הטכניון
מכון טכנולוגי לישראל
TECHNION
Israel Institute of Technology

ספריות הטכניון
The Technion Libraries

בית הספר ללימודי מוסמכים ע"ש ארווין וג'ואן ג'ייקובס
Irwin and Joan Jacobs Graduate School

Memory Management: From Theory to Practice

Nachshon Cohen

Memory Management: From Theory to Practice

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Nachshon Cohen

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tamuz 5776 Haifa July 2016

This research was carried out under the supervision of Prof. Erez Petrank, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period, the most up-to-date versions of which being:

Nachshon Cohen and Erez Petrank. Limitations of partial compaction: towards practical bounds. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI'13*, pages 309–320, 2013.

Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures - SPAA'15*, pages 254–263, 2015.

Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPLSA'15*, pages 260–279, 2015.

Nachshon Cohen and Erez Petrank. Data structure aware garbage collector. *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management - ISMM'15*, pages 28–40, 2015.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Erez Petrank, for the guidance during my doctoral studies. Your faith in my research, your kind guidance, and your insistence on providing complete and clear results were greatly helpful. I wish to say a special thank you for your availability. The best results in this thesis started out by simply storming to your office with crazy new ideas. Without being able to discuss these ideas and to explore them further, this thesis would not come to completion.

I would also like to thank my parents for allowing me to be curious and helping me explore new ideas.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

Today, all programming languages provide dynamic memory allocation. The application allocates memory when data arrives; this memory is recycled when the application no longer needs it. While significantly simplifying programming, recycling memory has nontrivial costs. In this dissertation, we study these costs, mainly in the context of today's multi-core machines. As these machines become ubiquitous, it is important to improve the efficiency of applications running on top of them.

Lock-free data structures serve as an important tool in the design of applications running on multi-core machines. These data structures provide a progress guarantee: eventually at least one thread will make progress. In practice, they provide excellent scalability and performance. But while lock-free data structures are widely accepted, providing memory management support for them (without foiling their progress guarantee) remains difficult. Existing techniques, such as the use of hazard pointers, may impose a high performance overhead. In addition, applying them to a given data structure is difficult. All known lock-free reclamation schemes are "manual" in the sense that the developer has to specify when nodes retire and may be safely reclaimed. Retiring nodes adequately is non-trivial and it sometime requires the modification of the original lock-free algorithm.

We begin by proposing a novel memory management scheme for lock-free data structures called *optimistic access*. This scheme provides efficient memory management support for lock-free data structures that can be presented in a normalized form. Our novel memory manager breaks the traditional memory management invariant, which never lets a program touch reclaimed memory. This broken invariant provides an opportunity to obtain high parallelism with excellent performance, but also requires careful design. Measurements show that it dramatically outperforms known lock-free memory reclamation methods.

We then extend the optimistic memory management scheme to be *automatic* in the spirit of mark-sweep garbage collection. The proposed algorithm works with any normalized lock-free algorithm and without requiring the programmer to manually retire nodes. Evaluation shows that this automation comes at a very low cost.

Next, we consider state-of-the-art garbage collectors running on multi-core platforms. Garbage collection may benefit greatly from knowledge about program behavior, but most managed languages do not provide means for the programmer to deliver such knowledge. We propose a very simple interface that requires minor programmer effort and achieves substantial performance and scalability improvements. Experience shows that this interface requires minor

modifications to the application. Measurements show that for some significant benchmarks this interface can dramatically reduce the time spent on garbage collection and also improve the overall program performance.

The last contribution of this dissertation is a theoretical study about the limits of compaction. To avoid fragmentation, garbage collectors employ compaction. However, compaction is a costly operation that commercial runtimes attempt to avoid. Instead, partial compaction is often used to defragment parts of the heap. Previous studies on the limitations of compaction provided some initial asymptotic bounds but no implications for practical systems. In this work, we extend the theory to obtain better bounds and make them strong enough to become meaningful for modern systems.

Chapter 1

Introduction

Almost all programming languages make use of dynamic memory allocation. This allows the programmer to dynamically respond to user input, simplifies the design of recursive data structures, and relieves the programmer from the burden of explicitly managing memory in long-running applications. Dynamic memory allocation is provided by the system's memory manager. It also handles memory recycling, which is required to avoid exhausting the memory.

Memory managers can be *manual* or *automatic*. The two types differ significantly in the way they work and the service they provide to the programmer. Manual memory managers require the programmer to identify which objects (chunks of memory) are no longer being used. Manual memory managers are thus (relatively) simple to design, allowing for faster implementation and easier maintenance. However, manually managing memory requires much more effort from the application programmer and typically makes debugging significantly harder. Automatic memory managers, also known as *garbage collectors*, automatically identify objects that are not used by the program and reclaim them. Automatic memory managers are widely acknowledged as greatly simplifying the task of the programmer and are considered as a major factor driving the adoption of managed languages. However, they are typically harder to develop and may incur higher overhead [JHM11, BCM04].

Our study is aimed at improving memory management efficiency on multi-core machines. Multi-core machines provide multiple computation units on which multiple threads (instruction streams) can be executed. The threads synchronize by accessing the shared memory, which is shared across all threads in the application. The rapid deployment of parallel machines shifts the focus toward *scalable* solutions that are able to effectively utilize the multiple threads supported by the machine without introducing too much overhead on synchronization.

To simplify the design of scalable algorithms, programmers often employ *concurrent data structures* , which allow threads to interact via the data structure's well-defined interface. Thus, the scalability and efficiency of concurrent data structures are crucial to parallel applications. In this dissertation, we focus on the interaction between concurrent data structures and memory management. We study efficient memory management techniques for concurrent data structures. We further study how data structures can improve the scalability and efficiency of garbage collection algorithms. Finally, we look at a theoretical study of compaction, which is of general

interest to the memory management field.

1.1 Non-blocking Synchronization

An important property of concurrent data structures is their progress guarantee. A progress guarantee ensures that threads can advance their execution in completely asynchronous systems. There are three types of progress guarantees: *obstruction-freedom*, *lock-freedom* (a.k.a. *non-blocking*), and *wait-freedom* [Her91, HS12]. Among these, lock-freedom is the most common. It guarantees that at least one thread will be able to progress eventually, even in the existence of an adversarial scheduler. Lock-free data structures cannot use the simple mutual exclusive paradigm. Instead, they use advanced synchronization techniques that allow threads to execute correctly despite the existence of concurrent threads operating on the same data, and despite the possibility of conflicts.

A basic building block in the design of lock-free data structures is the compare-and-swap (CAS) instruction. The CAS instruction modifies a memory location if and only if its current value matches the expected value. Herlihy [Her91] showed that every algorithm can be designed using the CAS instruction by using a universal construction. However, universal constructions yield very poor performance [TP14]. Ad hoc lock-free data structures use the CAS instruction to build highly efficient and scalable solutions [Har01, Mic02, FR04, MNSS05, BCCO10, BP12, NM14, DVY14, MA15]. Such data structures are typically built to apply each operation by a single CAS instruction; this structure was formalized by [TP14]. This CAS instruction is preceded by a preparation phase, which searches for the correct location to apply the operation and prepares the CAS instruction to apply. After the CAS is applied, a wrap-up method is executed to clean the data structure and optimize its performance. In the case of a race condition, the typical behavior is to restart the operation from an appropriate location.

1.1.1 Memory Management Support for Lock-free Data Structures

A major problem in the design of lock-free data structures is memory management. Since lock-free data structures do not use the mutual exclusive paradigm, it is usually impossible to know whether a specific (deleted) node is still accessible by other threads. The problem is that reclaiming a node that is later accessed by other threads may affect these threads in unpredictable ways, thus harming correctness. Due to the complexity of the memory reclamation problem, many lock-free data structures presented in the literature are measured without memory management support. This prevents such schemes from being employed by long running applications, which must rely on memory reclamations.

As a solution to this problem, Michael [Mic04] presented the manual reclamation scheme paradigm. Using this paradigm, the program first deletes the node and unlinks it from the data structure. Then it passes the unlinked node to the memory management scheme through the retire function. This node is not reclaimed immediately. Instead, the memory management scheme defers the actual reclamation of retired nodes to a time when the reclamation is safe. While

several memory reclamation schemes exist[Val95, Sun05, Har01, BKP13, DHY10, Mic04], they either break the lock-freedom guarantee of the data structure or add significant overhead. In this dissertation, we present a manual memory reclamation scheme that outperforms existing schemes while still preserving lock-freedom. This scheme is presented in Chapter 2.

While manual reclamation schemes provide memory management support for lock-free data structures, they rely on programmer effort to identify unlinked (fully removed) nodes. Surprisingly, identifying unlinked nodes turns out to be extremely hard, and even impossible for some widely used data structures. In this dissertation, we propose an automatic reclamation scheme: a lock-free memory management scheme that does not rely on programmer effort for memory management. This scheme is presented in Chapter 3.

1.2 Garbage Collection

Garbage collection is used by managed languages to relieve the programmer of the task of memory reclamation. While it is desirable to reclaim all objects that will never be used by the program, it is impossible to determine whether an object will be accessed. Instead, garbage collectors operate (conservatively) by identifying and reclaiming objects that are unreachable by the application. The vast majority of garbage collectors operate in phases, or *cycles*. In each cycle, the garbage collector collects the set of the application roots by traversing all global/static pointers and all thread-local pointers. Objects unreachable from the roots are unreachable by the application.

Once root objects are identified, there are three basic algorithms for reclaiming the set of objects unreachable from the roots: mark-sweep, copying, and reference counting [JHM11]. In the mark-sweep algorithm, the garbage collector performs graph tracing to identify the set of objects reachable from the root. All objects that were not marked during this traversal are reclaimed. Another algorithm is copying. A copying garbage collector copies all reachable objects to a specific location in the heap called to-space. Then, the entire memory where the original objects resided is reclaimed. Finally, a reference counting garbage collector keeps for each object a count of the number of live objects referencing it. An object is reclaimed once its reference count drops to zero[1]. Garbage collection algorithms that employ multiple base algorithms also exist. The best-known example is the generational garbage collector, which uses the copying algorithm for newly allocated objects (young generation) and the mark-sweep algorithm for other objects (mature). Other algorithms also exist; see [JHM11] and the references therein. However, they are outside the scope of this dissertation.

1.2.1 Parallel Garbage Collection

Parallel garbage collectors utilize multiple threads to finish the reclamation task quickly. In the mark-sweep garbage collector, the most time-consuming task is the tracing, which traces and

[1] Reference counting can be implemented without traversing the set of roots. However, this results in an inefficient design. All efficient reference counting algorithms work in cycles (phases) and traverse the set of roots at the beginning of each cycle.

marks all objects reachable from the roots. This task is also the most challenging to parallelize.

In parallel garbage collection, tracing is done concurrently by multiple threads. To allow all threads to participate in the tracing algorithm, the *markstack*, the set of objects to be traced, is shared between the threads. However, fully sharing the markstack requires synchronization on every access to the markstack, which is too expensive. To reduce the overhead of synchronization, the markstack is only partially shared. Some of the traced objects reside in a shared pool while others are owned by a specific thread (sometimes in a stealable queue that allows other threads to steal such objects).

While parallel tracing algorithms have been extensively studied and are widely used in practice, they still have limited scalability, both theoretically and practically. In theory, even when employing an optimal synchronization primitive and an infinite number of threads, a graph cannot be traced in a single step [BP10]. Instead, it takes time proportional to the depth of the graph, the maximal length of the shortest path from a root to a descendant. In high-depth graphs, no parallel algorithm can offer perfect scalability. For example, the existence of a large linked list in the application may negatively impact the scalability of the garbage collection.

In practice, measurements show that graph depth is typically small compared to the graph size, so the theoretical bound is not reached in today's systems [EP13, MRM+12]. However, parallelizing the tracing algorithm presents an additional crucial challenge: reducing the overhead of synchronization between the participating threads. To this end, tracing algorithms use heuristics that attempt to balance workload while reducing the interaction between threads. But these heuristics essentially depend on the shape of the heap. A heuristic may perform poorly in some cases, resulting in high synchronization overhead or unequal work balancing, leading to long delays and reduced application efficiency.

In this dissertation, we study a solution to this problem that relies on the centrality of data structures for applications. For various important programs, data structure nodes constitute a significant portion of the live objects. We show that programmers can easily inform the garbage collector about the data structures the application is using. The garbage collector can use this information to speed up the tracing algorithm by significantly improving scalability and locality while reducing synchronization overhead. These programmer hints even break the theoretical bound on parallel tracing of a high-depth heap, thus allowing the garbage collector to parallelize its work simply and efficiently even in this case. This work is presented in Section 4.

1.2.2 Fragmentation and Compaction

Fragmentation is the state where the free space is scattered around the heap in small chunks. This has various bad effects on the program, including the inability to allocate large objects, reducing locality, and increasing allocator overhead. To reduce fragmentation, the live objects can be compacted to a part of the memory, while leaving the other parts unused. However, compaction is an expensive operation [JHM11, AOPS04, KP06a]. Thus, garbage collection algorithms often employ partial compaction, which limits the amount of compaction used during the garbage collection process [BYGK+02, BCR03, CTW05, DFHP04, PPS08].

Understanding the theoretical bounds on fragmentation can assist designers of real-time systems, where timely responses are crucial. In such systems, the inability to allocate large objects may lead to application crashes and potentially disastrous implications. Furthermore, understanding the theoretical properties of garbage collection may help practitioners design their systems.

When unlimited compaction is allowed, there is no fragmentation problem. When compaction is not allowed at all, Robson [Rob71, Rob74] computed the maximum amount of fragmentation. However, not much is known about the middle case, where only partial compaction is allowed [BP11]. In this dissertation, we study a lower bound on the amount of fragmentation, given that compaction is allowed but restricted. This result appears in Section 5.

Chapter 2

Efficient Memory Management for Lock-Free Data Structures with Optimistic Access

2.1 Background

The rapid deployment of highly parallel machines has resulted in the acute need for parallel algorithms and their supporting parallel data structures. *Lock-free* data structures (a.k.a. *non-blocking*) [Her91, HS12] are immune to deadlocks and livelocks, fast, scalable and widely used in practice. However, as discussed in the Introduction, the challenge of their memory reclamation must also be addressed. The problem arises when one thread attempts to reclaim an object while another thread is still using it. Accounting for all accesses of all threads before reclaiming an object is difficult and costly, especially when threads may be delayed for a while while still holding pointers to nodes in the shared memory.

One easy approach to this problem is to not reclaim memory at all during the execution. But this solution is only applicable to short-running programs. Another approach to reclaiming memory is to assume automatic garbage collection, which guarantees that an object is never reclaimed while it is being used. However, this only delegates the problem to the garbage collector. There has been much work on garbage collectors that obtain some partial guarantee for progress [HM92, HM01, PFPS07, PPS08, ABC$^+$08, PZM$^+$10], but current literature offers no garbage collection that supports lock-free execution [Pet12].

A different approach is to coordinate the accessing threads with the threads that attempt reclamations. The programmer uses a memory management interface to allocate and reclaim objects and the reclamation scheme coordinates the memory recycling of reclaimed objects with the accessing threads. The most popular schemes of this type are *hazard pointers* and *pass the buck* [Mic04, HLMM05]. These (similar) methods require that each thread announces each and every object it accesses. To properly announce the accessed objects, a memory fence must be used for each shared memory read, which is costly. Employing one of these schemes for

a linked list may slow its execution down by a factor of 5 [BKP13]. To ameliorate this high cost, a recent extension by Braginsky et al. [BKP13] proposed the *anchors* scheme, which is a more complex method that requires a fence only once per several accesses. The *anchors* scheme reduces the overhead substantially, but the cost still remains non negligible. Furthermore, the *anchors* scheme is difficult to design and it is currently available for Harris-Michael linked list [Mic02] data structure only.

