pointer to an object that is about to be freed. Thus, carelessly freeing an object can cause a sleeping process to access freed memory when it wakes up, crashing the program or producing subtle errors. Since nodes are not locked, processes must coordinate to let each other know which nodes are safe to reclaim, and which might still be accessed. (This is also a problem for data structures with lock-based updates and lock-free queries. The solutions presented hererin apply equally well to lock-based data structures with lock-free queries.)

Techniques for determining which nodes are safe to reclaim can be divided into *automatic techniques*, which do not require a programmer to specify when an object has been removed from the data structure, and *non-automatic techniques*, which do. It is conceptually useful to divide the work of reclaiming an object into two parts: determining when an object is removed from the data structure, and determining when this object can be freed to the operating system. The former is the responsibility of either an *automatic* memory reclamation scheme, or a lock-free data structure. The latter is always the responsibility of the memory reclamation scheme. Automatic techniques offer a significantly simpler programming en-

vironment, but can be inefficient. Non-automatic techniques are particularly useful when developing data structures for software libraries, since optimization can pay large dividends when code is reused. Additionally, non-automatic techniques can be used to build data structures that serve as building blocks to construct new automated techniques.

Garbage collectors comprise the largest class of automated techniques. The literature on garbage collection is vast, and lies outside the scope of this paper. Garbage collection schemes are surveyed in [16, 22]. Reference counting is another technique that can be automated. Limited forms of reference counting are also used to construct non-automatic techniques.

Non-automatic techniques can broadly be grouped into four categories: *object pools*, *hazard pointers*, *epoch based reclamation* and *transactional memory assisted reclamation*. As will be discussed in Section 3, these techniques either do not work for, or are inefficient for, many natural lock-free data structures.

In epoch based reclamation (EBR), the execution is divided into epochs. Each object removed from the data structure in epoch e is placed into a shared *limbo bag* for epoch e. Limbo bags are maintained for the last three epochs. Each time a process starts an operation, it reads and announces the current epoch, and checks the announcements of other processes. If all processes have announced the current epoch, then a new epoch begins, and the contents of the oldest limbo bag can be reclaimed. If a process sleeps or crashes during an operation, then no memory can be reclaimed, so EBR is not fault tolerant.

We begin by developing a more efficient, distributed variant of EBR, called DEBRA. DEBRA replaces the shared limbo bags of EBR with per-process bags, and amortizes the checking of processes' announced epochs by reading only one announcement at the start of each operation. Each limbo bag is implemented as a singly-linked list of *blocks*, which each contain 256 pointers to objects. Whenever a process announces a new epoch, it appends its oldest limbo bag to a local object pool in constant time. Whenever local pools grow too large or small, entire blocks are moved to or from a shared object pool. Operating on blocks instead of individual objects significantly reduces overhead. DEBRA performs O(1) steps at the beginning and end of each operation supported by the lock-free data structure and O(1) steps each time an object is removed from the data structure.

Our main contribution is DEBRA+, the first fault tolerant epoch based reclamation scheme. The primary challenge was to find a way to allow a process to advance the current epoch without waiting for another process, which may have crashed. In order to advance the epoch, we must ensure that such a process will not access any retired object if it takes another step. The technique we introduce for adding fault tolerance to DEBRA uses signals, an interprocess communication mechanism supported by many operating systems (e.g., Linux and UNIX). With DEBRA+, the number of objects waiting to be freed is $O(mn^2)$, where n is the number of processes and m is the largest number of objects removed from the data structure by one operation.

A major problem arises when auditioning non-automatic techniques to see which performs best for a given lock-free data structure. The set of operations exposed to the programmer by non-automatic techniques varies widely. This is undesirable for several reasons. First, from a theoretical standpoint, there is no reason that a data structure should have to be aware of how its objects are allocated and reclaimed. Second, it is difficult to interchange memory reclamation schemes to determine which performs best in a given situation. Presently, doing this entails writing many different versions of the data structure, each version tailored to a specific memory reclamation scheme. Third, as new advancements in memory reclamation appear, existing data structures have to be reimplemented (and their correctness painstakingly re-examined) before they can reap the benefits. Updating lock-free data structures in this way is non-trivial. Fourth, moving code to a machine with a different memory consistency model can require changes to the memory reclamation scheme, which currently requires changing the code for the lock-free data structure as well. Fifth, if several instances of a data structure are used for very different purposes (e.g., many small trees with strict memory footprint requirements and one large tree with no such requirement), then it may be appropriate to use different memory reclamation schemes for the different instances. Currently, this requires writing and maintaining code for different versions of the data structure.

These issues were all considered in the context of sequential data structures when the C++ standard template libraries were implemented. Their solution was to introduce an *Allocator* abstraction, which allows a data structure to perform *allocate* and *deallocate* operations. With this abstraction, the memory allocation scheme for a data structure can easily be changed without modifying or compiling (or even having access to) the data structure code. Unfortunately, the Allocator abstraction cannot be applied directly to lock- free programming, since its operations do not align well with the operations of lock-free memory reclamation schemes. (For example, it requires the data structure to know when an object can be freed.) The main challenge in developing an appropriate generalization of the Allocator abstraction is to find the right set of operations to expose to the data structure implementation, so that the result is simultaneously highly efficient, versatile and easy to program. In Section 6, we present the first generalization of the Allocator abstraction for lock-free programming.

Experiments on C++ implementations of DEBRA and DEBRA+ show that overhead is very low. Compared with performing no reclamation at all, DEBRA is on average 4% slower, and at worst 21% slower, over a wide variety of thread counts and workloads. In some experiments, DEBRA actually *improves* performance by as much as 20%. Although it seems impossible to achieve better performance when computation is spent reclaiming objects, DEBRA reduces the memory footprint of the data structure, which improves memory locality and cache performance. Adding fault tolerance to DEBRA adds 2.5% overhead, on average.

2. MODEL

We consider an asynchronous shared memory system with n processes. Each process has local memory that is not accessible by any other process, and there is a shared memory accessible by all processes. Memory is divided into primitive objects, which have atomic operations that are provided directly by the hardware. Examples include read/write registers, compare-and-swap (CAS) objects, and double-wide compare-and-swap (DWCAS) objects. A *record* is a collection of primitive objects, which we refer to as *fields*. A data structure consists of a fixed set of *entry points*, which are pointers to records, and the records that are reachable by following one or more pointers from an entry point. A record is *retired from a data structure* when it changes from being in the data structure to not being in the data structure. A record is *inserted into a data structure* when it changes from being not in the data structure to being in the data structure. The system is equipped with a *memory allocator*. Initially, no record is accessible to any process. *Allocating* a record makes it accessible and provides the process that requested it with a pointer to the record. A record can also be *freed*, which makes it inaccessible. Accessing a freed record results in the failure of the entire system.

3. RELATED WORK

There are many existing techniques for reclaiming memory in lock-free data structures. However, all of these techniques have some disadvantages.