All known memory management techniques, including garbage collection and the above ad-hoc reclamation methods, provide a guarantee that a thread never accesses a reclaimed object. Loosely speaking, supporting this guarantee causes a significant overhead, because whenever a thread reads a pointer to an object, the other threads must become aware of this read and not reclaim the object. For an arbitrary program, this might mean a memory fence per each read, which is very costly. For more specialized programs or for specific lock-free data structures, better handling is possible, but a substantial performance penalty seems to always exist.

We propose to deviate from traditional methods in a novel manner by letting the program execute optimistically, allowing the threads to sometimes access an object that has been previously reclaimed. Various forms of optimistic execution have become common in the computing world (both hardware and software) as a mean to achieve higher performance. But optimistic access has never been proposed in the memory management literature due to the complications that arise in this setting. Optimistically accessing memory that might have been reclaimed requires careful checks that must be executed at adequate locations; and then, proper measures must be taken when the accessing of a reclaimed object has been detected. When a thread realizes that it has been working with stale values, we let it drop the stale values and return to a point where the execution is safe to restart.

Achieving such timely checks and a safe restart in this setting is quite difficult for arbitrary lock-free programs. Therefore, we chose to work only with lock-free data structures that can be presented in a *normalized form*. We used the normalized form proposed in [TP14]. This normalized form is on the one hand very general: it covers all concurrent data structure that we are aware of. On the other hand, it is very structured and it allows handling the checks and restarts in a prudent manner. As with other optimistic approaches, we found that the design requires care, but when done correctly, it lets the executing threads run fast with low overhead. We denote the obtained memory reclamation scheme *optimistic access*.

Measurements show that the overhead of applying the optimistic access scheme is never more than 19% compared to no reclamation, and it consistently outperforms the hazard pointers and anchors schemes. Moreover, it is easy to apply the optimistic access method to a normalized lock-free data structure[1]. The optimistic access mechanism is lock-free and it may reclaim nodes even in the presence of stuck threads that do not cooperate with the memory reclamation process.

In order for the optimistic access to be possible at all, the underlying operating system and runtime are required to behave "reasonably". The specific required assumptions are detailed in

[1] Given proper retire calls, which are required by all existing memory management schemes; see discussion in Section 2.3.3.

Section 2.3. Loosely speaking, the assumption is that reading or writing a field in a previously allocated object does not trigger a trap, even if the object has been reclaimed. For example, a system in which a reclaimed object is returned to the operating system and the operating system unmaps its memory thereafter, is not good for us since reading a field of that object would create a segmentation fault and an application crash.[2] It is easy to satisfy an adequate assumption by using a user-level allocator. This may be a good idea in general, because a user-level allocator can be constructed to provide a better progress guarantee. For example, using an object pooling mechanism for the nodes of the data structure would be appropriate.

The main contribution of this paper is an efficient memory reclamation scheme that supports lock-freedom for normalized lock-free data structures. The proposed scheme is much faster than existing schemes and is easy to employ. We exemplify the use of the optimistic access scheme on a linked list, a hash table, and a skip list. The obtained memory recycling scheme for the **skip list** incurred an overhead below 12%, whereas the overhead of the hazard pointers scheme always exceeded a factor of 2. For the **hash** table, the optimistic access scheme incurred an overhead below 12%, whereas the overhead of the hazard pointers method was $16\% - 40\%$ for $1 - 32$ threads (and negligible for 64 threads). For **linked list**, the optimistic access method always outperforms the hazard pointers and the anchors mechanisms. The optimistic access method typically incurs an overhead of a few percents and at a worst setting it incurs an overhead of 19%. The hazard pointers mechanism typically incurs a large overhead of up to 5x. The anchors mechanism improves performance significantly over the hazard pointers but with short lists and high contention it incurs a significant overhead as well.

2.2 Overview

Let us start with an intuitive overview of the optimistic access scheme. The main target of this scheme is to provide fast reads, as reads are most common. In particular, we would like to execute reads without writing to the shared memory. On the other hand, a lock-free memory reclamation scheme must be able to reclaim memory, even if some thread is stuck just before a read of an object that is about to be reclaimed. Thus, we achieve fast reads by allowing a thread to sometimes read an object after it was reclaim (and allocated to other uses).

The optimistic access scheme maintains correctness in spite of reading reclaimed objects using three key properties. First, a read *must not fault*, even when accessing a reclaimed memory. Second, the scheme *identifies* a read that accesses a reclaimed object immediately after the read. Third, when a read of such stale value is detected, the scheme allows a *rollback* of the optimistic read. We follow by describing how these three properties can be satisfied.

The first requirement is obtained by the underlying memory management system. We will require that accessing previously allocated memory will never cause a fault. This can be supported by using user-level allocators that allocate and de-allocate without returning pages

[2] As an aside, we note that the implementation of unmap is typically not lock-free and it is not to be used with lock-free data structures. For example, in the Linux operating system, an unmap instruction both acquires a lock and communicates with other processes via an interprocess interrupt.

to the system. Such allocators can be designed to support lock-free algorithms. (Typically, returning pages to the system foils lock freedom.)

Jumping to the third property, i.e., the roll back, we first note that the ability to roll back is (informally) made possible in most lock-free data structures. Such data structures handle races by simply restarting the operation from scratch. The same restarting mechanism can be used to handle races between data-structure operations and memory reclamation; indeed, such a roll-back mechanism is assumed and used in previous work (e.g., [Mic04]). However, to formally define a roll-back (or restart) mechanism, we simply adopt the normalized form for lock-free data structures [TP14]. This normalized form is on one hand very general - it covers all data structure we are aware of. On the other hand, its strict structure provides a well-defined restart mechanism, which can be used for rolling back the execution when a stale value has been read.

Next we discuss how to satisfy the second property, i.e., noting that a stale read has occurred due to a race between the read and memory reclamation. The optimistic access scheme divides the memory reclamation into phases, which may be thought of as epochs, and poses the following restrictions. First, an object is never reclaimed at the same phase in which it is unlinked from the data structure. It can only be reclaimed at the next phase or later. Second, a thread that acknowledges a new phase does not access objects that were unlinked in previous phases. These two restrictions provide a lightweight mechanism to identify a potential read of a stale value. If a thread is not aware that a phase has changed, then his read may potentially be of a stale value. Otherwise, i.e., if the thread is aware of the current reclamation phase, then his read is safe.

To make the (frequent) read operation even lighter, we move some of the related computation work to the (infrequent) reclaiming mechanism. To this end, each thread is assigned with an associated warning flag that a phase has changed. This flag is called the *warning-bit*. This bit is set if a new phase had started without the thread noticing, and clear otherwise. During a phase change the warning bits of all threads are set. When a thread acknowledges a phase change it resets its bit. This way, checking whether a read might have read a stale value due to reclamation, is as simple as checking whether the flag is non-zero.

To summarize, reading of shared memory is executed as follows. First the shared memory is read. Next, the thread's warning-bit is checked. Finally, if the warning bit is set, a restart mechanism is used to roll back the execution to a safe point.

We now deal with program writes. We cannot allow an event in which a thread writes to an object that has previously been reclaimed. Such an event may imply a corruption of objects in use by other threads. Therefore, for writes we adopt a simplified version of the hazard pointers scheme that prevents writes to reclaimed objects. A thread declares a location it is about to write to in a hazard pointer. Reclamation is avoided for such objects. Since writes are less frequent, the overhead of hazard pointers for writes is not high. The warning flag allows a quick implementation, as explained in Section 2.4 below.

Finally, it remains to describe the the memory reclamation scheme itself. A simplified version of such an algorithm may work as follows. It starts by incrementing the phase number, so that it can identify objects that were unlinked before the reclamation started. It can then

reclaim all objects that were unlinked in previous phases and are not pointed by hazard pointers.

The problem with the simple solution is that each thread that starts reclamation will increment the phase number and trigger restarts by all other threads. This should not happen too frequently. To reduce this overhead, we accumulate retired objects in a global buffer and let a reclaiming thread process objects unlinked by *all* threads. This reduces the number of phase changes and hence also the number of restarts. Even when using global pools, the optimistic access scheme naturally benefits from using temporary local pools that are used to reduce the contention on the global pools. Performance is somewhat reduced when space is limited and measurements of the tradeoff between space overhead and time overhead are provided in Section 2.5.

Advantage of the optimistic access scheme. Hazard pointers and anchors require an involved and costly read barrier that runs a verification process and a memory fence. In contrast, ours scheme works with a light-weight read barrier (that checks the warning bit). Hazard pointers are used for writes in a ways that is easy to install (practically, automatic), and being used for writes only, hazard pointers also incur a low overhead, as shown by the measurements.

2.3 Assumptions and Settings

In this section we specify the assumption required for our mechanism to work and define the normalized representation of data structures. Finally, in Subsection 2.3.4 we present a running example: the delete operation of Harris-Michael linked list.

2.3.1 System Model

We use the standard computation model of Herlihy [Her91]. A shared memory is accessible by all threads. The threads communicate through memory access instructions on the shared memory; and a thread makes no assumptions about the status of any other thread, nor about the speed of its execution. We also assume the TSO memory model, used by the common x86 architecture [OSS09].

Finally, as discussed in Section 2.2, we assume that accessing previously allocated memory does not trigger traps. Formally, we assume the following of the underlying system.

Assumption 2.3.1. Suppose a memory address p is allocated at time t. Then, if the program at time $t' > t$ executes an instruction that reads from p, then the executing of this instruction does not trigger a runtime-system trap.

2.3.2 Normalized Data Structures

The optimistic access scheme assumes that the data structure implementation is given in a normalized form. In this subsection we provide a motivating discussion and an overview over normalized representation. The formal definition following [TP14] is provided in Appendix A.1. The memory management scheme proposed in this paper lets threads infrequently access

reclaimed space. When this happens, the acting thread will notice the problem thereafter and it will restart the currently executing routine. The strict structure of the normalized algorithm provides safe and easily identifiable points of restart. Let us now informally explain how a normalized implementation looks like.

Loosely speaking, a normalized implementation of a data structure partitions each operation implementation into three parts. The first part, denoted the *CAS generator*, prepares a list of CASes that need to be executed for the operation. It may modify the shared data structure during this process, but only in a way that can be ignored and restarted at any point, typically these modifications improve the underlying representation of the data structure without changing its semantics[3]. The second part, denoted the *CAS executor*, attempts to execute the CASes produced by the CAS generator one by one. It stops when a CAS fails or after all have completed successfully. The third part, denoted the *wrap-up*, examines how many CASes completed successfully and decides whether the operation was completed or whether we should start again from the CAS generator. A particular interesting property of the CAS generator and the wrap-up methods, is that they can be restarted at any time with no harm done to the data structure.

Very loosely speaking, think, for example, of a search executed before a node is inserted into a linked list. This search would be done in a CAS generator method, which would then specify the CAS required for the insertion. For reasonable implementations, the search can be stopped at any time and restarted. Also, when the wrap-up method inspects the results of the (single) CAS execution and decides whether to start from scratch or terminate, it seems intuitive that we can stop and restart this examination at any point in time. The normalized implementation ensures that the CAS generator and the wrap-up methods can be easily restarted any time with no noticeable effects.

In contrast, the actual execution of the CASes prepared by the CAS generator is not something we can stop and restart because they have a noticeable effect on the shared data structure. Therefore, the optimistic access scheme must make sure that the CAS executor method never needs to restart, i.e., that it does not access reclaimed space. Here, again, thanks to the very structured nature of the executed CASes (given in a list), we can design the protection automatically and at a low cost.

One additional requirement of a normalized data structure is that all modifications of the structure are done in a CAS operation (and not a simple write). Efficient normalized representations exist for all lock-free data structures that we are aware of.

The formal definition of the normalized method is required for a proof of correctness. These details appear in the Appendix A.1 and also in the original paper that defined this notion, but the informal details suffice to understand the optimistic access scheme as described below.

[3] A typical example is the physical delete of nodes that were previously logically deleted in Harris-Michael linked list implementation.

2.3.3 Assumptions on the Data Structure

Here, we specify assumptions that we make about the data structure to which memory management is added. Most of these assumptions are assumed also by all previous memory reclamation schemes.

Many lock-free algorithms mark pointers by modifying a few bits of the address. The programmer that applies the optimistic access scheme should be able to clear these bits to obtain an unmarked pointer to the object. Given a pointer O, the notation *unmark(O)* denotes this unmarked pointer. This is one issue that makes our scheme not fully automatic.

Second, we assume that the data structure operations do not return a pointer to a reclaimable node in the data structure. Accessing a node can only happen by use of the data structure interface, and a node can be reclaimed by the memory manager if there is no possibility for the data structure interface functions to access it.

The data structure's functions may invoke the memory management interface. Following the (standard) interface proposed for the hazard pointers technique of [Mic04], two instructions are used: *alloc* and *retire*. Allocation returns immediately, but a *retire* request does not immediately reclaim the object. Instead, the retire request is buffered and the object is reclaimed when it is safe. Deciding where to put the (manual) *retire* instructions (by the programmer) is far from trivial. It sometimes require an algorithmic modification [Mic02] and this is the main reason why the optimistic access scheme is not an automatic transformation.

The third assumption is a proper usage of *retire*. We assume that retire is operated on a node in the data structure only after this node is unlinked from the data structure, and is no longer accessible by other threads that traverse the data structure. For example, we can properly retire a node in a linked list only after it has been disconnected from the list. We further assume that only a single thread may attempt to retire a node.

We emphasize that an object can be accessed after it has been properly retired. But it can only be accessed by a method that started before the object was retired. Nevertheless, because of this belated access, an object cannot be simply recycled after being retired.

2.3.4 A Running Example: Harris-Michael delete operation

We exemplify the optimistic access scheme throughout the paper by presenting the required modifications for the delete operation of Harris-Michael linked list. In Listing 2.1 we present the delete operation in its normalized form and including a retire instruction for a node that is removed from the list. In its basic form (and ignoring the constraints of the normalized representation), Harris-Michael delete operation consists of three steps: (1) *search*: find the node to be deleted and the node before it, (2) *logical delete*: mark the node's next pointer to logically delete it, and (3) *physical delete*: update the previous node's next pointer to skip the deleted node. During the search of the first stage, a thread also attempts to physically delete any node that is marked as logically deleted.

The normalized form of the operation is written in the three standard methods. The first method is the CAS generator method which performs the search and specifies the CAS that

will logically delete the node by marking its next pointer. If the key is not found in the linked list then a list of length zero is returned from the CAS generator. The CAS executor method (not depicted in Listing 2.1) simply executes the CAS output by the CAS generator, and thus performs the logical deletion. The wrap-up method checks how many CASes were on the CAS list and how many of them were executed to determine if we need to return to the CAS generator, or the operation is done. The wrap-up interprets an empty CAS list as an indication that the key is not in the structure and then FALSE can be returned. Otherwise, if the CAS succeeded, then a TRUE is returned. If the CAS failed, the wrap-up determines a restart.

Note that the third step of the basic algorithm, physical delete, is not executed at all in the normalized form. The reason is that in its strict structure the wrap-up method does not have access to the pointer to the previous node and so it cannot execute the physical delete. However, this is not a problem because future searches will physically delete this node from the list and the logical delete (that has already been executed) means that the key in this node is not visible to the contains operation of the linked list. Another difference between the original implementation and the normalized one is that the original implementation may simply return FALSE upon failing to find the key in the structure. The normalized implementation creates an empty CAS list that the wrap-up method properly interprets and returns FALSE.

Finally, we added a retire instruction to Listing 2.1 after any physical delete. This is proper reclamation because new operations will not be able to traverse it anymore. Using a retire after the logical deletion is not proper because it is still accessible for new list traversals.

2.4 The Mechanism

In this section we present the optimistic access mechanism, which adds lock-free memory recycling support to a given data structure implementation with a memory management interface (i.e., alloc and proper retire instructions).

The mechanism uses a single bit per thread, denoted *thread.warning*. The warning bit is used to warn a thread that a concurrent recycling had started. If a thread reads *true* of its warning bit, then its state may contain a stale value. It therefore starts the CAS generator or wrap-up methods from scratch. On the other hand, if the thread reads *false* from its warning bit, then it knows that no recycling had started, and the thread's state does not contain any stale values. We assume that a thread can read its warning bit, clear its warning bit, and also set the warning bits of all other threads (non-atomically).

Modification of the shared memory is visible by other threads, and modifying an object that has been recycled is disastrous to program semantics. Therefore, during any modification of the shared data structure we use the hazard pointers mechanism [Mic04] to mark objects that cannot be recycled. Each thread has a set of three pointers denoted $thread.HP_1$, $thread.HP_2$, $thread.HP_3$ that are used to protect all parameters of any CAS operation in the CAS generator or the wrap-up methods.

In addition, the CAS executor method and the wrap-up method can access all the objects mentioned in the CASes list that is produced by the CAS generator. The optimistic access

Listing 2.1 Harris-Michael linked list delete operation: normalized form with retire instructions

```
1  bool delete(int sKey, Node *head, *tail);
2  descList CAS_Generator(int sKey, Node *head, *tail){
3      descList ret;
4  start:
5      while(true)  {/*Attempt to delete the node*/
6          Node *prev = head, *cur = head->next, *next;
7          while(true) {            /*search for sKey position*/
8              if(cur==NULL){
9                  ret.len=0;
10                 return ret;        /*not found*/
11             }
12             next = cur->next;    /*optimistic read*/
13             cKey = cur->key;
14             if(prev->next != cur)
15                 goto start;
16             if(!is_marked(next)){
17                 if(cKey>=sKey)
18                     break;
19                 prev=cur;
20             }
21             else{
22                 if( CAS(&prev->next, cur, unmark(next)) )    /*unlink CAS*/
23                     retire(cur);
24                 else
25                     goto start;
26             }
27             cur=unmark(next);
28         }
29         if(cKey!=sKey){
30             ret.len=0;
31             return ret;    /*not found*/
32         }
33         ret.len=1;
34         ret.desc[0].address=&cur->next; /*prepared CAS*/
35         ret.desc[0].expectedval=next;
36         ret.desc[0].newval=mark(next); /*set marked bit*/
37         return ret; /*Return to CAS executor*/
38     }
39 }
40 int WRAP_UP(descList exec, int exec_res,
41                             int sKey, Node *head, *tail){
42     if(exec.len==0)    return FALSE;
43     if(exec_res==1)    /*CAS failed*/
44             return RESTART_GENERATOR;
45     else    return TRUE;
46 }
```

scheme prevents these objects from being recycled by an additional set of hazard pointers. Let C be the maximum number of CASes executed by the CAS executor method in any of the operations of the given data structure. Each thread keeps an additional set of $3 \cdot C$ pointers denoted $thread.HP_1^{owner}, \ldots, thread.HP_{3C}^{owner}$. These hazard pointers are installed in the end of the CAS generator method and are cleared in the end of the wrap-up method. A thread may read the hazard pointers of all threads but it writes only its own hazard pointers.

Modifications to the Data Structure Code In Algorithm 2.1 we present the code for reading from shared memory (the read-barrier for shared memory). This code is used in the CAS generator and wrap-up methods. (There are no reads in the CAS executor method.)

Algorithm 2.1 Read shared memory (var = *ptr)

```
1: temp = *ptr
2: if thread.warning == true then
3:     thread.warning:= false
4:     restart
5: end if
6: var = temp
```

As a read from the shared memory may return a stale value, when using the optimistic access memory recycling scheme, checking the warning bit lets the reading thread identify such an incident. If the warning bit is false, then we know that the read object was not recycled, and the read value is not stale. If, on the other hand, the warning bit is true, the read value may be arbitrary. Furthermore, pointers previously read into the thread's local variables may point to stale objects. Thus, the thread discards its local state, and restarts from a safe location: the start of the CAS generator or wrap-up method.

The code in Algorithm 2.1 resets the warning bit before restarting. This can be done because the recycler will only recycle objects that appear in the recycling candidates list when it starts. This means that the warning bit is set after the data structure issued a retire instruction on the objects in the list. Since the retire instruction is *proper* in the data structure implementation, we know that all these objects are no longer accessible from the data structure and we will not encounter any such object after we restart the method from scratch. Therefore, upon restarting, we can clear the warning bit.

In order to exemplify such a read on our running example, consider, for example, Line 12 from Listing 2.1. It should be translated into the following code (COMPILER-FENCE tells the compiler to not change the order during compilation).

Listing 2.2 Algorithm 2.1 for Listing 2.1 Line 12

```
1 next = cur->next;
2 COMPILER-FENCE;
3 if(thread->warning){
4     thread->warning=0;
5     goto start;
6 }
```

Next we define the write-barrier for all instructions that modify the shared memory. By the properties of normalized representation, this only happens with a CAS instruction. Algorithm 2.2 is applied to all modifications, except for the execution of the CASes list in the CAS executor, which are discussed in Algorithm 2.3. A simplified version of the hazard pointer mechanism [Mic04] is used to protect the objects whose address or body is accessed in this instruction. If a CAS modifies a non-pointer field then only one hazard pointer is required for the object

being modified. Recall that *unmark* stands for removing marks embedded in a pointer to get the pointer itself.

Algorithm 2.2 An observable instruction *res*=CAS($\&O.field, A_2, A_3$)

```
1:  thread.HP_1 = unmark(O)
2:  if A_2 is a pointer then thread.HP_2 = unmark(A_2)
3:  if A_3 is a pointer then thread.HP_3 = unmark(A_3)
4:  memoryFence (ensure HP are visible to all threads)
5:  if thread.warning == true then
6:      thread.warning:= false
7:      thread.HP_1=thread.HP_2=thread.HP_3 = NULL
8:      restart
9:  end if
10: res=CAS(&O.field, A_2, A_3)
11: thread.HP_1=thread.HP_2=thread.HP_3 = NULL
```

Running Example. The only case where Algorithm 2.2 is used in the running example is in Line 22 of Listing 2.1, which is translated to the following code in Listing 2.3. We stress that

Listing 2.3 Algorithm 2 for Line 22 of Listing 2.1

```
1  HP[0]=prev;//already unmarked
2  HP[1]=cur;
3  HP[2]=unmark(next);
4  __memory_fence();
5  if(thread->warning){thread->warning=0;goto start;}
6  if( CAS(&prev->next, cur, unmark(next)) ){
7      HP[0]=HP[1]=HP[2]=NULL;
8      ...
9  else{
10     HP[0]=HP[1]=HP[2]=NULL;
11     ...
```

prev and *cur* are unmarked. If, for example, *prev* was possibly marked, Line 1 would contain HP[0]=unmark(prev);.

Finally, the optimistic access scheme also protects all the objects that are accessed during the execution of the CASes list. Recall that this list is generated by the CAS generator method and executed thereafter by the CAS executor method. We need to protect all these objects so that no CAS is executed on reclaimed memory. To that end, we protect the relevant objects by hazard pointers at the end of the CAS generator method. The protection is kept until the end of the wrap-up method, because these objects are available to the wrap-up method and can be written by it. All these hazard pointers are nullified before the wrap-up method completes. The code for this protection is presented in Algorithm 2.3.

A basic optimization. A trivial optimization that we have applied in our implementation is to not let two hazard pointers point to the same object. Algorithm 2.3 guards objects until the end of the wrap-up method. So all these objects need not be guarded (again) during this interval of execution. Moreover, in case this optimization eliminates all assignment of hazard pointers in

Algorithm 2.3 End of CAS Generator.
Input: A list of ℓ CASes S= CAS(&O_1.field, $A_{1,2}$,$A_{1,3}$),...,CAS(&O_C.field, $A_{C,2}$,$A_{C,3}$).

1: **for** i $\in 1 \dots \ell$ **do**
2: $thread.HP^{owner}_{3\cdot i+1} = unmark(O_i)$
3: **if** $A_{i,2}$ is a pointer **then** $thread.HP^{owner}_{3\cdot i+2} = unmark(A_{i,2})$
4: **if** $A_{i,3}$ is a pointer **then** $thread.HP^{owner}_{3\cdot i+2} = unmark(A_{i,3})$
5: **end for**
6: memoryFence (ensure HP are visible to all threads)
7: **if** thread.warning == true **then**
8: thread.warning:= false
9: **for** i $\in 1 \dots \ell$, $j \in 1,2,3$ **do** $thread.HP^{owner}_{i,j}$=NULL
10: restart
11: **end if**
12: **return** S (finish the CAS generator method)

Algorithm 2.3 or 2.2, then the memory fence and the warning check can be elided as well.

Running Example. There are three places where the CAS generator method finishes: Line 10, Line 31, and Line 37. For Lines 10 and 31 there is no need to add any code because, in this case, the CAS generator method returns a CAS list of length zero and there is no object to protect. Thus there is no need to execute the memory fence or the warning check. For Line 37 we add the following code:

Listing 2.4 Algorithm 2.3 for Listing 2.1 Line 37

```
1  //Original code before the return (Lines 34 - 36).
2  ret.desc[0].address=&cur->next;
3  ret.desc[0].expectedval=next;
4  ret.desc[0].newval=mark(next);
5  //Algorithm 2.3 added instructions
6  HP[3]=cur;
7  HP[4]=next;
8  //No need to set HP[5] (equal to HP[4])
9  __memory_fence();
10 if(thread->warning)
11     {thread->warning=0; HP[3]=HP[4]=NULL;goto start;}
12 //End of Algorithm 2.3
13 return ret;
```

The Recycling Mechanism Having presented code modifications of data structure operations, we proceed with describing the recycling algorithm itself: Algorithms 2.4-2.6.

The recycling is done in *phases*. A phase starts when there are no objects available for allocation. During a phase, the algorithm attempts to recycle objects that were retired by the data structure until the phase started. The allocator can then use the recycled objects until exhaustion, and then a new phase starts.

The optimistic access scheme uses three shared objects pools, denoted *readyPool, retirePool*, and *processingPool*. The readyPool is a pool of ready-to-be-allocated objects from which the

allocator allocates objects for the data structure. The retirePool contains objects on which the retire instruction was invoked by the data structure. These objects are waiting to be recycled in the next phase. In the beginning of the phase, the recycler moves all objects from the retirePool to the processingPool. The processingPool is used during the recycling process to hold objects that were retired before the current phase began and can therefore be processed in the current phase. Note that while the recycling is being executed on objects in the processingPool, the data structure may add newly retired objects to the retirePool. But these objects will not be processed in the current phase. They will wait for the subsequent phase to be recycled. During the execution of recycling, each object of the processingPool is examined and the algorithm determines whether the object can be recycled or not. If it can, it is moved to the readyPool, and if not, it is moved back to the retirePool and is processed again at the next phase.

Since we are working with lock-free algorithms, we cannot wait for all threads to acknowledge a start of a phase (as is common with concurrent garbage collectors). Since no blocking is allowed and we cannot wait for acknowledgements, it is possible that a thread is delayed in the middle of executing a recycling phase r and then it wakes up while other threads are already executing a subsequent recycling phase $r' > r$. The optimistic access scheme must ensure that threads processing previous phases cannot interfere with the execution of the current phase. To achieve this protection, we let a modification of the pools only succeed if the phase number of the local thread matches the phase number of the shared pool. In fact, we only need to protect modifications of the retirePool and the processingPool. Allocations from the readyPool do not depend on the phase and also once a thread discovers that an object can be added to the readyPool, this discovery remains true even if the thread adds the object to the readyPool much later.

There are various ways to implement such phase protection, but let us specify the specific way we implemented the pools and the matching of local to global phase numbers. Each pool is implemented as a lock-free stack with a version (phase) field adjacent to the head pointer. The head pointer is modified only by a wide CAS instruction that modifies (and verifies) the version as well. When adding an object or popping an object from the pool fails due to version mismatch, a special return code VER-MISMATCH is returned. This signifies that a new phase started, and the thread should update its phase number. Each thread maintains a local variable denoted *localVer* that contains the phase that the thread thinks it is helping. The thread uses this variable whenever it adds or removes objects to or from the pool.

A phase starts by moving the content of the retirePool to the processingPool (which also empties the retirePool), and increasing the version of both pools. This operation should be executed in an atomic (or at least linearizable) manner, to prevent a race condition where an object resides in both the retirePool and the processingPool. We use a standard trick of lock-free algorithms to accomplish this without locking (in a lock-free manner). The versions (localVer and the versions of the pools) are kept even (i.e., they represent the phase number multiplied by 2) at all times except for the short times in which we want to move the items from the retirePool to the processingPool. Swapping the pools starts by incrementing the retirePool version by 1. At this point, any thread attempting to insert an object to the retirePool will fail and upon

discovering the reason for the failure, it will help swapping the pools before attempting to modify the retirePool again. Then the thread attempting to recycle copies the content of the retirePool into the processingPool while incrementing its version by 2. This can be done atomically by a single modification of the head and version. Finally, the retirePool version is incremented by 1 (to an even number), and the pool content is emptied. Again, these two operations can also be executed atomically.

When the data structure calls the retire routine, the code in Algorithm 2.4 is executed. It attempts to add the retired object to the retirePool (and it typically succeeds). If the attempt to add fails due to unequal versions (VER-MISMATCH), the thread proceeds to the next phase by calling Recycling (Algorithm 2.6), and then retries the operation.

Algorithm 2.4 Reclaim(obj)

```
1: repeat
2:     res=MM.retirePool.add(obj,localVer)
3:     if res==VER-MISMATCH then
4:         Call Recycling (Algorithm 2.6)
5:     end if
6: until res!=VER-MISMATCH
```

The allocation procedure appears in Algorithm 2.5. It attempts to pop an object from the readyPool. If unsuccessful, the thread calls Recycling (Algorithm 2.6), which attempts to recycle objects. Then it restarts the operation.

Algorithm 2.5 Allocate

```
1: repeat
2:     obj = MM.readyPool.pop()
3:     if obj==EMPTY then
4:         Call Recycling (Algorithm 2.6)
5:     end if
6: until obj!=EMPTY
7: return obj
```

The recycling procedure of the optimistic access scheme is presented in Algorithm 2.6. It starts by moving the content of the retirePool into the processingPool in a linearizable lock-free manner as described above, and it then increments the local phase counter. In Line 12, the thread checks if the (new) retirePool version matches the thread version. If not, the current phase was completed by other threads and the thread returns immediately. Note that the thread is unable to access the retirePool or the processingPool until it calls the recycling procedure again. The thread then sets the warning bits of all threads at Line 14. This tells the threads that objects for which the retire procedure was invoked before the current phase are candidates for recycling, and accessing them may return stale values. Finally, the thread collects the hazard pointer records of all threads. Objects that are pointed to by an hazard pointer are potentially modified, and should not be recycled in the current phase.