Object Pools (OP). A particularly simple way to prevent processes from accessing a freed record is to never free any record, but allow processes to reuse retired records. Whenever a record is retired, it is returned to a pool of records (of the same type), from which it can be retrieved by any process that wants to allocate a record. Implementations of shared object pools are described in [14, 24]. If a shared pool is highly contended, then per-process pools can be added, so each process synchronizes on the shared pool only when one of its local pools is either empty or larger than some threshold.

Reference Counting (RC). RC augments each record o with a counter that records the number of pointers that processes, entry points and records have to r. An record can safely be freed once its reference count becomes zero. Reference counts are updated every time a pointer to a record is created or destroyed. Naturally, a process must first read a pointer to reach a record before it can increment the record's reference counter. This window between when a record is reached and when its reference counter is updated reveals the main challenge in designing a RC scheme: the reference count of a freed record must not be accessed. However, the reference count of a retired record can be accessed.

Detlefs et al. [6] introduced lock-free reference counting (LFRC), which is applicable to arbitrary lock-free data structures. LFRC uses the double compare-and-swap (DCAS) synchronization primitive, which atomically modifies two arbitrary words in memory, to change the reference count of a record only if a certain pointer still points to it. DCAS is not natively available in modern hardware, but it can be implemented from CAS. Herlihy et al. [15] subsequently improved LFRC to single-word lock-free reference counting (SLFRC), which uses single-word CAS instead of DCAS. To prevent the reference count of a freed record from being accessed, the implementation of SLFRC uses a variant of Hazard Pointers (described below) to prevent records from being freed until any pending accesses have finished. Lee [17] developed a distributed reference counting scheme from fetch-and-increment and swap. Each node contains several limited reference counts, and the true reference count for a node is distributed between itself and its parent. The scheme was developed for in-trees (wherein each node has a pointer only to its parent), but it may be generalizable.

RC requires extra memory for *each record* and it cannot reclaim records whose pointers form a cycle (since their reference counts will never drop to zero). Manually breaking cycles to allow reclamation requires knowledge of a data structure and adds more overhead. RC has high overhead, since following a pointer involves (at least) two costly atomic synchronization primitives; experiments confirm that RC is less efficient than other techniques [13].

Hazard Pointers (HPs). Michael introduced HPs [20], and provided a wait-free implementation from atomic read/write registers. (Herlihy et al. [15] independently developed another version of HPs, providing a lock-free implementation from CAS, and a wait-free implementation from double-wide CAS.) HPs track which records might be accessed by each process. Before a process can access a field of a record r, or use a pointer to r as the expected value for a CAS, it must first acquire a hazard pointer to r. To correctly use HPs, one must satisfy the following constraint. Suppose a record r is retired at time t_r and later accessed by a process p at time t_a. If r remains retired between t_r and t_a, then one of p's HPs must continuously point to r from before t_r until after t_a. It is easy to verify that, after a record r is retired and is not pointed to by any HP, no process can obtain a HP to r until it is next inserted into the data structure. Therefore, a process can safely free a retired record after scanning all HPs and seeing that none point to r.

To acquire a HP to r, a process first announces the HP by writing a pointer to r in a shared *announce* array with one part for each process. This announces to all processes that r might be accessed and cannot safely be freed. Then, the process verifies that r is in the data structure. If the record is not in the data structure, then the process can behave as if its operation had failed due to contention (typically by aborting and restarting its operation), without threatening the progress guarantees of the data structure. For many data structures, a process cannot easily determine whether a record is in the data structure, and they are modified in an ad-hoc way to allow operations to restart whenever they cannot tell. This requires reproving the data structures' progress guarantees.

Many lock-free algorithms require only a small constant number k of HPs per process, since a HP can be released once a process will no longer access the record to which it points during an operation. Scanning the HPs of all processes takes $\Theta(nk)$ steps, so doing this each time a record is retired would be costly. However, with a small modification, the expected amortized cost to retire an object is $O(1)$. Each process maintains a local collection of retired records. When a collection contains $nk + \Omega(nk)$ objects, the process creates a hash table T containing every HP, and, for each object o in its collection, checks whether o is in T. If not, o is freed. Since there are at most nk records in T, $\Omega(nk)$ records can be freed. Thus, the number of records waiting to be freed is $O(kn^2)$.

Problems with Hazard Pointers and marking. Many lock-free data structures use *marking* to prevent processes from erroneously performing modifications to records just before, or after, they are removed from the data structure. Specifically, before a record is removed from the data structure, all of its pointers are marked, and a process is allowed to change a pointer only if it is not marked. A subtle, but significant problem arises when one considers the interaction between HPs and marking in lock-free data structures that can contain a marked node that points to another marked node (e.g., [4, 5, 7, 10, 21, 23]). Specifically, it occurs when a process reads a marked pointer to a record and attempts to acquire a HP to it.

For example, consider a lock-free singly-linked list that marks a node's next pointer before removing it from the list. (It is easy to see that two consecutive nodes u and u' can be marked if one process marks u, in preparation to delete it, and another process simultaneously marks u'.) Suppose that, while searching this list, a process p has acquired a HP to node u and needs to obtain a HP to the next node, u'. To acquire the HP to u', p reads and announces the pointer to u', and must then verify that u' is in the list. If the pointer to u' is unmarked, then u is in the list, and, consequently, so is u'. If this pointer is marked and u' is not in the list, then p can behave as if its operation had failed, without threatening the progress guarantees of the list. However, behaving as if its operation had failed when this pointer is marked and u' is in the list might invalidate the progress guarantees of the list. If one does not want to reprove the data structure's progress guarantees, then p must find some way to determine whether u' is actually in the list.

By definition, u' is in the list if and only if it is reachable from an entry point or an unmarked node. One option is for p to traverse the list backwards from u until it reaches an unmarked node, and then search for u' from there. However, unless p has already acquired HPs to every node it will return to, it can encounter the same problem while moving backwards in the list. Furthermore, even after p reaches an unmarked node, it can encounter the same problem while searching for u' from that node. Each time p reads and an-

nounces a marked pointer, and recursively starts a search, its set of announcements grows. Consequently, the number of HPs p must hold at once (and, hence, the number of retired nodes that cannot be freed) can be arbitrarily large. This problem also occurs if p searches for u' from the beginning of the list, instead of traversing backwards. Additionally, for some data structures, the amortized cost of operations depends on the number of marked nodes they traverse. Searching from an entry point can provably increase the amortized cost of operations [9].

More about Hazard Pointers. Aghazadeh et al. [1] recently introduced an improved version of HPs with a worst case constant time procedure for scanning HPs each time a record is retired. Their algorithm maintains two queues of length nk for each process. These queues are used to incrementally scan HPs as records are retired. The algorithm adds a limited type of reference count to each record that tracks the number of incoming references from one of the queues. Note that $\Theta(\log nk)$ bits are reserved in each record for the reference count.