Next, the processingPool is processed. For each candidate object in the processingPool, if it

Algorithm 2.6 Recycling

```
1: //Start a new phase
2: localRetire=MM.retirePool
3: localProcessingPool=MM.processingPool
4: if localRetire.ver==localVer OR
5: localRetire.ver==localVer+1 then
6:    MM.retirePool.CAS(<localRetire, localVer>, <localRetire, localVer+1>)
7:    MM.processingPool.CAS(<localProcessingPool,localVer>,<localRetire, localVer+2>)
8:    MM.retirePool.CAS(<localRetire,localVer+1>, <Empty,localVer+2>)
9: end if
10: localVer=localVer+2
11: if MM.retirePool.ver > localVer then
12:    return    //Phase already finished
13: end if
14: for each thread T do
15:    Set T.warning = true
16: end for
17: memoryFence (ensure warning bits are visible)
18: for each HP record R do
19:    Save R in a LocalHParray.
20: end for
21: //Processing the objects
22: while res = MM.processingPool.pop(localVer) is not empty do
23:    if res!=VER-MISMATCH then
24:       if res does not exist in LocalHParray then
25:          MM.readyPool.add(n)
26:       else
27:          res=MM.retirePool.add(res, localVer)
28:       end if
29:    end if
30:    if res==VER-MISMATCH then return    //Phase already finished
31:    end if
32: end while
```

is not referenced by a hazard pointer, then it is eligible for recycling and therefore it is moved to the readyPool. Otherwise, the object cannot be recycled and it is returned to the retirePool, where it will be processed again in the next phase. Accesses to the processingPool and to the retirePool are successful only if the phase number is correct.

In order to determine if a given object is protected by a hazard pointer it is advisable to sort the hazard pointers to make the search faster or to insert the hazard pointers into a hash table (which is what we did).

Optimizations Whenever possible, we considered the following optimizations to the optimistic access scheme. First, whenever the program accessed an object that was known to be unreclaimable during the read, we omitted the warning bit check after reading it. Example for this optimizations are reading from a sentinel node, or reading from a node that is protected by

a hazard pointer.

Second, when there were two independent reads (e.g. reading the key and the next pointer of a node), we executed both reads and checked the warning bit only once after both reads completed. The correctness follows since the reads are independent and hence for each of these reads, it holds that the warning bit is checked before the read values are used.

Third, we computed the address of the warning bit once in the beginning of the operation and not repeatedly. We notified the compiler that the warning bit is unlikely to be true, so it may optimize the code for this case. Finally, we let the warning bit be of type char (which contains zero or one values) because it produced slightly faster results than the int type.

Finally, we set the warning bit by a CAS that succeeded only once per phase changed. This reduce the number of restarts to once per thread per phase; using a simple write may results in n restarts per thread per write, where n stands for the number of participating threads. We used a (standard) quad-word CAS, where the warning bit uses 8 bits, and the phase number uses the remaining 56 bits.

2.5 Methodology and Results

To evaluate the performance of the optimistic access reclamation scheme with lock-free data structures, we have implemented it with three widely used data structures: Harris-Michael linked list and hash table [Mic02], and Herlihy and Shavit's skip list [HS12]. The optimistic access memory management scheme (and additional schemes as well) were applied to the baseline algorithm in a normalized form, which performs no memory recycling. The baseline algorithm, denoted *NoRecl*, serves as a base for performance comparison. The proposed optimistic access method is denoted *OA* in the measurements.

To obtain an allocator that does not unmap pages that were previously allocated (as specified in Assumption 2.3.1), we use object pooling for allocation. The pool is implemented using a lock-free stack, where each item in the stack is an array of 126 objects. To allocate, a thread obtains an array from the stack (using the lock-free stack pop operation) and then it can allocates 126 times locally with no synchronization. To fairly compare the memory management techniques and not just the underlying allocator, we converted all implementations to use the same object pool allocation for all allocations of objects of the data structure (except for the EBP method, discussed below, which uses its own allocator). As a sanity check, we verified that the object pooling method performed similarly (or better) than malloc on all measured configurations.

Additional Memory Management Schemes compared. We discuss related memory management techniques for lock-free data structures that are available in the literature in Section 2.6. Let us now specify which methods were compared per data structure.

For Harris-Michael linked list [Mic02], a comprehensive comparison of memory management techniques was done by [BKP13]. We used their baseline implementation (*NoRecl*), their hazard pointers implementation (*HP*), and their *anchors* implementation[4]. We also compare the

[4]Loosely speaking, the *anchors* implementation installs a hazard pointer once every every K reads. We picked

Epoch Base Reclamation (*EBR*), proposed by Harris [Har01]; we took an implementation of this method by Fraser [Fra], which uses its integrated allocator. Namely, we did not replace the allocator. The latter method is not lock-free, but is sometimes used in practice to implement lock-free algorithms (to reduce the overhead associated with lock-free reclamation methods). Its disadvantage is that it does not deal well with threads failures, which is a major concerns for the lock-free methods. Finally, we implemented the optimistic access technique proposed in this paper. All implementations were coded in C.

For the hash table, each bucket was implemented as a linked list of items and the above linked-list implementations were used to support each of the buckets. For the hash table size, we used a load factor of 0.75. While both the linked list test and the hash table test use Harris-Michael linked list, the list length differed greatly for the two tests. In the linked list test all items reside on a single (relatively long) linked list, while in the hash table test the average length of a linked list is below one item. A hash table is probably a better example of a widely used data structure. We did not implement the *anchors* version of a hash table because the anchors improve accesses to paths of pointer dereferencing, while the lists in the hash table implementation are mostly of size 1, i.e, contain only a single entry.

For Herlihy and Shavit's skip list [HS12], we ported the Java implementation of [HS12] into C, and then converted it into a normalized form. We implemented the hazard pointers scheme for the skip list; the implementation uses $2 \cdot MAXLEN + 3$ hazard pointers. Finally, we implemented the optimistic access technique. The CAS generator method of the delete operation generates at most $MAXLEN + 1$ CASes to mark the next fields of the deleted node. This implies that $3 \cdot MAXLEN + 6$ hazard pointers are needed by the *OA* implementation. However, many of the protected objects are the same, and so the actual number of hazard pointers required is $MAXLEN + 5$: all CASes executed by the CAS executor method share one single modified object, and for each level the expected and new object pointers are the same. An *anchor* version for the skip list was not used because it is complex to design and no such design appears in the literature.

Methodology. It is customary to evaluate data structures by running a stressful workload that runs the data structure operations repeatedly on many threads. Similarly to Alistarh et al. [AEH+14], in all our tests, 80% of the operations were read-only. The hash table and the skip list were initialized to 10,000 nodes before the measurement began. The linked list was initialized to 5,000 nodes (thus denoted LinkedList5K). We also measure a short linked list which is initialized to 128 nodes (thus denoted LinkedList128), which creates reasonable high contention. Each micro-benchmark was executed with a varied number of threads being power-of-2 numbers between from 1 and 64 to check the behavior in different parallel settings. Each execution was measured for 1 seconds, which captures the steady-state behavior. We ensure that a 10-seconds test behave similarly to a 1-second test.

The code was compiled using the GCC compiler version 4.8.2 with the -O3 optimization flag. We ran the experiments on two platforms. The first platform featured 4 AMD Opteron(TM)

$K = 1000$ for best performance results, as thread failures are rare.

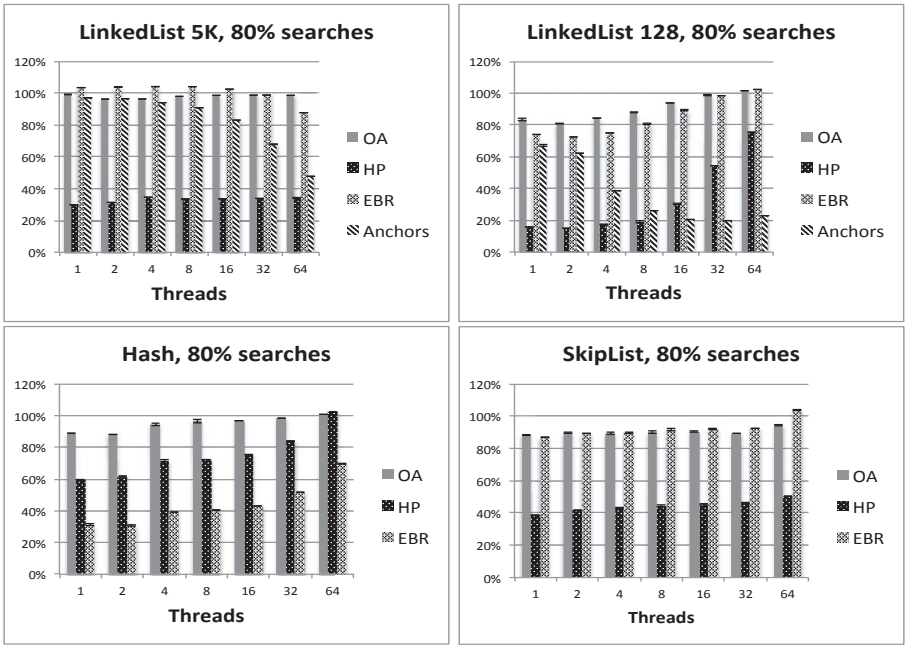

Figure 2.1: Throughput ratios for the various memory management techniques and various data structures. The *x*-axis is the number of participating threads. The *y*-axis is the ratio between the throughput of the presented scheme and the throughput of NoRecl.

6272 2.1GHz processors, each with 16 cores (64 threads overall). The second platform featured 2 Intel Xeon(R) CPU E5-2690 2.90GHz processors, each with 8 cores with each core running 2 hyper-threads (32 threads overall).

For each micro-benchmark we tested, we depict the ratio of the throughput between the evaluated memory management mechanism and the baseline algorithm (*NoRecl*), across different numbers of threads. A high number is better, meaning that the scheme has higher throughput. E.g., a result of 90% means the throughput was 0.9 of the baseline's throughput. Each test was repeated 20 times and the ratio of the average throughput is reported with error bars that represent 95% confidence level. The *x*-axis denotes the number of threads, and the *y*-axis denotes the average throughput for the evaluated method divided by the average throughput for the *NoRecl* method.

Results. In Figure 2.1 we compare the running time of the measured data structures with the various memory management methods. In this test, reclamation is triggered infrequently, once every 50,000 allocations, to capture the base overhead of the reclamation schemes. For the LinkedList5K micro-benchmark, operations have long execution time that is mostly spent on traversals. The *OA* has a very low overhead in most configurations and at max it reaches 4%. The EBR also has very low overhead, and in some cases it even ran slightly faster (recall that it uses a different allocator). However, for 64 threads its overhead was 12%. The HP overhead always exceeds 3*x*. The Anchors has an overhead of $3\% - 52\%$, and the overhead increases as the number of threads increases.

For the LinkedList128 micro-benchmark, operations are shorter, and traversals do not take

over the execution. Also, the baseline reaches maximum throughput at 16 threads, and then throughput decreases due to contention. For higher numbers of threads, memory reclamation methods behave better and even slightly improve performance of over no-reclamation by reducing contention. (This was previously reported by [DHLM13, AEH+14].) The *OA* has an overhead of $-1\% - 19\%$ and the overhead is lower for a high number of threads. The EBR has an overhead of $-2\% - 26\%$, again lower for high number of threads. The overhead of the HP method is above $3x$ for up to 16 threads. For $32 - 64$ threads it behaves better with an overhead of $25\% - 46\%$. The Anchors responds poorly to the shorter linked-list length, and incurs an overhead of $2x - 5x$. For a large number of threads (and higher contention), it became even slower than the hazard pointers method.

For the hash micro-benchmark, operations are extremely short, and modifications have a large impact on insert and delete operations. Contention has noticeable effect for 64 threads, letting executions with memory reclamation demonstrate low overheads and sometimes even slightly improved performance. The *OA* has an overhead of $-1\% - 12\%$. EBR responds poorly to the short operation execution times, and it demonstrates an overhead of $2x - 3x$ for $1 - 32$ threads. For 64 threads it slightly improves with an overhead of 30%. HP has an overhead of $16\% - 40\%$ for $1 - 32$ thread and -2% for 64 threads.

For the skip list micro-benchmark, operations take approximately the same time as LinkedList128, but face less contention. Moreover, operations are significantly more complex (executes more instructions). The *OA* has an overhead of $8\% - 12\%$. the EBR has overhead of $8\% - 13\%$ for $1 - 32$ threads, but slightly improved performance for 64 threads. The HP has overhead of $2x - 2.5x$.

To summarize, the *OA* overhead is at most 19% in all measured configurations, which is significantly faster than currently state-of-the-art lock-free reclamation methods. The optimistic access method has comparable performance to the EBR method (which is not lock-free), and is significantly better than EBR for the hash micro-benchmark.

Next, we study how the various choice of parameters affects performance. The impact of choosing the size of the local pools is depicted in Figure 2.2. Measurements for the LinkedList5K and the Hash micro benchmarks are depicted, showing the behavior with long and short operations time. All tests were executed with 32 threads. We started a new reclamation phase approximately every $16,000$ allocations. We later show that this choice is immaterial. It can be seen that the choice of local pool size has minor effect on the LinkedList5K micro-benchmark. For the hash micro-benchmark, all methods suffer a penalty for small local pools, but the *OA* scheme suffers a penalty also for a medium sized local pool. Using the *perf* Linux tool, we found that Algorithm 2.6 was the source of this slow-down. The reason is that local pools are popped from the *processingPool* and pushed into the *readyPool* in a tight loop, so contention becomes noticeable for medium sizes. Reasonably sized pools of 126 objects are sufficiently large to obtain excellent performance for *OA*.

Next, we study how the frequency of reclamation phases affects performance; the results are depicted in Figure 2.3. All tests were executed with 32 threads. The *OA* triggers a new phase when no object is available for allocation. We initialized the number of entries available for

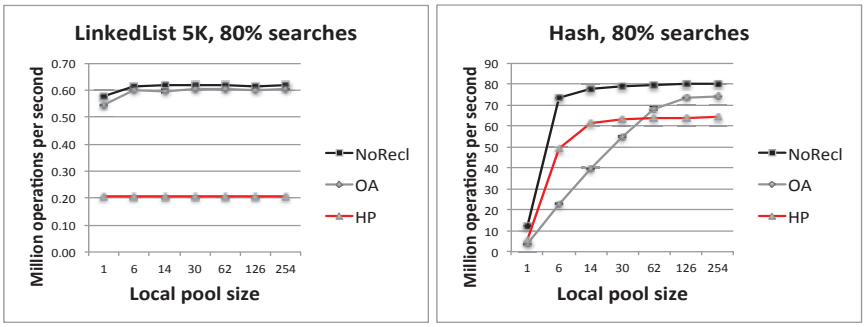

Figure 2.2: Throughput as a function of the size of the local pools.

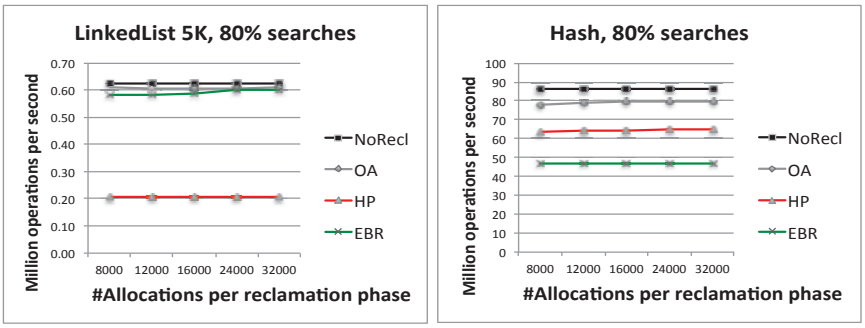

Figure 2.3: Throughput as a function of collection phases frequency.

allocation to be the data structure size plus an additional δ, for δ equals 8000, 12000, 16000, 24000, or 32000. Thus a new phase is triggered approximately every δ allocations. We started with $\delta = 8000$ because allocations are counted system-wide. For example, $\delta = 32000$ means that each thread allocated approximately $32000/32 = 1000$ objects before a phase begin. Each thread has two local pools: one for allocation and one for retired objects. The local pool size is 126. Thus $\delta = 8000 \approx 32 \cdot 126 \cdot 2$ is the minimum size where threads do not starve. With $\delta < 7600$ performance indeed drops drastically, due to thread starvation. In Figure 2.3 it is possible to see that different frequencies have a low impact on the performance of *OA*.

The other schemes do not use a global counting mechanisms. Therefore, measuring same effect for other schemes might mean a significant change in their triggering. Instead, we made an effort to compare the reclamation schemes without changing the algorithmic behavior. In the HP scheme we picked $k = \delta/32$, where each thread starts a reclamation phase *locally* after it retired k objects. In the EBR scheme, a reclamation starts *locally* after q *operations* started. We picked $q = (\delta/32) \cdot 10$ since deletions are about 10% of the total operations. We also made sure that threads do not starve in the other schemes for these settings of δ.

2.6 Additional Related Work

The basic and most popular lock-free reclamation schemes are the *hazard pointers* and *pass the buck* mechanisms of [Mic04] and [HLMM05]. In these schemes every thread has a set of *hazard*

pointers or *guards*, which mark objects the thread is accessing. Before accessing a data structure node, the object's address is saved in a thread's hazard pointer. A validation check is executed to verify that the object was not reclaimed just before it became guarded by a hazard pointer. If the validation fails, the thread must restart its operation. A node can be reclaimed only if it is not guarded by any thread's hazard pointers. The main disadvantage of these schemes is their cost. Each access (even a read) of a node requires a write (to the hazard pointer), a memory fence (to make sure that the hazard pointer value is visible to all other threads), and some additional reads for computing the validation test. Fence elision techniques [DHY10] can be used to ameliorate this overhead, but such methods foil lock-freedom.

Braginsky et al. [BKP13] proposed the *anchor* scheme as an improvement to the hazard pointer scheme. The anchor scheme allows a hazard pointer to be registered only once per each k reads of the data structure. The anchor method significantly reduces the overhead, but not to a negligible level (see measurements in Section 2.5). Moreover, the complexity of designing an anchor reclamation scheme for a given data structure is high, and the authors only provided an example implementation for the linked list.

Alistarh et al. [AEH$^+$14] have recently proposed the StackTrack method, which utilizes hardware transactional memory to solve the memory reclamation problem. This method breaks each operation to a series to transactions, such that a successfully committed transaction cannot be interfered with a memory reclamation.

Another method for object reclamation is reference counting, where each object is associated with a count of the threads that access it. Each thread increments the count before accessing an object and decrements the count afterward. This is usually done using an atomic *fetch-and-increment* instruction. A node can be reclaimed if its reference count is dropped to 0. A problem arises if an object is reclaimed immediately before its reference count is incremented. There are two variants for overcoming this difficulty. The first assumes type persistence of memory locations: once an instance of a class C is allocated at a memory location, the memory location only contains instances of class C [Val95, Sun05]. Thus, incrementing the reference count is alway valid. Still, it is necessary to check that the reference count was incremented on the correct instance, either by rereading the pointer [Val95] or requesting help when needed [Sun05]. The second variant uses an uncommon atomic primitive such as a double compare-and-swap (DCAS) [DMMSJ02] to validate the pointer and set the reference count in an atomic operation. The major drawback of these schemes is, again, their runtime overhead since they require (at least) two atomic operations per object read [HMBW07].

If lock-freedom is not required, then in most cases the epoch-based reclamation method is a high performant solution [Har01, Mic02]. Before an operation starts, the executing thread reports the timestamp it reads, and upon operation completion it clears the published timestamp. An object A can be reclaimed when every thread that started an operation before A was retired completes its operation.

Chapter 3

Automatic Memory Reclamation for Lock-Free Data Structures

3.1 Background

Several memory reclamation schemes for lock-free data structures were proposed in the literature [Mic04, HLMM05, BKP13, CP15, AEH+14], including the one presented in the last chapter. All are manual in the sense that they require the programmer to install *retire* statements that notify the memory manager each time an object is unlinked from the data structure and cannot be reached by subsequent threads. It is not easy to determine the point where the latter requirement is satisfied. Sometimes, such a point does not exist, in which case the lock-free algorithm must be modified in order to determine the point in the execution from which no thread can access an unlinked node. Harris's linked-list [Har01] and Herlihy & Shavit's skip-list [HS12], are examples of two well-known lock-free data structures that require modification to allow memory reclamation [Mic02]. Moreover, for data structures that contain several links to a node (such as the skip-list) and in the presence of concurrent inserts and deletes, it is sometimes difficult to determine a safe point at which time the node is completely unlinked. Similarly to incorrect de-allocate statements in unmanaged program code, incorrect retire statements may result in notoriously hard-to-debug memory reclamation errors. Note that it is not possible to simply use a managed language and its built-in garbage collector, as there is no garbage collection scheme available in the literature today that supports lock-free executions [Pet12].

Interestingly, the algorithmic modifications required for the insertions of the retire operations to lock-free data-structures may sometimes change the algorithms' properties and performance. For example, the linked-list of [HS12] allows wait-free searches. But modifying it to allow the retire operations [Mic02] foils this property. A detailed discussion on the linked-list and skip-list examples is provided in Section 3.7.

In this paper we propose the Automatic Optimistic Access (AOA) scheme, a memory reclamation scheme for lock-free data-structures that does not require the programmer to install retire statements. This scheme is inspired by mark-sweep garbage collection, but applied at a smaller scale – on a single data-structure with a structured algorithm, instead of over the entire

heap with a general program. Recall that we do not know how to perform full-scale lock-free garbage collection over the entire heap [Pet12]). The AOA scheme automatically determines (at run time) the set of unreachable data-structure nodes and reclaims them for future allocations. The obtained benefits are similar to the benefits of garbage collection for the heap: improve reliability, reduce debug time, and make the programmer life easier.

There has been much work on garbage collectors that obtain some partial guarantees for progress [HM92, HM01, PFPS07, PPS08, ABC$^+$08, PZM$^+$10]. However, none were able to eliminate the seemingly required synchronization barrier (e.g., handshake), when the garbage collector waits for all threads to acknowledge the beginning (or the end) of a collection cycle. A handshake foils lock-freedom because if a thread gets stuck, it does not respond to a handshake and eventually other threads run out of memory. We stress that lock-freedom requires making progress for any possible (and even a worst-case) scheduling. The AOA scheme, unlike all garbage collectors presented in the literature, strictly satisfies the lock-freedom property. In particular, it does not assume that all threads respond to a handshake.

While state-of-the-art garbage collectors can ensure the progress of the program itself, they all fail to guarantee the progress of the collector itself, causing an eventual failure of allocations and the entire program. Three major difficulties that arise in the full garbage collection setting need be dealt with here as well. First, there is the challenge of getting all threads' roots in a lock-free manner. Second, the collector must be able to function with no handshake, meaning that some threads may be executing under the assumption that a previous collection is still active. Finally, to prevent memory bloat, some form of lock-free compaction must be implemented. The last is a minor problem for us, because the nodes of the data-structure are of a fixed size. But we will deal with the first two challenges in the smaller scale of data-structure nodes under data-structure operations.

To read the roots of thread that does not respond, one needs access to its registers but modern CPUs do not allow reading other thread's registers. A naive solution is to make sure that each pointer held by a register also resides in the memory. But that requires a write and a memory fence for each register modification, making the overhead impractical. Building on the structured algorithms of lock-free data-structures, the AOA scheme solves this by making the threads record their roots at known interval and by making threads (in rare cases) return to execute from locations where their roots were known. A carefully merge of root snapshotting and execution restart allows avoiding the continuous recording of local roots. Restarting lock-free operations at various points requires some known structure of the algorithm. An adequate normalized algorithmic structure has been proposed for a different reason in [TP14]. The AOA scheme is proposed for normalized lock-free algorithms. All lock-free data-structures we are aware of can be represented in an efficient normalized representation.

Finally, since a handshake is not allowed, we must deal with program threads or collector threads that wake up and execute operations without realizing that the garbage collection has advanced to a newer collection cycle and phase. To do that, we employ standard versioning schemes to protect sensitive collector and program shared data. Careful use of versioning provides a correct scheme that reclaims data-structure nodes properly.

Our main contribution in this chapter is the proposal of the AOA scheme, a lock-free memory reclamation scheme that relieves the programmer from manually inserting retire calls. The AOA scheme is applicable to any lock-free data-structure in the normalized form of [TP14]. It eliminates the need to modify algorithms to make them amendable to insertions of retire instructions, and it eliminates the need to find adequate locations for the retire instructions themselves. Both of the above tasks are non-trivial as exemplified in Section 3.7. The AOA scheme allows a separation of concerns: it facilitates an independent algorithm design that is not memory-management-aware. Memory management can then be added automatically without any modifications to the original algorithm.

The AOA scheme deals with the main challenges of lock-free garbage collection on a small scale. We hope that the techniques and solutions proposed herein will be of use for future attempts on solving the full lock-free garbage collection problem.

3.2 Settings and Problem Statement

3.2.1 Shared Memory With TSO Execution Model

We use the standard computation model of Herlihy [Her91]. A shared memory is accessible by all threads. The threads communicate through memory access instructions on the shared memory, and a thread makes no assumptions about the status of any other thread, nor about the speed of its execution. We also assume the TSO memory model [OSS09].

3.2.2 Problem Definition

The AOA scheme automatically finds unreachable objects and reclaims them. However, unlike a full garbage collection, the AOA scheme only reclaims objects of a predetermined data-structure and it requires that the data-structure algorithm be presented in the normalized form of [TP14]. Next we specify the data-structure's nodes and links and the interplay between the data-structure, the AOA scheme, and the underlying system.

We think of a data-structure as a set of objects that are connected via data-structure links (or pointers). The data-structure objects are denoted *nodes*. Each node may have *link* fields and *data* fields; link fields contain pointers to other nodes while data fields do not contain pointers to other nodes. A pointer in the node that references an (outside) object that is not a data-structure node (i.e., that the AOA is not responsible for reclaiming) is not called a link. It is a data field. In addition, there exists a set of global pointers from outside the data-structure to nodes in the data-structure, that allow access to the data-structure. These pointers are denoted *GlobalRoots* and their location must be supplied to the AOA scheme via the interface defined in Subsection 3.2.3. We assume that the location of GlobalRoots are not modified during the execution. For example, a head pointer, pointing to the head of a linked-list is a global pointer. Note that the content of a global root can be modified or nullified, but the location of the root cannot be modified. We assume that all pointers that allow access to nodes in the data-structure are registered as GlobalRoots using the proper interface. The pointers that are used by the

data-structure operations to traverse the structure are not called global roots. They are *local* roots (local to each operating thread) and will be discussed below.

All pointers into the data-structure, including links, global roots, and local roots may contain *marked* (tainted) pointers that do not point to the beginning of a node. Specifically, many lock-free algorithms steal a bit or two of the address for other purposes, such as preventing the modification of the pointer. We assume a function that gets a marked pointer and computes the address of the node implied by it. We denote this function as *unmark*. In many implementation, using the function *unmark(P)* that simply clears the two least significant bits of *P* is adequate.

Similarly to a garbage collector, the AOA scheme reclaims all nodes that are not reachable (by a path of links) from the global and local roots. However, the AOA scheme does not consider all the pointers that are directly available to the program as local roots. The AOA scheme may reclaim an object that is being read by the program. In other words, when the program reads an object without modifying it, the pointer to this object does not protect it from being reclaimed. Thus, local pointers, that reference objects that are only read and not modified, are not considered local roots for the reachability scan, and objects reachable only from such pointers can be reclaimed. So in the AOA scheme we only consider local pointers that participate in modifications of the shared memory as roots for the traversal of live objects. The interval during which a local pointer counts as a root for the automatic memory reclamation is defined and explained in Subsection 3.5.2.

Since nodes that the thread read can be reclaimed before the thread accesses them, an effort is made to make the access of reclaimed nodes safe from hitting a trap. This is achievable by using a user level allocator and not returning pages to the operating system. Alternatively, the signal handler can be configured to ignore the trap in such a case.

For the lock-freedom guarantee we assume that enough space is allocated for the program. Otherwise, an out-of-memory exception will foil lock-freedom in a non-interesting manner. We also assume that data-structure nodes and the content of global roots are modified only via data-structure operations that are presented in the normalized form of [TP14].

An Example: the Harris-Michael Linked-List Let us exemplify the definitions above by considering the Harris-Michael linked-list [Mic02]. Consider the following snippet of code: The

Listing 3.1 Harris-Michael linked-list

```
1 struct node{
2     struct node *next;
3     int key;
4     void *data;
5 };
6 struct node *list1head, *list2head;
```

data-structure node in this case is the `struct node` structure. This struct contains a single link at offset 0 (the next pointer) and two data fields. The void* field is considered a data field and it must not point to a node. This data-structure has two global roots: {&list1head,&list2head},

which are located at fixed locations in memory. Each global root contains a pointer to a node. Note that while the nodes referenced by list1head and list2head may change, the locations of the pointers list1head and list2head are fixed throughout the execution.

3.2.3 The AOA Interface

Let us now briefly describe the interface between the AOA scheme and the data structure it serves. A complete description of the interface is presented at Section 3.8.

The data structure is responsible for providing a description of the data-structure node class to the AOA implementation. The description includes the class size, the number of pointers, and the location (offset) of each pointer. In addition, the program provides the location of all global roots to the AOA scheme. Before starting to execute data structure operations, the program initializes the AOA scheme by calling its initialization function.

Each read and write of the original algorithm operations must be modified to include a read- and write-barrier. This can be done automatically by a compiler, but in our simple implementation we let the programmer add these barriers manually. All allocations of data-structure nodes must be handled by the AOA functionality using a dedicated function.

The AOA scheme can be applied to multiple data structures in the same program. In this case, the AOA scheme interface is replicated. All components of the interface, including the allocation function, the write-barrier, etc. are implemented once for each data structure. A data-structure name prefix is used to distinguish the different interfaces.

3.3 Overview

In this section we provide an overview of the AOA scheme. The AOA scheme extends the optimistic access scheme (Chapter 2), using it to reclaim nodes. The AOA scheme is a fast lock-free reclamation scheme and our goal is to keep the overhead of reclaiming nodes low. But while the (manual) optimistic access scheme assumes *retire* instructions were installed by the programmer at adequate locations to notify a disconnect of a node from the data-structure, the AOA scheme can work without any retire statements.

The AOA scheme design is inspired by the mark-sweep garbage collector. It marks nodes that are "reachable" by the program threads and it then reclaims all nodes that were not marked. On one hand, the task here is simpler, because we are only interested in reachability of nodes of the data-structure and not the reachability of all objects in the heap. However, the reclaimer must avoid scanning the roots of the threads (as some threads may not be responding), it must make sure that threads that wake up after a long sleep do not cause harm, and it must make sure that threads that stop responding while scanning reachable nodes do not make the entire tracing procedure miss a node.

Reachability, Roots, and Concurrence. The first step in the design is an important observation. The observation is that when running the optimistic access scheme (Chapter 2), all nodes

that must not be reclaimed at any point in time are reachable from either global pointers (as defined in Section 3.2.2) or from hazard pointers used by that scheme. The AOA is built to preserve this temporal observation in the presence of concurrent execution. While the observation is correct for a snapshot of the execution, the AOA scheme works with it in a concurrent manner. First, the reclamation scheme is modified so that all active threads join the reclamation effort as soon as possible. Second, the setting of hazard pointers and the cooperation of other threads (by checking whether a reclamation has started and then moving to cooperate) is modified to ensure the correctness of the reclamation. These modifications ensure that active threads do not modify the reachability graph in a harmful way before joining the reclamation effort. Note that joining the reclamation effort does not block a thread's progress as the reclamation execution is lock-free and finite.