A technique called *Beware & Cleanup* (B&C) was introduced by Gidenstam et al. [12] to allow processes to acquire hazard pointers to retired records. A limited form of RC is added to HPs to ensure that a retired record is freed only when no other record points to it. The reference count of a record counts incoming pointers from other records, but does not count incoming pointers from processes' local memories. Before reclaiming a retired record, a process must check that its reference count is zero, and that no HP points to it. Consequently, after announcing a HP to a record r, it suffices to check that another record points to r. It follows that operations do not have to restart when they reach a retired record. The goal of the work was to optimize the performance of algorithms with searches that frequently encounter retired records and, consequently, have to restart. They do not identify the problem with HPs described above, but their technique solves it. However, B&C's algorithm for retiring records is very complicated, and the technique has higher overhead than either RC or HPs.

Hardware transactional memory (HTM). Dragojević et al. [8] explored simple schemes for memory reclamation using HTM. Alistarh et al. [2] introduced an elegant algorithm called StackTrack (ST). The key idea is to execute each operation of the lock-free data structure in a transaction, and use the implementation of HTM to automatically monitor all pointers stored in the private memory of processes without having to explicitly announce pointers before they are accessed. The HTM system will abort any transaction that accesses a record which is freed during the transaction. To decrease the probability of aborting transactions, each operation is split into many small transactions. This takes advantage of the fact that lock-free algorithms do not depend on transactions for correctness. Splitting operations into small transactions introduces overhead, and also causes some problems whose solutions introduce further overhead. For example, at the end of each transaction, any pointers that will be used by the next transaction are announced, so that they are not freed before the next transaction begins. Any time a process wants to free a record r, it must first verify that r is not announced.

Since currently available implementations of HTM do not offer progress guarantees, a *fallback path* that does not use HTM must be provided to make the algorithm lock-free. ST falls back to HPs. The paper does not give an upper bound on the number of retired records that are not freed. Although no such constraint is stated in the paper, ST cannot be applied to any data structure with an operation that atomically removes two or more records from the data structure [18]. Currently, ST requires a programmer to insert code before and after each operation, and after every few lines of lock-free data structure code (although, it should be possible to write a compiler to automate much of this).

Epochs. A process is in a *quiescent state* whenever it does not have a pointer to any record in the data structure. A grace period is any time interval during which every process has a point when it is in a quiescent state. Quiescent state-based reclamation (QSBR) [19] uses the fact that a record retired by a process can safely be freed after any subsequent grace period. Applying QSBR to a data structure requires manually identifying quiescent states, which may be impractical for some data structures. Additionally, QSBR is not fault-tolerant, because a stalled process can indefinitely prevent reclamation of any memory.

Fraser [11] described epoch based reclamation (EBR), which is similar to QSBR, but assumes that a process is in a quiescent state between its successive data structure operations. More specifically, EBR can be applied only if processes cannot save pointers read during an operation and access them during a later operation. EBR uses a single global counter, which records the current *epoch*, and an announce array. Each data structure operation first reads and announces the current epoch ε, and then checks whether all processes have announced the current epoch. If so, it increments the current epoch using CAS. The key observation is that the period of time starting from when the epoch was changed from $\varepsilon-2$ to $\varepsilon-1$ until it was changed from $\varepsilon-1$ to ε is a grace period (since each process announced a new value, and, hence, started a new operation). So, any records retired in epoch $\varepsilon-2$ can safely be freed in epoch ε. Whenever a record is retired in epoch ε, it is appended to a *limbo bag* for that epoch. It is sufficient to maintain three limbo bags (for epochs ε, $\varepsilon-1$ and $\varepsilon-2$, respectively). Whenever the epoch changes, every record in the oldest limbo bag is freed, and that limbo bag becomes the limbo bag for the current epoch.

Since EBR introduces overhead per *operation*, it is significantly more efficient than HPs, which requires costly synchronization for each pointer read. However, synchronizing on shared limbo bags makes it significantly less efficient than DEBRA. The penalty for writing to memory on a per operation basis, rather than a per pointer basis, is that processes have little information about which records might be accessed by a process that is suspected to have crashed. Consequently, EBR is not fault tolerant; no memory can be reclaimed after a process crash, and a slow process can prevent an unbounded number of retired records from being freed.

Drop the Anchor (DTA). Braginsky, Kogan and Petrank [3] introduced DTA, a specialized technique for singly-linked lists, which explores a middle ground between HPs and EBR. Instead of acquiring a HP each time a pointer to a node is read, a HP is acquired only once for every c pointers read. If another process q suspects that p has crashed, then q will *cut p out of the list* by replacing all nodes that p might access with new copies, and marking the old nodes so that p can tell what has happened if it eventually wakes up. This allows memory reclamation to continue, even if processes crash.

DTA has been shown to be efficient [2, 3]. However, it is not clear how it could be extended to work for other data structures. Additionally, DTA needs to be integrated with the mechanism for synchronizing updates to the linked list, because processes are cut out of the list concurrently with updates.

4. DEBRA: DISTRIBUTED EPOCH BASED RECLAMATION

Our new memory reclamation scheme, DEBRA, provides four operations: *leaveQstate*(), *enterQstate*(), *retire*(r) and *isQuiescent*(), where r is a record. Each of these operations runs in worst-case constant time. Let T be a lock-free data structure. To use DEBRA,

T simply invokes *leaveQstate* at the beginning of each operation, *enterQstate* at the end of each operation, and *retire*(r) each time a record r is retired (i.e., removed from T). Like EBR, DEBRA assumes that a process does not hold a pointer to any record between successive operations on T. Another constraint is that *retire*(r) can be invoked only once each time r is retired. Each process alternates invocations of *leaveQstate* and *enterQstate*, beginning with an invocation of *leaveQstate*. Each process is said to be quiescent initially and after invoking *enterQstate*, and is said to be non-quiescent after invoking *leaveQstate*. An invocation of *isQuiescent* by a process returns true if it is quiescent, and false otherwise.

In DEBRA, each process p has three limbo bags, denoted bag_0, bag_1 and bag_2, which contain records that it removed from the data structure. At any point, one of these bags is designated as p's limbo bag for the current epoch, and is pointed to by a local variable *currentBag*. Whenever p removes a record from the data structure, it simply adds it to *currentBag*. Each process has a *quiescent bit*, which indicates whether the process is currently quiescent. The only thing p does when it enters a quiescent state is set its quiescent bit. Whenever p *leaves* a quiescent state, it reads the current epoch e and announces it in $announce_p$. If this changes the value of $announce_p$, then the contents of the oldest limbo bag can be reused or freed. In this case, p changes *currentBag* to point to the oldest limbo bag, and then moves the contents of *currentBag* to an object pool. (Alternatively, the records in *currentBag* could be freed.) Next, p attempts to determine whether the epoch can be advanced, which is the case if each process is either quiescent or has announced e. Process p incrementally scans the announcements and quiescent bits of all processes, amortizing the cost over n *leaveQstate* operations. A local variable *checked* keeps track of the number of processes that p has verified are quiescent or have announced e since p last announced a new epoch. Once *checked* is n, p performs a CAS to increment the current epoch.