Protecting the Collector's Data-Structures. A classical mark-sweep garbage collector works in *cycles*, where in each cycle, one collection (mark and sweep) is executed. In this paper we use the term *phase* and not *cycle* to denote a reclamation execution in order to stress that the execution is asynchronous. Not all threads are assumed to cooperate with the currently executing reclamation and some threads may be (temporarily) executing operations that are relevant to previous phases. All garbage collectors we are aware of assume that the phases are executed in order; no thread starts a phase before all threads have finished all stages of the previous phase. This is not the case for us as we must deal with threads that do not respond.

A second challenge for the AOA is to deal with threads executing in an outdated reclamation phases and make sure that they do not corrupt the reclamation execution of the current phase. A proper execution is ensured using versioning of the reclaimers' shared data. Variables that can be corrupted by threads executing the wrong phase are put alongside a field that contains the current phase number. These variables are modified via a CAS that fails if the current phase number does not match the local phase (the phase observed by the thread that executes this modification). Before starting a new phase, all such *phase-protected* variables are updated to a new phase. Thus a late thread will never erroneously modify such phase-protected collector data.

An example of a phase-protected variable is the mark-bit table. The concern here is that a thread executing the wrong phase may mark a spurious node, corrupting the currently executing marking stage. Since the mark-bits are phase-protected, a thread executing the incorrect phase will fail to mark a spurious node.

Similarly to standard garbage collection, we are able to ensure that in each phase, all nodes that were unreachable at the beginning of the phase are reclaimed and recycled for new allocations.

Completing Work for Failing Threads. The final challenge is the proper completion of a collection phase even if a thread fails in the middle of the phase. Before discussing the solution, we start by explaining the race condition that may harm the progress guarantee of the AOA algorithm. During the mark stage, it is customary to execute the following loop: pop an item

from the mark-stack, attempt to mark this node, and if successful, scan the node's children. Suppose a thread drops dead after marking a node and before inserting its children to its mark-stack. Then the node's children must be scanned, but other threads have no way of knowing this. This race condition is traditionally handled by waiting for the thread to respond, which revokes the algorithm's progress guarantee.

To handle this challenge, we modified the mark stage as follows. First, we let the local mark-stack of each thread be readable by other threads. Second, we designed the mark procedure to satisfy the invariant that children of a marked node are either marked or are visible to other threads. Thus, even if a thread get stuck, other threads can continue to work on nodes that reside on its stack and no node is lost. This solution allows the AOA scheme to complete a garbage collection phase even when a thread crashes in the middle of marking a node.

A similar race may happen during the sweep phase, but with less drastic consequences. When a thread processing a chunk of memory for sweep purposes is delayed, we allow nodes that reside in that chunk to not be processed and not be handed to the allocator. The unprocessed chunk is a small fraction of the memory, making the harm negligible. Furthermore, these free spaces are recovered at the next phase.

3.4 Incorporating Optimistic Access Scheme Code into the AOA Scheme

The read- and write-barriers of the AOA scheme are inherited from the optimistic access scheme presented in the previous chapter. In this section we describe how the optimistic access scheme is incorporated into the AOA scheme.

In Listings 3.2, 3.3, and 3.4 we present code executed in the AOA scheme. Listing 3.2 is executed with each read from shared memory (a.k.a. a read-barrier). The code in Listing 3.3 is executed with every write to the shared memory during the execution of the parallelizable CAS generator method or the parallelizable wrap-up method. The code in Listing 3.4 is executed at the end of the CAS generator method. These algorithms are the same algorithms as in the optimistic access scheme, except for one difference that is crucial for correctness: threads that discover that a phase has changed (their warning bit is set) move to helping the reclamation procedure. As mentioned in Section 3.2.2, we sometimes need to access a pointer that was marked by the original lock-free algorithm. We use the notation UNMARK(O) for a routine that returns a pointer to the beginning of the node O, without any mark that the algorithm might have embedded in the pointer.

At the end of the wrap-up routine, all SafePointHPs are cleaned (i.e., set to NULL).

3.5 The Mechanism

In this section we present the details of the AOA scheme. We start by describing the memory layout, and then describe the three stages of the algorithm: the root collecting, the mark, and the

Listing 3.2 Reading shared memory

```
//Original instruction: var=*ptr
var=*ptr
if(thread->warning){
    thread->warning=0;
    helpCollection(); // different from the OA scheme.
    restart; // from the beginning of currently executing routine
}
```

Listing 3.3 Write shared memory

```
// Original instruction: res=CAS(O.field, A, B)
WriteHPs[0]=unmark(O);
if (A is a node pointer) WriteHPs[1]=unmark(A);
if (B is a node pointer) WriteHPs[2]=unmark(B);
__memory_fence();
if(thread->warning){
    thread->warning=0;
    helpCollection();// different from the OA scheme.
    restart; // from the beginning of currently executing routine
}
res=CAS(O.field, A, B);
WriteHPs[0]=WriteHPs[1]=WriteHPs[2]=NULL;
```

sweep. We assume that the program has the cooperation code described in Listings 3.2, 3.3, and 3.4 described above.

Recall that we use the term *phase* (and not *cycle*) to denote a reclamation execution in order to stress that the execution is asynchronous. Not all threads are assumed to be cooperating with the currently executing reclamation. Each phase is further partitioned into three *stages*: root collection, mark, and sweep. In the first stage of a phase all global pointers plus all references residing in threads' hazard pointers are collected. In the second stage of a phase we mark all nodes reachable from the set of root pointers collected in the first stage. In the third stage of a phase we sweep, i.e, reclaim all non-marked nodes and remove the marks from marked nodes, cleaning them to prepare for the mark of the next phase.

3.5.1 Memory Layout

The AOA scheme uses four globally visible variables: a phase counter, an allocation pool, a mark-bit table, an additional (small) mark-bit table for each array, and a sweep chunk index.

The phase counter is the first variable that is incremented when a new phase starts. It holds the current collection phase number and is used to announce the start of a new phase.

The allocation pool contains the nodes ready to be allocated during the current phase. It is emptied at the beginning of a phase, refilled during the sweep phase, and then used for new allocations. The allocation pool is phase-protected, meaning that adding or removing items can succeed only if the attempting thread is updated with the current phase number. The allocation pool is implemented as a lock-free stack, where the head pointer is put alongside a phase-number

Listing 3.4 End of CAS generator routine

```
1  //Assume output of the CAS generator has been set to
2  //a sequence of CASes: CAS(O_i.field_i, A_i, B_i), 0 ≤ i < k
3  SafePointHPs = all node pointers in the CAS sequence
4  __memory_fence();
5  if(thread->warning){
6      thread->warning=0;
7      helpCollection();// different from the OA scheme.
8      restart;      // from the beginning of currently executing routine (CAS generator)
9  }
10 // End of CAS generator method
```

field. Modifications of the head pointer are executed by a wide CAS, which succeeds only if the phase number is correct. Every item in the pool is an array containing 126 entries; thus a thread accesses the global pool only once per 126 allocations. Like the global pool, the local pools are also emptied at the beginning of a phase.

The mark-bit table is used for the mark stage. To avoid spuriously marking nodes by threads operating on previous phases, every 32 mark-bits are put alongside a 32-bit phase number. The marking procedure marks a node with a 64-bit CAS that sets the respective bit while checking that the phase number was not modified. If necessary, it is possible to extend the phase number to 64-bits by using a wide CAS instruction of 128 bits. However, even for frequent triggering in which a new reclamation phase starts after allocation of 1000 nodes, wraparound of the 32-bits phase number occurs only once per $2^{32} \cdot 1000 \approx 4.3 \cdot 10^{12}$ allocations.

The sweep chunk index is a phase-protected index used to synchronize the sweeping effort. It contains a 32-bit index that represents the next sweep page that should be swept. This variable is phase protected, put alongside a 32-bit phase number, and modified only via a 64-bit CAS.

Load Balancing for Tracing Arrays of Pointers

The AOA scheme employs an additional mark-bit table for each large arrays of links. (For example, for a hash table.) An array may create a problem for load balancing if not partitioned. It is preferable that each thread will trace a different part of the array and not compete with the other threads on tracing the entire array. Thus the AOA scheme divides each array into *chunks* and associates a mark bit for each chunk. These mark-bits are phase-protected; similarly to the mark-bit table, every 32 bits are put alongside a 32-bits phase number. The mark bit is set if the respective chunk was traced and all references already appear in the mark-stack. In addition, the AOA scheme associates an additional counter for each array that is used for synchronizing the tracing efforts. This counter is used to divide the chunks between threads. If a chunk is treated as a simple node, threads may continuously compete for tracing a chunk, reducing efficiency. The counter is used to optimistically divide chunks into threads such that each thread will start by tracing a different chunk. But at the end, when the counter goes beyond the array limits, threads make sure all chunks have been marked before moving on to trace the next node.

3.5.2 Root Collecting

Collecting the roots for the AOA collection phase amounts to gathering all hazard pointers and global roots. This is a huge simplification over general garbage collection for which efficient lock-free root scanning is a major obstacle. Note that we do not need thread cooperation to obtain these roots, and we can gather the roots even if threads are inactive. The code for the root collection procedure appears in Listing 3.5.

Listing 3.5 Root collecting

```
1 void collectRoots(LocalRoots){
2     For each thread T do
3         LocalRoots += T.WriteHPs
4         LocalRoots += T.SafePointHPs
5     For each root in GlobalRoots
6         LocalRoots += *root
7 }
```

Root collection is done by every thread participating in the collection phase. In fact, if a node is unreachable from the roots collected by some thread, then this node is guaranteed to be unreachable. But threads that observed previous roots may declare it as reachable (denoted floating garbage). It is possible to reach a consensus about the roots, or even to compute the intersection of the collected roots, but a simpler solution is to let every thread to collect roots on its own and to trace them.

Let us say a few words of intuition on why it is enough to trace only nodes accessible by hazard pointers (in addition to the global roots). In a sense, the hazard pointers represent local roots for the garbage collection of the data-structure. A memory reclamation phase starts with raising the warning flags of all threads. Suppose that threads could immediately spot the warning flag and could immediately respond to it. In this case all threads immediately start helping the collection and then they return to a safe point in which the hazard pointers actually represent all the local roots that they have into the data-structure. However, the threads do not respond immediately. They may perform a few instructions before detecting the warning flag. The barriers represented in Listings 3.2, 3.3, and 3.4 ensure that if a thread modifies a link during a collection phase (after the flag is set), then the modified node, the old value, and the new value are stored in hazard pointers (and thus are considered roots). Thus no harm happens during this short execution until the flag is detected.

3.5.3 The Mark Stage

In the mark phase we start from the data-structure root pointers and traverse inner-data-structure pointers to mark all reachable data-structure nodes. This procedure is executed while other threads may be executing operations on the data-structure but in a very limited way. Each thread that discovers that reclamation is running concurrently joins the reclamation effort immediately.

One of the things we care about is that if a thread fails, then other threads can cover for it, completing the work on its place. To do this, we let each thread's mark-stack be public and

also a thread keeps a public variable called *curTraced* which holds a pointer to the node it is currently working on. This allows all other threads to pick up the tracing of a failed thread.

Next, let us describe the basic tracing routing that traces one single node that currently resides on the top of the mark-stack. The reclaiming thread starts by peeking at the top of its local mark-stack, reading the first node. We stress that this node is not popped, since popping it makes it "invisible" to other threads and then they cannot help with tracing it. Second, the thread checks that the node is still unmarked and the phase is correct. If the node is already marked it is popped from the mark-stack and we are done. If the global phase was incremented, we know that we have fallen behind the reclamation execution, so tracing terminates immediately. It is fine to terminate reclamation at this point because the reclamation phase that we are in the middle of has already been completed by concurrent threads and we can attempt to continue allocating after updating to the current phase. Third (if the node is unmarked and the phase is correct), the mark procedure saves the traced node in *curTraced*. Now there is a global reference to the traced node and we pop it from the mark-stack. Next, the mark procedure traces the node, pushing the node's children to the mark-stack. Finally, the thread attempts to mark the traced node by setting the relevant bit in the mark-bit table. This mark operation succeeds if the node was unmarked and the local phase matches the global phase. If the node is successfully marked, we are done. If we fail to mark the node, then the node's children are removed from the mark-stack and we return. The code of the tracing routine appears in Listing 3.6. This procedure returns "LOCAL_FINISH" if the local mark-stack has been exhausted, and the constant 0 otherwise.

Listing 3.6 Mark a node

```
1  //Attempt to process (mark) a single node from the
2  //markstack. Return LOCAL_FINISH if local markstack
3  //is exhausted, 0 otherwise.
4  int markNode(){
5      if(markstack.is_empty())
6          return LOCAL_FINISH;
7      obj = markstack.peek();
8      if(is_marked(obj) ||
9                  phase(markbit(obj))!=localPhase){
10         markstack.pop();
11         return 0;
12     }
13     curTraced=obj;
14     pos = --markstack.pos;//popping obj.
15     for each child C of obj do
16         markstack.push(C);
17     if(mark(obj))//successfully modified mark-bit table
18         return  0;
19     else{
20         //undo pushing children.
21         markstack.pos=pos;
22         delete entries above pos (set to NULL);
23         return 0;
24     }
25 }
```

The markNode routine is invoked repeatedly until the local mark-stack is empty. But an empty (local) mark-stack of a thread does not imply the termination of the marking stage. Determining global termination in a lock-free manner is our next challenge. The tracing stage completes when all mark-stacks and the *curTraced*s of all threads contain only marked nodes. However, a thread cannot simply read all mark-stacks, because the mark-stacks can be modified during the inspection. This challenge is solved by noting a special property of the *curTraced* variable. A thread T modifies its curTraced variable when it discovers an unmarked object. If a thread inspects T's mark-stack, finds everything marked (and also T's curTraced is marked) and if and during the inspection period T's curTraced variable is not modified and T's local phase does not change, then T's mark-stack was also not modified during this period. If, however, something did change, or something was not marked, then T either finds a unmarked node and helps marking it or it simply starts the inspection from scratch.

Checking whether the marking stage completed starts by recording the current phase and the curTraced variables of all threads. Then it inspects mark-stacks of threads executing the current phase. Finally it reinspects the curTraced variable and the local phase of all threads. If no local phase was modified, no curTraced variable was modified, and all mark-stacks and all curTraced variables contained only marked nodes, then tracing is complete. If the thread observes an unmarked node and the thread is executing the current phase, it helps with tracing it. If the curTraced variables were modified and the inspecting thread observed no unmarked node, then the thread restarts the inspection. The code that checks whether the marking effort completed and the code that executes the marking stage are presented in Listing 3.7.

3.5.4 Sweep

The sweep stage is simpler than the previous stages. The sweep is done in granularity of pages, denoted *sweep pages*. Each thread synchronously grabs a sweep page and places all unmarked nodes in this page in the allocation pool. Synchronization is done by a phase-protected atomic counter, so that threads executing an incorrect phase will not grab a sweep page. A thread executing the sweep of an incorrect phase will not corrupt the allocation pool since the allocation pool is phase protected as well. A thread that drops dead at a middle of a sweep causes all unmarked nodes on this sweep page to be abandoned for the current phase. However, at most a single sweep page is lost per unresponsive thread.

Typically, sweep is also used to clear the marks from all nodes in preparation for the next collection phase. This is not done here. Instead, the mark clearing is executed at the beginning of a new phase. During a new phase initiation we need to protect all mark-bit words with the new phase number. While creating this protection we can clear the marks simultaneously with no additional cost.