DEBRA reclaims a record only when no process has a pointer to it: Suppose p places a record r in limbo bag b at time t_1, and moves r from b to the pool at time t_2. Assume, to obtain a contradiction, that a process q has a pointer to r at time t_2. At time t_1, b is p's current limbo bag, and just before time t_2, b is changed from being p's oldest limbo bag to being p's current limbo bag, again. Thus, *currentBag* must be changed at least three times between t_1 and t_2. Since p changes *currentBag* only in an invocation of *leaveQstate* that changes $announce_p$, p must perform at least three such invocations between t_1 and t_2. The current epoch must change between any pair of invocations of *leaveQstate* that change $announce_p$, so the current epoch must change at least twice between t_1 and t_2. By definition, at some time between two changes of the current epoch, say from e to e' and from e' to e'', q must either be quiescent or have announced e'. Process q cannot announce e' prior to the current epoch changing to e', thus, it must announce e' between these two epoch changes. Since q can only announce an epoch when it is in a quiescent state, q must be quiescent at some point between t_1 and t_2. This means q must obtain its pointer to r by following pointers from an entry point after it was quiescent, which is after t_1. However, r is removed from the data structure before t_1, and, hence, it is no longer reachable by following pointers from an entry point, which is a contradiction.

The object pool shared by all processes is implemented as a collection of *n pool bags*, one per process, and one shared bag. Whenever a process moves a record to the pool, it places the record in its pool bag. If its pool bag is too large, it moves some records to the shared bag. Whenever a process wants to allocate a record, it first attempts to remove one from its pool bag. If its pool bag is empty, it attempts to take some records from the shared bag. If the process fails to take any record from the shared bag, then it will allocate a new record.

For efficiency, each pool bag and limbo bag is implemented as a *blockbag*, which is a singly-linked list of *blocks*. Each block contains a *next* pointer and up to B records. (In our experiments, $B = 256$.) The head block in a blockbag always contains fewer than B records, and every subsequent block contains B records. With this invariant, it is straightforward to design constant time operations to add and remove records in a blockbag, and to move all full blocks from one blockbag to another. This allows a process to move all full blocks of records in its oldest limbo bag to the pool highly efficiently after announcing a new epoch. However, if the process only moves *full* blocks to the pool, then this limbo bag may be left with some records in its head block. These records *could* be moved to the pool immediately, but it is more efficient to leave them in the bag, and simply move them to the pool later, once the block that contains them is full. One consequence of not moving these records to the pool is that each limbo bag can contain at most $B-1$ records that were retired two or more epochs ago. This does not affect correctness. The shared bag is implemented as a lock-free singly-linked list of blocks with operations to add and remove a full block. Moving entire blocks to and from the shared bag reduces the frequency with which processes must synchronize on the shared bag.

Operating on blocks instead of individual records significantly reduces overhead. However, it also requires a process to allocate and deallocate blocks. To reduce the number of blocks that are allocated and deallocated during an execution, each process has a bounded *block pool* that is used by all of its blockbags. Instead of deallocating a block, a process returns the block to its block bool. If the block pool is already full, then the block is freed. Experiments show that allowing each process to keep up to 16 blocks in its block pool reduces the number of blocks allocated by more than 99.9%. No blocks are allocated for the shared bag, since blocks are simply moved between pool bags and the shared bag.

Two additional optimization are made. First, the least significant bit of $announce_p$ is used as p's quiescent bit. This allows both values to be read and written atomically, which reduces the number of reads and writes to shared memory. Second, a process increments the current epoch only after invoking *leaveQstate* at least MIN_CALLS times, where MIN_CALLS is a constant (100 in our experiments). This is especially helpful when the number of processes is small. For example, in a single process system, without this optimization, the process will advance the epoch and try to move records to the pool at the beginning of every single operation, introducing unnecessary overhead.

5. ADDING FAULT TOLERANCE

The primary disadvantage of DEBRA is that a crashed or slow process can stay in a non-quiescent state for an arbitrarily long time. This prevents any other processes from freeing memory. Although we did not observe this pathological behaviour in our experiments, many applications require a bound on the number of records waiting to be freed.

Lock-free data structures are ideal for building fault tolerant systems, because they are designed to be provably fault tolerant. If a process crashes while it is in the middle of any lock-free operation, and it leaves the data structure in an inconsistent state, other processes can always repair that state. The onus is on a process that wants to access part of a data structure to restore that part of the data structure to a consistent state before using it. Consequently, a lock-free data structure always provides procedures to repair and access parts of the data structure that are damaged (by a process

```
1  process local variables:
2    descriptor *desc;
3  void signalHandler(args):
4    if (isQuiescent()) then
5      enterQstate();
6      siglongjmp(...);          // jump to recovery code
7  int doOperationXYZ(args):
8    ...                         // quiescent preamble
9    while (!done)
10     if (sigsetjmp(...))       // begin recovery code
11       if (isRProtected(desc)) done = help(desc);
12       RUnprotectAll();
13     else                      // end recovery code
14       leaveQstate();          // begin body
15       do search phase
16       initialize *desc
17       RProtect each record will be accessed, or used
             as the old value of a CAS, by help(desc)
18       RProtect(desc);
19       done = help(desc);
20       enterQstate();          // end body
21       RUnprotectAll();
22   ...                         // quiescent postamble
23   perform retire() calls
```

Figure 1: Applying DEBRA+ to a typical lock-free operation. Lines 14 and 20 are necessary for both DEBRA and DEBRA+. Gray lines are necessary for fault tolerance (DEBRA+).

crash) or undergoing changes. (Furthermore, these procedures are necessarily designed so that processes can crash while executing them, and other processes can still repair the data structure and continue to make progress.) We use these procedures to design a mechanism that allows a process to *neutralize* another process that is preventing it from advancing the epoch.

A novel aspect of DEBRA+ is our use of two features offered by Unix, Linux and other POSIX-compliant operating systems. The first is *signaling*, an inter-process communication mechanism. Signals can be sent to a process by the operating system, and by other processes. When a process receives a signal, the code it was executing is interrupted, and the process begins executing a *signal handler*, instead. When the process returns from the signal handler, it resumes executing from where it was interrupted. A process can specify what action it will take when it receives a particular signal by registering a function as its signal handler.

The second feature is *non-local goto*, which allows a process to begin executing from a different instruction, *outside* of the current function, by using two procedures: *sigsetjmp* and *siglongjmp*. A process first invokes *sigsetjmp*, which saves its local state and returns false. Later, the process can invoke *siglongjmp*, which restores the state saved by *sigsetjmp* immediately prior to its return, but causes it to return true instead of false. The standard way to use these primitives is with the idiom: "if (sigsetjmp(...)) alternate(); else usual();". A process that executes this code will save its state and execute usual(). Then, if the process later invokes *siglongjmp*, it will restore the state saved by *sigsetjmp* and immediately begin executing alternate().

At the beginning of each operation by a process q, q invokes *sigsetjmp*, following the idiom described above, and then proceeds with its operation. Another process p can interrupt q by sending a signal to q. We design q's signal handler so that, if q was interrupted while it was in a quiescent state, then q will simply return from the signal handler and resume its regular execution from wherever it was interrupted. However, if q was interrupted in a non-quiescent state, then it is neutralized: it will enter a quiescent state and perform a *siglongjmp*. Then, q will execute special *recovery code*, which allows it to clean up any mess it left because it was neutralized. Since q's signal handler performs *siglongjmp* only if q was interrupted in a non-quiescent state, q will not perform *siglongjmp* while it is executing recovery code. Hence, if q receives a signal while it is executing recovery code, it will simply return from the signal handler and resume executing wherever it left off.

Our technique requires the operating system to guarantee that, after a process p sends q a signal, the next time process q takes a step, it will execute its signal handler. (This requirement can be weakened with small modifications to DEBRA+, which are discussed in the full version of this paper.) With this guarantee, after p has sent a signal to q, it knows that q will not access any retired record until q has executed its recovery code and subsequently executed *leaveQstate*. Thus, as soon as p has sent q a signal, p can immediately proceed as if q is quiescent.

Recovery code must be tailored to the data structure, but it is straightforward for lock-free operations of the following form. Each operation is divided into three parts: a quiescent bookkeeping *preamble*, a non-quiescent *body*, and a quiescent bookkeeping *postamble*. Processes can be neutralized while executing in the body, but cannot be neutralized while executing in the preamble or postamble (because a process will not call *siglongjmp* while it is quiescent). Consequently, processes should not do anything in the body that will corrupt the data structure if they are neutralized part way through. Allocation, deallocation, manipulation of process-local data structures that persist between operations, and other non-reentrant actions should occur only in the preamble and postamble. Figure 1 shows how to apply DEBRA+ to such an operation. The remainder of this section explains the steps shown there.

The body proceeds by first reading some records, then initializing and announcing a special record (allocated in the preamble) called a *descriptor*, which describes how to perform the operation. It then executes a *help* procedure, which uses the information in the descriptor to perform the operation. We assume that the descriptor includes all pointers that the *help* procedure will follow (or use as the expected value of a CAS). The *help* procedure can also be used by other processes to help the operation complete. In a system where processes may crash, a process whose progress is blocked by another operation cannot simply wait for the operation to complete, so helping is necessary. The *help* procedure for any lock-free algorithm is always reentrant and idempotent, because, at any time, one process can pause, and another process can begin helping. Likewise, *enterQstate* must be both reentrant and idempotent.

We now describe recovery for an operation O performed by a process p. Suppose p receives a signal, enters a quiescent state, and then performs *siglongjmp* to begin executing its recovery code. Although there are many places where p might have been executing when it was neutralized, it is fairly simply to determine what action it should take next. The main challenge is determining whether another process already performed O on p's behalf. To do this, p checks whether it announced a descriptor for O before it was neutralized. If it did, then some other process might have seen this descriptor and started helping O. So, p invokes *help* (which is safe even if another process already helped O, since *help* is idempotent). Otherwise, p can simply restart the body of O.

DEBRA allows a non-quiescent process executing an operation to safely access any record that it reached by following pointers from an entry point during the operation. However, DEBRA does *not* allow quiescent processes to safely access any records. Once a process p has been sent a signal, other processes treat p as if were quiescent. Furthermore, p enters a quiescent state before executing recovery code, and it remains in a quiescent state throughout the recovery code. Consequently, after p starts executing its signal handler, it cannot safely access any records it has pointers to,

since they might already have been freed. However, the help procedure in the recovery code must access the descriptor record, and possibly some of the records to which it points. We use HPs in a very limited way to prevent these records from being freed by other processes before p has finished either its operation or its recovery code. This lets p safely run its recovery code in a quiescent state, so that other processes can continue to advance the current epoch and reclaim memory.

We now describe how HPs are used. Before p invokes *help* for a descriptor d, it must acquire HPs by invoking *RProtect* for each record that $help(d)$ will access, and finally invoking $RProtect(d)$. At the end of each operation (just after performing *enterQstate*), and at the end of its recovery code, p invokes *RUnprotectAll* to release all of its HPs. Before p can move a record r to the pool, it must first verify that no HP points to it by invoking $isRProtected(r)$, which returns true if some HP points to r, and false otherwise. When executing recovery code, p first invokes *isRProtected* to determine whether the last descriptor it announced is protected. If so, then each r in S is also protected, and p is ready to safely execute $help(d)$. Otherwise, p has not yet invoked $help(d)$, so no other process is aware of p's operation, and p can simply terminate its recovery code and restart its operation. Note that, since *RProtect* is performed while a process is non-quiescent, p might be neutralized while executing it. Hence, it must be reentrant and idempotent.

Using our new *neutralizing* mechanism, we can bound the number of records waiting to be freed. Each time a process p performing *leaveQstate* encounters a process q that is not quiescent and has not announced epoch e, p checks whether the size of its own current limbo bag exceeds some constant c. If so, p neutralizes q. After p's current limbo bag contains at least c elements, and p performs n more data structure operations, it will have performed *leaveQstate* n times, and each non-quiescent process will either have announced the current epoch or been neutralized by p. Consequently, p will advance the current epoch, and, the next time it performs *leaveQstate*, it will announce the new epoch and reclaim records. It follows that p's current limbo bag can contain at most $c + O(nm)$ elements, where m is the largest number of records that can be removed from the data structure by a high-level operation. Therefore, the total number of records waiting to be freed is $O(n(nm+c))$.

In DEBRA, all full blocks in a limbo bag are moved to the pool in constant time. In DEBRA+, records can be moved to the pool only if no HP points to them, so this is no longer possible. One way to move records from a limbo bag b to the pool is to iterate over each record r in b, and check if a HP points to r. To make this more efficient, we move records from b to the pool only when b contains $nk + \Omega(nk) + B$ records, where k is the number of HPs needed per process, and B is the maximum number of records in a block. Before we begin iterating over records in b, we create a hash table containing every HP. Then, we can check whether a HP points to r in O(1) expected time. We can further optimize by rearranging records in b so that we can still move full blocks to the pool, instead of individual records. To do this, we iterate over the records in b, and move all records that are pointed to by HPs to the beginning of the blockbag. All full blocks that do not contain a record pointed to by a HP are then moved to the pool (in constant time). Since there are at most nk HPs, and we scan only when b contains at least $nk + B + \Omega(nk)$ records, we will be able to move at least $max\{B, \Omega(nk)\}$ records to the pool. Thus, the expected amortized cost to move a record to the pool (or free it) is O(1).