The sweep algorithm is presented in Listing 3.8. It is divided into two functions. The first function, sweepPage, processes a single page, moving all unmarked nodes to the allocation pool. The second function, sweep, synchronizes the sweeping effort between all participating threads.

Listing 3.7 mark-finish-or-progress

```
bool finish_or_progress(){
    int curPhases[numThreads];
    void *curTraces[numThreads];
    for each thread T at index i do
        curPhases[i]=T.localPhase;
        curTraces[i]=T.curTraced;
        if(curPhases[i]==thread.localPhase && !is_marked(curTraces[i]))
        {
            help(curTraces[i]);
            return false;
        }
    for each thread T at index i do{
        if( curPhases[i] != thread.localPhase ) continue;
        for each markstack record R of T do{
            if(!is_marked(R)){
                help(R);
                return false;
            }
        }
    }
    for each thread T at index i do
        void *ct = T.curTraced;
        if(curTraces[i]!=ct)
            return false;//progress made by other thread.
        if(curPhases[i]!=T.localPhase)
            return false;//thread i joined the mark.
    return true;//finished.
}
void help(void *node){
    if(thread.localPhase == phase)
        markstack.push(node);
    else
        markstack.clear(); //finish mark stage.
}
void markStage(){
    do{
        while(markNode()!=LOCAL_FINISH){}
    }while(finish_or_progress()==false);
}
```

3.5.5 Phase Triggering

Similarly to triggering of general garbage collection, triggering a reclamation is more of an art than science. Various triggering mechanisms can be used. In our simple implementation we used a very simple heuristic. Before the measurement began we allocated a fixed number of nodes and inserted them to the allocation pool. This can be thought of as analogue to a memory manager that allocates a heap for its use in an execution. A new phase was triggered when the allocation pool was exhausted. During the collection phase all unreachable nodes were returned back to the allocation pool.

This implementation ensures that everything is lock-free. If one uses an underlying allocator to extend the allocation pool dynamically, then one should make sure that the underlying allocator is lock-free and does not allow unmapping of previously allocated pages (note that

Listing 3.8 Sweep

```
sweepPage(int pageNumber){
    void *page = getPage(pageNumber);
    for each node O residing in page do{
        if(mark-bit(O)==false)
            localPool.insert(O);
        if(localPool.size ==LOCAL_POOL_SIZE){
            allocationPool.insert(localPool, localPhase);
            localPool.reset();
        }
    }
}
sweep(){
    while(true){//while sweep not finished.
        do{//obtain a single page for sweep
            int64 old = sweep_chunk_index;
            if(index(old)>=numSweepPages)
                return;//sweep finished
            if(phase(old)!=localPhase)
                return;//entire phase finished
            int64 new=old+1;//same phase, index+1
        }while(!CAS(&sweep_chunk_index, old, new));
        //Sweep nodes on obtained  page.
        sweepPage(index(old));
    }
}
```

unmapping pages generally foils lock-freedom), then it is possible to consider other triggering schemes that adapt to the runtime behavior of the data-structure. In Listing 3.9 we present the code for allocating a node and the code for triggering a new collection phase.

3.6 Methodology and Results

We used the AOA reclamation mechanism with two widely used data-structures: Harris-Michael linked-list and a hash table. We compared the execution of these data-structures with the AOA mechanism to executions of these data-structures with the basic manual optimistic access mechanism presented in Chapter 2. As we shown, this scheme represents the most efficient manual reclamation available today. We also compared to executions with no reclamation and to executions with a reference counting mechanism. Although not stated by its author, the reference counting method of Valois [Val95], enhanced by Michael and Scott [MS95], can be applied automatically by the compiler if there is no cyclic garbage. Since cyclic garbage is not common in many lock-free data-structures, reference counting can be considered an alternative to the AOA method proposed in this paper. It offers comparable guarantees.

On commodity hardware, reading the node reference and incrementing the reference count is not atomic. If the node is reclaimed after reading the reference and before incrementing its counter, the counter of the non-existing node is modified, possibly corrupting memory. Thus, [Val95] assumed type-persistency: once a node is allocated at a memory location, the memory location will occupy only instances of the same class type as the first occupying node. Thus a

Listing 3.9 allocation and phase triggering

```
void *alloc_DS(threadData t){
    if(t.localPool is empty){
        t.localPool = allocationPool.pop(t.localPhase);
        if(t.localPool is empty)
            TriggerNewPhase(t);
    }
    return t.localPool.pop();
}

void TriggerNewPhase(threadData t){
    CAS(&phase, t.localPhase, t.localPhase+1);
    t.localPhase = phase;
    //Initialize phase-protected variables
    For each phase-protected variable V do
        tmp = V
        while(tmp.version<t.localPhase){
            CAS(&V, tmp, <0,t.localPhase>);
            tmp = V
        }
    Set warning flag for all threads.
    collectRoots(t.markstack);
    markStage();
    sweep();
}
```

belated thread modifies only a valid reference count field. The reference counting scheme is denoted RC. The manual optimistic access scheme is denoted MOA and the baseline algorithm that performs no memory reclamation is denoted NR.

Methodology. Following the tradition in previous work, we evaluated the data-structure performance by running a stressful workload that executes the data-structure operations repeatedly on many threads. In all our tests, 80% of the operations were read-only. For the linked-list data-structure we tested two configurations: one where the list length was 5000 denoted LinkedList5000, and another with list length 128 denoted LinkedList128. The hash table size was 10000, and the load factor was 0.75. These micro-benchmarks cover a wide range of behaviors. The LinkedList5000 has low contention as each operation is relatively long. The LinkedList128 has higher contention as the operations are of medium length. The hash has low contention, but extremely fast operations. Similar settings were used in previous work (e.g., [AEH+14, CP15]).

To check the scalability of the proposed scheme, each micro-benchmark was executed with a varying number of threads, all of which were power-of-2's ranging between 1 and 64. Each execution lasted 1 second, and the total number of executed operations was recorded (throughput). Longer execution times (e.g., 10 seconds) produce similar results. Each execution was repeated 10 times, and the average throughput and 95% confidence levels are reported. For the AOA scheme we initialized the allocation pool with 32,000 nodes before the measurements began; a new collection phase is triggered when the allocation pool is exhausted.

The code was compiled using the GCC compiler version 4.8.2 with the -O3 optimization

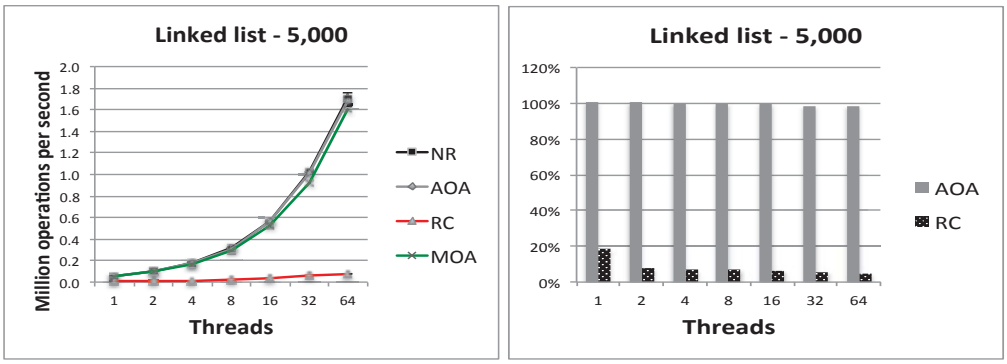

Figure 3.1: Comparing the throughput of the proposed AOA scheme, reference counting, and no reclamation with Harris-Michael linked-list with 5,000 items. The x-axis is the number of participating threads. The y-axis is (a) the throughput of the presented scheme or (b) the throughput ratio between the presented scheme and NoRecl.

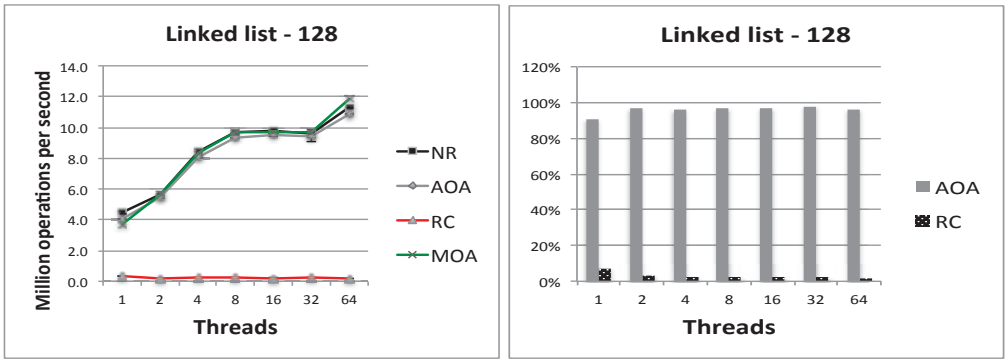

Figure 3.2: Comparing the throughput of the proposed AOA scheme, reference counting, and no reclamation, with Harris-Michael linked-list with 128 items. The x-axis is the number of participating threads. The y-axis is (a) the throughput of the presented scheme or (b) the throughput ratio between the presented scheme and NoRecl.

flag, running on an Ubunto 14.04 (kernel version 3.16.0) OS. The machine featured 4 AMD Opteron(TM) 6376 2.3GHz processors, each with 16 cores (64 threads overall). The machine used 128GB RAM, an L1 cache of 16KB per core, an L2 cache of 2MB for every two cores, and an L3 cache of 6MB per processor.

Results. In Figure 3.1 we compare the performance of the proposed AOA scheme, the manual optimistic access scheme (MOA), the reference counting scheme, and no reclamation, with Harris-Michael linked-list of length 5,000 nodes. It can be seen that the overhead of the AOA scheme, compared to no reclamation, is very low, and at max reaches 3%. Compared to the reference counting scheme, the AOA scheme improves the performance by $3x - 31x$. The reason is that reading memory is very lightweight in the AOA scheme, whereas the reference counting scheme requires two atomic operations per read. The overhead of the reference counting scheme is much higher as the number of threads grows, due to the contention on the reference counting field. The AOA scheme performed similar or slightly better than the manual optimistic access scheme.

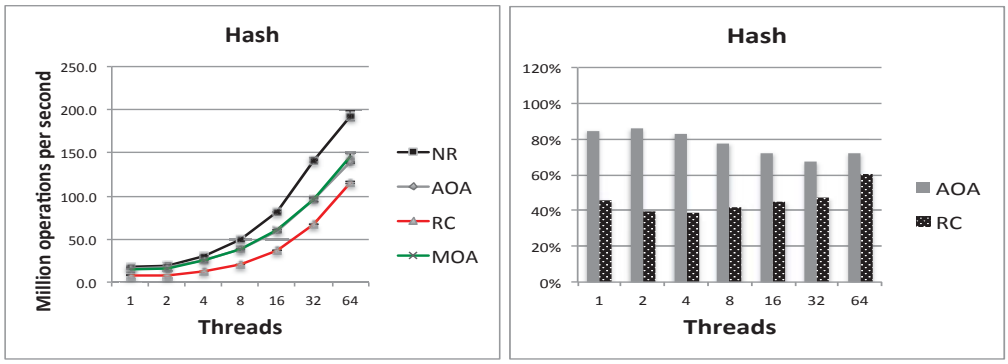

Figure 3.3: Comparing the throughput of the proposed AOA scheme, reference counting, and no reclamation, with the hash table. The x-axis is the number of participating threads. The y-axis is (a) the throughput of the presented scheme or (b) the throughput ratio between the presented scheme and NoRecl.

In Figure 3.2 we compare the performance of the proposed AOA scheme, reference counting, manual optimistic access reclamation, and no reclamation, when used with the Harris-Michael linked-list of length 128 nodes. The AOA scheme incurs an overhead of $0-10\%$ compared to the no reclamation scheme. Compared to the reference counting scheme, the AOA scheme improves performance by $6x-25x$. The AOA scheme performed similar to the manual optimistic access scheme: comparing the AOA scheme with the manual optimistic access scheme shows that it was better by 9% for a single thread and slower by 9% for 64 threads. For a single thread the additional retire calls had some overhead, while for 64 threads these local computations probably reduced contention and improved scalability.

In Figure 3.3 we compare the performance of the proposed AOA scheme, reference counting, manual optimistic access reclamation, and no reclamation, when used with a hash table of size 10,000 nodes. The overhead of the AOA scheme, compared to the no reclamation, is $14\%-33\%$, which slightly increases as the number of threads increases. Compared to the reference counting scheme, the AOA scheme improves performance by $30\%-54\%$ for $1-32$ threads, and by 16% for 64 threads. With the hash micro-benchmark, the fraction of CAS instructions (vs. read instructions) is greater than in previous micro-benchmarks, so the reference counting cost of 2 atomic instructions per read is somewhat reduced. Furthermore, the read operations do not contend since each thread picks a random bucket, reducing the reference counting overhead in this case. The AOA scheme performed almost exactly like the manual optimistic access scheme. In fact it is difficult to see the gray line as it is almost completely hidden by the green one.

In all of the measurements so far, we used one data-structure implementation with different memory reclamation schemes. But some algorithms cannot be used with manual reclamation, yet, they can be used with automatic reclamation (as discussed in Section 3.7). So we also compared the execution of the Harris-Michael linked-list which is amendable to manual memory reclamation to executions of the linked-list with wait-free searches of [HS12]. The latter cannot be used with manual reclamation. Each of these algorithms was also run with no memory reclamation for comparison. For the Herlihy-Shavit algorithm we considered NoRecl (HS-NR)

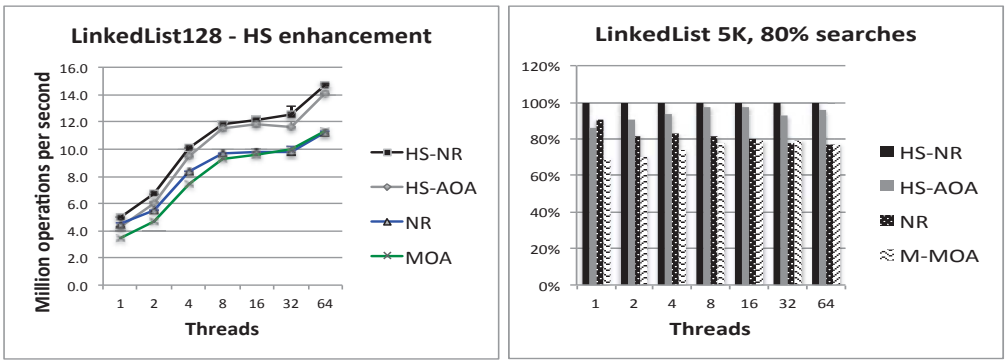

Figure 3.4: Studying the benefit of applying Herlihy-Shavit enhancement. NR and MOA stands for the baseline implementation with no reclamation (NR) and the manual optimistic access scheme. HS-NR and HS-AOA are the implementation enhanced by Herlihy-Shavit wait free searches with no reclamation and the AOA scheme. This enhancement does not satisfy the requirements needed to apply a manual memory reclamation scheme. Showing actual throughput (a) and throughput ratio over HS-NR (b).