One problem with DEBRA+ is that it seems difficult to apply to lock-based data structures, because it is dangerous to interrupt a process and force it to restart while it holds a lock. Of course, there is little reason to use DEBRA+ for a lock-based data structure, since locks can cause deadlock if processes crash. For lock-based data structures, DEBRA can be used, instead.

6. A LOCK-FREE MEMORY MANAGEMENT ABSTRACTION

There are many compelling reasons to separate memory allocation and reclamation from data structure code. Although the steps that a programmer must take to apply a non-automatic technique to a data structure vary widely, it is possible to find a small, natural set of operations that allows a programmer to write data structure code once, and easily plug in many popular memory reclamation schemes. In this section, we describe a record management abstraction, called a Record Manager, that is easy to use, and provides sufficient flexibility to support (all three versions of) HPs, DTA, EBR, DEBRA and DEBRA+. (ST is not supported because it currently requires a programmer to insert transactions throughout the code, and annotate the beginning and end of each stack frame.)

A Record Manager has three components: an Allocator, a Reclaimer and a Pool. The Allocator determines how records will be allocated (e.g., by individual calls to *malloc* or by handing out records from a large range of memory) and freed. The Reclaimer is given records after they are removed from the data structure, and determines when they can be safely handed off to the Pool. The Pool determines when records are handed to the Allocator to be freed, and whether a process actually uses the Allocator to allocate a new record.

We implement data structures, Allocators, Reclaimers and Pools in a modular way, so that they can be combined without recompiling them. This clean separation into interchangeable components allows, e.g., the same Pool implementation to be used with both a HP Reclaimer and a DEBRA Reclaimer. Modularity is typically achieved with inheritance, but inheritance introduces significant runtime overhead. For example, when the precompiled data structure invokes *retire*, it does not know which of the precompiled versions of *retire* it should use, so it must perform work at runtime to choose the correct implementation. In C++, this can be done more efficiently with *template parameters*, which allow a compiler to reach into compiled code and replace placeholder calls with calls to the correct implementations. Unlike inheritance, templates introduce no overhead, since the correct implementation is compiled into the code. Furthermore, if the correct implementation is a small function, the compiler can simply insert its code directly into the calling function (eliminating the function call altogether).

A programmer interacts with the Record Manager, which exposes the operations of the Pool and Reclaimer. A Pool provides *allocate* and *deallocate* operations. A Reclaimer provides operations for the basic events that memory reclamation schemes are interested in: starting and finishing data structure operations (*leaveQstate* and *enterQstate*), reaching a new pointer and disposing of it (*protect* and *unprotect*), and retiring a record (*retire*). It also provides operations to check whether a process is quiescent (*isQuiescent*) and whether a pointer can be followed safely (*isProtected*). Finally, it provides operations for making information available to recovery code (*RProtect*, *RUnprotectAll*, *isRProtected*).

Most of these are described in Section 4 and Section 5. We describe the rest here. *protect*, which must be invoked on a record r before accessing any field of r, returns true if the process successfully protects r (and, hence, is permitted to access its fields), and returns false otherwise. Once a process has successfully protected r, it remains protected until the process invokes $unprotect(r)$ or becomes quiescent. $isProtected(r)$ returns true if r is currently protected by the process.

Reclaimers for DEBRA and DEBRA+ are effectively described in Section 4 and Section 5. For these techniques, *unprotect* does nothing, and *protect* and *isProtected* simply return true. (Consequently, these calls are optimized out of the code by the compiler.) For HPs, *leaveQstate*, *RProtect* and *RUnprotectAll* all do nothing, and *isQuiescent* and *isRProtected* simply return false. *unprotect*(r) releases a HP to r, and *enterQstate* clears all announced HPs. *protect* announces a HP to a record and executes a function, which determines whether that record is in the data structure. *retire*(r) places r in a bag, and, if the bag contains sufficiently many records, it constructs a hash table T containing all HPs, and moves all records not in T to the Pool (as described in Section 3).

A predicate called *supportsCrashRecovery* is added to Reclaimers to allow a programmer to add crash recovery to a data structure without imposing overhead for Reclaimers that do not support crash recovery. For example, a programmer can check whether *supportsCrashRecovery* is true before invoking *RProtect*. The code statement "if (*supportsCrashRecovery*)" statically evaluates to "if (true)" or "if (false)" at compile time, once the Reclaimer template has been filled in. Consequently, these if-statements are completely eliminated by the compiler. In our experiments, this predicate is used to invoke *sigsetjmp* only for DEBRA+ (eliminating overhead for the other techniques).

7. EXPERIMENTS

We did experiments to compare the performance of various Reclaimers: DEBRA, DEBRA+, HP, ST and no reclamation (None). We used the Record Manager abstraction to perform allocation and reclamation for a lock-free balanced binary search tree [4]. This BST can contain marked nodes that point to other marked nodes, so ST cannot be used, and we must confront the problems described in Section 3 to apply HP. Properly dealing with HP's problems would be highly complex and inefficient, so we simply restart any operation that encounters a marked node. Consequently, applying HP causes the BST to lose its lock-free progress guarantee. To determine whether this significantly affects the performance of HP, we added the same restarting behaviour to DEBRA, and observed that its impact on performance was small.

Code for ST was graciously provided by its authors. They used a lock-based skip list to compare None, HP and ST. We modified their code to use a Record Manager for allocating and pooling nodes, and used it to compare None, DEBRA, HP and ST. The actual reclamation code for HP and ST is due to the authors of ST. Since the skip list uses locks, it cannot use DEBRA+.

We ran our experiments on an Intel i7 4770 machine with 4 cores, 8 hardware threads and 16GB of memory, running Ubuntu 14.04. (There is currently no system with more than 4 cores that provides HTM and, hence, supports ST.) All code was compiled with GCC 4.9.1-3 and the highest optimization level (-O3). Google's high performance Thread Caching malloc (tcmalloc-2.4) was used.

Our first experiment compared the overhead of performing reclamation for the various Reclaimers. In this experiment, each Reclaimer performed all the work necessary to reclaim nodes, but nodes were not actually reclaimed (and, hence, were not reused). The Record Manager used a *Bump Allocator*: each process requests a large region of memory from the operating system at the beginning of an execution, and then divides that region into nodes, which it allocates in sequence. Since nodes were not actually reclaimed, we eliminated the Pool component of the Record Manager. In this experiment, a data structure suffers the overhead of reclamation, but does *not* enjoy its benefits (namely, a smaller memory footprint and fewer cache misses).

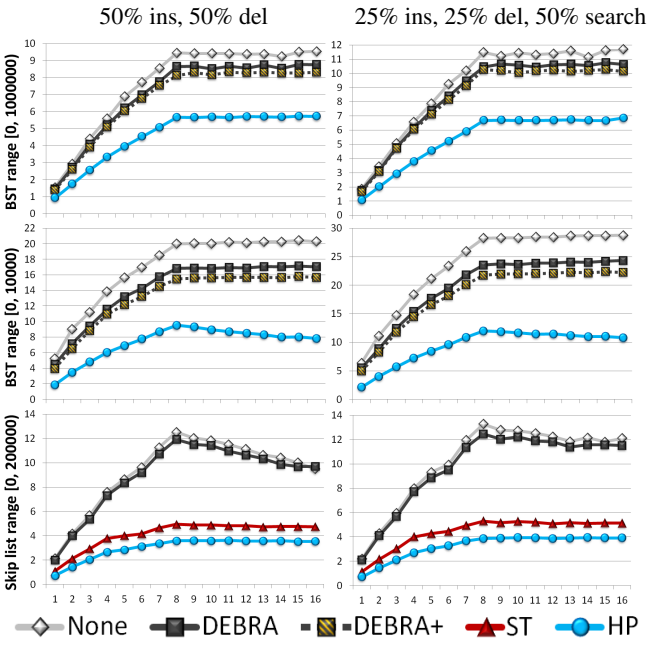

Figure 2: Results for Experiment 1 (Overhead of reclamation). The x-axis shows the number of processes. The y-axis shows throughput, in millions of operations per second.

For the balanced BST, we ran eight *trials* for each combination of Reclaimers in {None, DEBRA, DEBRA+, HP}, thread counts (in {1, 2, ..., 16}), operation mixes in {25i-25d, 50i-50d} (where xi-yd means x% insertions, y% deletions and $(100-x-y)$% searches) and key ranges in {[0, 10000), [0, 1000000)}. For the skip list, the thread counts and operation mixes were the same, but ST was used instead of DEBRA+, and there was only one key range, [0, 200000). In each trial, the data structure was first prefilled to half of the key range, then the appropriate number of threads performed random operations (according to the operation mix) on uniformly random keys from the key range for two seconds. The average of each set of eight trials became a data point in a graph. (Unfortunately, the system quickly runs out of memory when nodes are not reclaimed, so it is not possible to run all trials for longer than two seconds. However, we ran long trials for many cases to verify that the results do not change.)

The results in Figure 2 show that DEBRA and DEBRA+ have extremely low overhead. In the BST, DEBRA has between 5% and 22% overhead (averaging 12%), and DEBRA+ has between 7% and 28% overhead (averaging 17%). Compared to HP, on average, DEBRA performs 94% more operations and DEBRA+ performs 83% more. This is largely because DEBRA and DEBRA+ synchronize once *per operation*, whereas HP synchronizes each time a process reaches a new node. In the skip list, DEBRA has up to 6% overhead (averaging 4%), outperforms HP by an average of 200%, and also outperforms ST by between 93% and 168% (averaging 133%). ST has significant overhead. For instance, on average, it starts almost four transactions per operation (each of which announces one or more pointers), and runtime checks are frequently performed to determine if a new transaction should be started.

In our second experiment, nodes were actually reclaimed. The Reclaimers were each paired with the same Pool as DEBRA. The only exception was None, which does not use a Pool. (Thus, None is the same as in the first experiment.)

The results appear in Figure 3. In the BST, DEBRA is only 8% slower than None on average, and, for some data points, DEBRA actually improves performance by up to 12%. This is possible be-

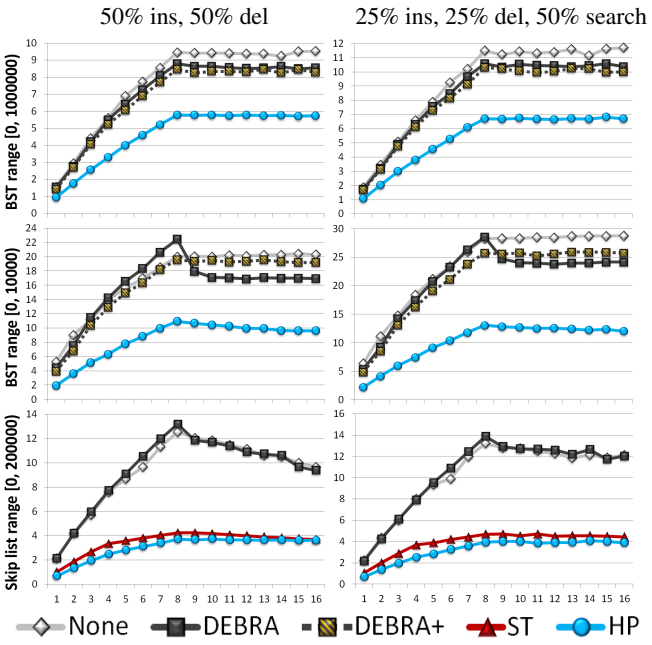

Figure 3: Experiment 2 (Using a Bump Allocator and a Pool).

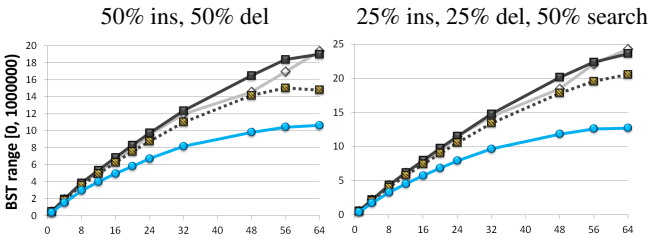

Figure 4: Extra results for Experiment 2 on Oracle T4-1.

cause DEBRA reduces the memory footprint of the data structure, which allows a larger fraction of the allocated nodes to fit in cache and, hence, reduces the number of cache misses. DEBRA+ is between 2% and 25% slower than None, averaging 10%. Compared to HP, DEBRA is between 48% and 145% faster, averaging 80%, and DEBRA+ is between 43% and 123% faster, averaging 76%. In the skip list, DEBRA performs *as well as None*. DEBRA also *outperforms* ST by between 108% and 211%, averaging 160%.

To measure the benefit of neutralizing slow processes, we tracked the total amount of memory allocated for records in each trial. Since we used bump allocation, this simply required determining how far each bump allocator's pointer had moved during the execution. Thus, we were able to compute the total amount of memory allocated *after* each trial had finished (without having any impact on the trial while it was executing). Figure 6 shows the total amount of memory allocated for records in the second experiment in the BST with key range 10,000 and workload 50i-50d. (The other cases were similar.) DEBRA, DEBRA+ and HP all perform similarly up to eight processes. However, for more than eight processes, some processes are always context switched out, and they often prevent DEBRA from advancing the epoch in a timely manner. DEBRA+ fixes this issue. With 16 processes, DEBRA+ neutralizes processes an average of 935 times per trial, reducing memory usage by an average of 94% over DEBRA.

We also ran the second experiment on a NUMA Oracle T4-1 system with 8 cores and 64 hardware contexts. Figure 4 shows a subset of the results (which is limited due to lack of space).

Our third experiment is like the second, except we used a different Allocator, which does not preallocate memory. The Allocator's

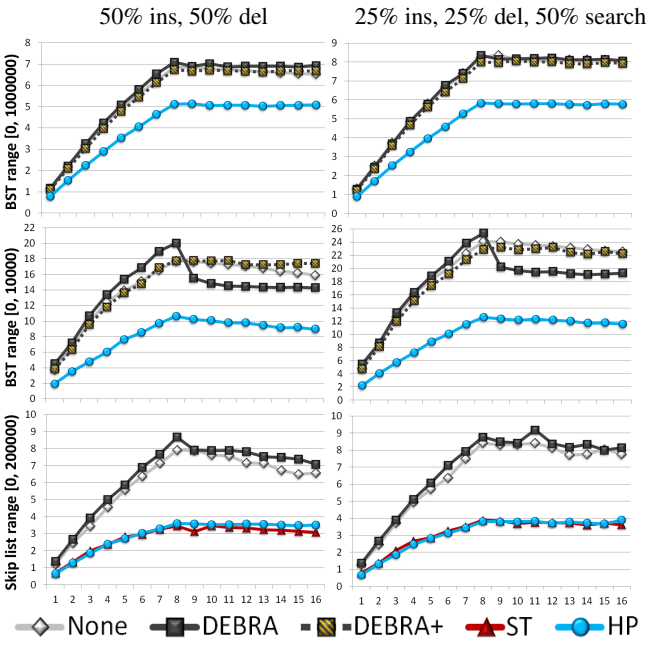

Figure 5: Experiment 3: (Using malloc and a Pool).

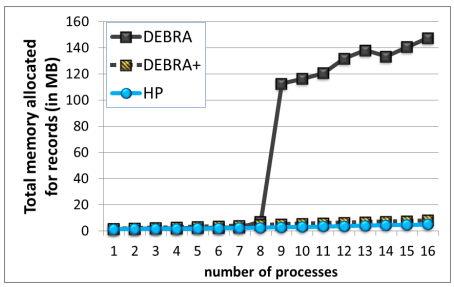

Figure 6: Memory allocated for records in Experiment 2 in the BST with keyrange 10,000 and workload 50i-50d.

allocate operation simply invokes *malloc* to request memory from the operation system (and its *deallocate* operation invokes *free*).

The results (in Figure 5) are similar to the results for the second experiment. However, the absolute throughput is significantly smaller than in the previous experiments, because of the overhead of invoking *malloc*. Although HP and ST are negatively affected, proportionally, they slow down less than None, DEBRA and DEBRA+. This illustrates an important experimental principle: *uniformly adding overhead to an experiment disproportionately impacts low-overhead algorithms, and obscures their advantage*.

8. CONCLUSION

In this work, we presented a distributed variant of EBR, called DEBRA. Compared to EBR, DEBRA significantly reduces synchronization overhead and offers high performance even with many more processes than physical cores. Our experiments show that, compared with performing no reclamation at all, DEBRA is 4% slower on average, 21% slower at worst, and up to 20% *faster* in some cases. Moreover, DEBRA outperforms StackTrack by an average of 138%. DEBRA is easy to use, and only adds O(1) steps per data structure operation and O(1) steps per retired record.

We also presented DEBRA+, the first epoch based reclamation scheme that allows processes to continue reclaiming memory after a process has crashed. In an n process system, the number of objects waiting to be freed is $O(mn^2)$, where m is the largest num-

ber of objects retired by one data structure operation. The cost to reuse or free a record is O(1) expected amortized time. In our experiments, DEBRA+ reduced memory consumption over DEBRA by 94%. Compared with performing no reclamation, DEBRA+ is only 10% slower on average. DEBRA+ also outperforms a highly efficient implementation of hazard pointers by an average of 70%.

We introduced the *Record Manager*, the first generalization of the C++ *Allocator* abstraction that is suitable for lock-free programming. A Record Manager separates memory reclamation code from lock-free data structure code, which allows a dynamic data structure to be implemented without knowing how its records will be allocated, reclaimed and freed. This abstraction adds virtually no overhead. It is highly flexible, allowing a programmer to interchange techniques for reclamation, object pooling, allocation and deallocation by changing one line of code.

Besides DEBRA and DEBRA+, the neutralizing technique introduced in this work is of independent interest. It would be useful to find different ways to neutralize processes, so, for example, the neutralizing technique could be used with different operating systems. There may also be opportunities to apply neutralizing in other contexts, such as garbage collection. Finally, it would also be interesting to understand whether these ideas can be extended to lock-based algorithms (even for a restricted class, such as reentrant and idempotent algorithms).

Acknowledgments

I would like to acknowledge my supervisor Faith Ellen for her insightful suggestions. I also extend my thanks to Ryan Johnson, whose advice and detailed knowledge of operating system mechanisms was invaluable to me. I thank Dave Dice for kindly running my experiments on the Oracle machine. Lastly, I thank the authors of StackTrack for sharing their code, and the anonymous reviewers for their helpful comments. This research was supported by the National Science and Engineering Research Council of Canada.

9. REFERENCES

[1] Z. Aghazadeh, W. Golab, and P. Woelfel. Making objects writable. In *Proceedings of the 2014 ACM symposium on Principles of distr. comp.*, pages 385–395. ACM, 2014.

[2] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the 9th European Conference on Comp. Sys.*, page 25. ACM, 2014.

[3] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the 25th ACM symposium on Parallelism in alg. and arch.*, pages 33–42. ACM, 2013.

[4] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 329–342. ACM, 2014.

[5] T. Brown and J. Helga. Non-blocking k-ary search trees. In *Principles of Distr. Sys.*, pages 207–221. Springer, 2011.

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

[7] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th annual ACM symposium on Principles and practice of parallel programming*, pages 343–356. ACM, 2014.

[8] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 99–108. ACM, 2011.

[9] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert. The amortized complexity of non-blocking binary search trees. In *Proc. of ACM Symp. on Princ. of Distr. Comp.*, PODC '14, pages 332–340, New York, NY, USA, 2014. ACM.

[10] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140. ACM, 2010.

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

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

[13] 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.

[14] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, Nov. 1993.

[15] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196, 2005.

[16] R. Jones and R. Lins. Garbage collection: Algorithms for automatic dynamic memory management. *John Wiliey & Sons Ltd., England*, 1996.

[17] H. Lee. Fast local-spin abortable mutual exclusion with bounded space. In *Principles of Distributed Systems*, pages 364–379. Springer, 2010.

[18] A. Matveev. Private communication.

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

[20] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.

[21] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th annual ACM symposium on Principles and practice of parallel programming*, pages 317–328. ACM, 2014.

[22] M. Schoeberl and W. Puffitsch. Nonblocking real-time garbage collection. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(1):6, 2010.

[23] N. Shafiei. Non-blocking patricia tries with replace operations. In *Distributed Comp. Sys. (ICDCS), 2013 IEEE 33rd Int. Conf. on*, pages 216–225. IEEE, 2013.

[24] R. K. Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.