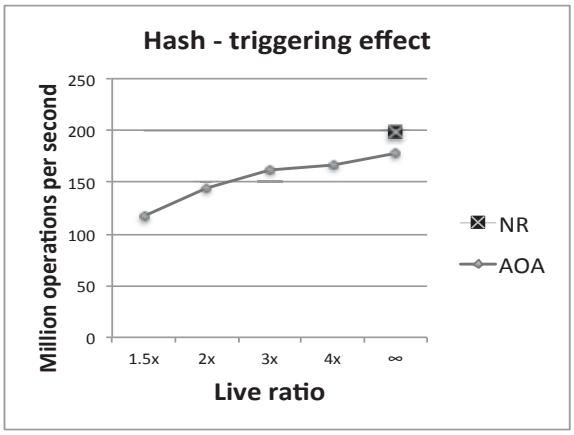

Figure 3.5: Studying the effect of triggering on performance. The x-axis is the number of nodes in the allocation pool divided by the number of live nodes. The y-axis is the throughput.

and AOA (HS-AOA). And for the Harris-Michael linked-list we used NoRecl (NR) and the manual optimistic access scheme (MOA). The results are depicted in Figure 3.4. The Herlihy-Shavit algorithm that cannot be used with manual reclamation performed better than the Harris-Michael algorithm, to which manual reclamation can be added. With no reclamation the difference is 9%-23%. Furthermore, the faster algorithm with the AOA scheme outperforms the slower algorithm even when the slower algorithm does not spend any time on memory reclamation.

In Figure 3.5, we study the effect of triggering on performance under infrequent recycling. The frequency of recycling is determined by the space reserved for allocation, which determines how much allocation can take place before recycling is needed. Similarly to the garbage collection literature, we set the available space for allocation as a multiplicative factor of the live space (nodes used by the data-structure). For this benchmark, we let the live space be 30,000 nodes and we vary the live ratio in the range $1.5x, 2x, 3x, 4x$ and ∞. Note that when we use ∞,

we don't reclaim at all and so the overhead originates only from the read- and write-barriers. We compared these runs to runs with no reclamation (for which triggering is not relevant). In Figure 3.5 it can be seen that the AOA scheme is sensitive to the frequency of triggering, and performance improves if the triggering is infrequent. Looking at ∞ live ratio (no reclamation), we see that the overhead that the AOA scheme incurs on the execution without reclamation is 11%. A ratio of 2 compared to a ratio of ∞ demonstrates that the overhead of the reclamation itself in this case is 19%.

For the AOA scheme the memory overhead is a constant which our parameters determine to be around $32,000$ objects. For the NoRecl method the space overhead is of-course the largest in the long run. It is simply determined by the number of allocations because there is no reclamation. For the data structures at hand, allocations only happen during insertions, when the key to be inserted is not found in the data-structure. The number of such allocations is about 5% of the overall number of operations executed, because in our workload 10% of the operations are insertions and for approximately half the key is not found in the data structure.

The RC method exhibits a small memory overhead in most of the cases as it reclaims space with no delay. However, for the hash table with 64 threads the overhead was approximately 20,000 nodes. The reason is that in the implementation we used (of [MS95]), if the thread's local list of reclaimed nodes is empty, it immediately allocates a new node, even though other lists are full. While it is possible to improve the memory overhead for the RC scheme, we did not investigate this avenue further. The MOA memory overhead was similar to the AOA scheme almost everywhere except when running the hash benchmark with 64 threads. There, the reclamation overhead of the MOA scheme was high when the memory overhead was set to 32,000 nodes. The reason for this reduced performance for the manual scheme is that this scheme allocates a local pool for retired objects to each thread and also a local pool for local allocations. This reduces the number of objects that are available for allocation (for all threads) and results in excessive reclamation cycles. In addition, during a reclamation cycle, there is a high contention on the shared pool of retired objects. Thus, for this case we let the memory overhead be approximately 50,000 nodes.

We also recorded the number of memory reclamation phases performed by the AOA scheme. For the hash micro-benchmark, the number of phases varies between 32 for a single thread and 375.8 for 64 threads. For the LL5K micro-benchmark, the number of phases varies between 0 and 3. This is a long list and the number of operations that execute in one second with a single thread did not require memory reclamation. We did verify (for all benchmarks) that executions of 10 seconds show similar throughput behavior with 10 times the number of collections. For the LL128 micro-benchmark, the number of phases varies between 5 and 18.

To summarize the performance evaluation, the AOA scheme outperforms the reference count scheme on almost all configurations, and many times in a drastic manner. For moderate workloads, it demonstrates a very low overhead over the baseline algorithm that reclaims no nodes. The AOA performs similarly to the manual optimistic access scheme but the automatic scheme is easier to apply and it can be applied in cases where the manual reclamation methods cannot be applied.

3.7 Motivating Examples

In this section we demonstrate the difficulty of applying manual memory management with *retire* instructions in order to motivate automatic memory management. We look at two basic and well-known lock-free data-structure algorithms: Harris's linked-list [Har01] and Herlihy and Shavits' skip-list [HS12]. For both data-structures we show that non-trivial algorithmic modifications are required in order to place retire instructions and apply a manual memory reclamation method. We then discuss possible solutions for making these algorithms amendable to manual memory reclamation.

3.7.1 Harris's Linked-List [Har01]

Harris's linked-list [Har01] was innovative when introduced and is still considered one of the most simple and efficient lock-free data-structures. Harris also proposed the Epoch-Base Reclamation (EBR) scheme for reclaiming nodes in the list, but using Epoch-Base reclamation foils lock-freedom as it is not robust to thread failures.

Michael [Mic02] discusses the difficulty of node reclamation for Harris' list and argues that it is not possible to apply the hazard pointers method to this data-structure implementation. The reason is that in order to apply hazard pointers or a similar reclamation method a retire operation must satisfy a condition that cannot be satisfies by Harris' list. The condition is that one issues a retire operation on node N only after N has been disconnected from the list and no thread will obtain access to it from now on. It is possible that a thread is looking at N simultaneously with the execution of the retire instruction and this case is fine because hazard pointers will be used to delay the actual reclamation of N. But no thread should be able to gain access to N after the retire instruction was issued. This is important because N may be now reclaimed and future accesses may view stale values.

Let us review why the condition is not satisfied in Harris' algorithm and then present Michael's fix. Note that this understanding and modification requires a subtle (i.e., deep) understanding of the algorithm and of the memory reclamation scheme.

Deleting a node N in Harris's linked-list happens in two stages. First, N's next pointer is marked (to prevent further modifications). Then, the node is physically deleted by rerouting the next pointer of N's preceding node to point to the node that follows N in the list. Concurrence may cause several subsequent nodes to be marked before physical deletion happens to any of them. Handling a series of consecutive marked nodes is a major issue for the installation of retire operations. Let us explain why. In Harris's implementation, if two (or more) consecutive nodes are marked, then they are simultaneously deleted physically by rerouting the next pointer of the node that precedes all these consecutive marked nodes to point to the node that follows the series of marked nodes. This seems a natural way to deal with a series of marked nodes and it is also a very efficient solution. But the simultaneous physical deletion makes the installation of hazard pointers fail because it foils the condition needed to apply the hazard pointers scheme. Suppose a thread is currently accessing one of the nodes in the sequence of marked nodes. Then this thread is able to access other nodes in this sequence even after they are physically deleted

from the list. This is bad news for the installation of retire instructions. This thread may access a node that has been retired.

Let us describe specific race condition. Suppose that A,B,C,D are consecutive nodes in the list $A \rightarrow B \rightarrow C \rightarrow D$. T_1 starts a search, and reaches B. T_2 marks C, and then T_3 marks B. Next, T_4 notices that B and C are marked and physically deletes B and C by connecting $A \rightarrow D$. At this point B and C are retired. Then, both T_2 and T_3 discovered that B and C were unlinked and restart the operation, cleaning their own hazard pointers to the nodes B and C. Since C was retired and no hazard pointer references it, it is next reclaimed. Now T_1 wakes up, reads B's next field (C), protects it with a hazard pointer, revalidates that B still points to C, and then accesses C in spite of the fact that C has been reclaimed and may contain stale values for T_1.

This race condition is rather tricky and non-expert developers may attempt to apply hazard pointers to this scheme. The resulting bug reveals itself only under rare conditions, which complicates debugging.

[HS12] proposed a variant of the Harris-Michael algorithm where the *search* (a.k.a. *contains*) operations do no participate in physical deletion. This yields faster and wait-free searches. But their variant also does not enable the installation of hazard pointers due to similar races.

Michael's solution to this problem is to physically delete one node at the time, starting at the first one. In addition, Michael makes sure that a thread does not traverse through a marked pointer without attempting to handle the physical deletion and starting from scratch if the delete fails. Michael's solution allows memory reclamation, and prior to this work there was no way around Michael's condition for inserting retire statements. But as shown in our measurements, Michael's solution also has a performance cost due to threads starting searches from scratch.

3.7.2 Herlihy and Shavits' [HS12] Skip-List

Skip-list is a widely used data-structure as it allows a simple implementation that (probabilistically) achieves a logarithmic execution time for the insert, the delete and the search operations. The lock-free skip-list algorithm of Herlihy and Shavit is much simpler than the competing lock-free tree algorithms.

The skip-list is composed of a sequence of sorted linked-lists. In the lock-free implementation [HS12] these lists are implemented according to the Harris-Michael linked-list [Mic02]. The lowest linked-list contains all the elements in the skip-list. Higher linked-lists contain only a subset of the elements, where on average, the size of the i^{th} linked-list is half the size of the $i-1^{\text{th}}$ linked-list. Each element of the skip-list has a height, and an element of height ℓ participates in the first ℓ linked-lists. Element insertion begins with the thread inserting the element to the bottom level and then proceeding with the insertions to the following levels, one level at a time. Deleting an element works the other way, starting from the linked-list of the element's height and finishing at the bottom level. As in Harris-Michael linked-list, the element is first logically deleted, and then (after all levels are logically deleted) it is disconnected from the list.

This implementation does not satisfy the conditions needed to apply the hazard pointers (or

any other similar) manual memory reclamation scheme. Consider an element with a height 2, but any height higher than 1 will do. The main difficulty here is that such an element is inserted *twice* to the data-structure. Once into the bottom level and then into the second level. If a thread is delayed before installing the last reference to this node, then it is unsafe to retire this node, because a reference to this node might be later installed into the skip-list. If one attempts to retire such a node, than there is a scenario in which a retired node is reached by a thread after it was reclaimed.

Let us describe a specific race condition. Suppose a thread gets delayed in the middle of inserting an element E of value X and height 2 into the data-structure. The thread has already inserted E to the bottom level but gets delayed before linking it to the second level. At this time, another thread starts a delete operation of value X, finds E, deletes it from the skip-list, then unlinks it. At this point it is natural to put a retire statement, with which the deleting thread notifies the reclamation system about E, and the reclamation system may in turn recycle the node and allocate it again as E'. Next, the inserting thread wakes up and installs the reference to E, which actually references E', into the skip-list. This corrupts the data-structure. Such a corruption is very difficult to debug.

Let us now describe a possible modification of the algorithm that enables the installation of retire statements, and so also the application of the hazard pointers [Mic04] or the optimistic access [CP15] memory schemes. The inserting thread prevents recycling of the currently inserted item using some methodology (a hazard pointer or some other coordination method) until the insertion operation is complete. When it is complete, the inserting thread checks whether the node was concurrently deleted. If it was, the inserting thread unlinks it from the data-structure. Thus, when the node becomes reclaimable, it is guaranteed not to be reachable from the data-structure. This solution is delicate and requires expertise and care about possible races. The AOA reclamation scheme is automatic and does not require such modifications.

3.8 The AOA Interface

Let us now describe the interface between the AOA scheme and the data-structure it serves. The AOA scheme is implemented as a header file in C. The rest of this chapter explains how to implement the functionalities included in this header file. But before including the header file in the data-structure's source code, several #define statements must be provided by the programmer. These #define statements are discussed below.

The programmer starts by providing a name space for the data-structure to be used by the AOA scheme for this data-structure and to avoid collision of AOA actions on different data-structures. This name space will be used as a suffix for all the AOA function names. The name space (denoted DS) is provided using a #define statement. The second definition provided by the programmer is the NODE_SIZE parameter that contains the size of the data-structure node. The third define statement specifies NCHILDREN that contains the number of link fields in the data-structure node. In our simple implementation NCHILDREN must be between 1 and 3, but it is easy to support any constant number of links. The fourth set of (NCHILDREN) definitions

is CHILD_OFFSET_{1,2,3}. The definition CHILD_OFFSET_i specifies the offset of the i^{th} link. The fifth (optional) definition specifies a function UNMARK(ptr) that gets a pointer and returns an unmarked pointer. If undefined, the implementation used the default of clearing the two least significant bits.

Each read and write of the original algorithm operations must be modified in a way that is named read- and write-barrier in the memory management nomenclature. This can be done automatically by a compiler, but in our simple implementation we let the programmer apply these additions. All allocations of data-structure nodes must be handled by the AOA functionality and this should be done using the alloc_DS function (where DS is the name space of the said data-structure). The alloc_DS function returns a newly allocated node.

An initiation function denoted init_allocator_DS must be called during the initiation of the data-structure to initiate the memory manager. The destroy_DS function may be invoked to de-allocate the AOA internal data-structure when it is no longer needed. The function addGlobalRoot_DS is used to declare the location of global roots. It is possible to declare an array of roots by calling this function once and providing the length of the array. (That may be useful for a hash table.)

Harris-Michael Linked-List Example Let us now exemplify the user of the AOA interface on the Harris-Michael linked-list. The example is provided in Listing 3.10.

Listing 3.10 Harris-Michael linked-list

```
1   #include "hmll.h" //the data-structure definitions
2   --------------- AOA memory manager
3   #define DS HMLL //Harris-Michael linked-list
4   #define NODE_SIZE sizeof(struct node)
5   #define NCHILDREN 1
6   #define CHILD_OFFSET_1 offsetof(struct node, next)
7   //default (unneeded) definition
8   //#define UNMARK(ptr) ((void*)(((long)(ptr))&(~3ul)))
9   #include "aoa.h" //  supplied by the AOA implementation
10  --------------- Implementation
11  void insert(...){
12      ...
13      struct node *newnode=alloc_HMLL(threadData[tid]);
14  }
15  void init(){
16      //initialize HMLL memory manager
17      init_allocator_HMLL(threadData, numThreads);
18      //add global roots
19      addGlobalRoot_HMLL(&list1head, 1);
20      addGlobalRoot_HMLL(&list2head, 1);
21      ...
```

Applicability to Multiple Data-Structures It is possible to apply the AOA scheme to multiple instances of a data-structure by providing the global roots of all instances via the AOA interface. It is also possible to apply the AOA scheme to multiple data-structures. In this case, each

data-structure must be compiled separately (on a different source file) and each data-structure must provide a unique name space. In our implementation there is no sharing between multiple instances of the AOA scheme, even though such sharing may be possible (e.g. using a single set of hazard pointers). Of course, the AOA scheme does not prohibit using different memory reclamation schemes on other data-structures. But if the data-structure uses the AOA interface, it is incorrect to allocate an instance without registering it properly.

Chapter 4

Data Structure Aware Garbage Collection

4.1 Background

Garbage collection is a widely accepted method for reducing the development costs of applications. It is used in many of today's programming languages, such as Java and C#. However, garbage collection does not come without a cost, and automatic memory management adds a noticeable overhead on application performance.

Much effort has been invested in improving garbage collection efficiency, see for example [BBYG+05, KP06b, LP99, DKP00, SBYM13, YBF+11, BM08, BM03, HMGJ04, CWZ98, HFB05, SMB04]. Most modern collectors employ a tracing procedure that discovers the set of objects reachable from a set of root objects. The tracing procedure is considered the most time consuming task, so most performance improvement efforts focus on it.

Knowledge about program behavior may greatly benefit tracing. Yet, today's programming languages do not provide interfaces to the application to pass on this information. Creating a good interface and garbage collector support for it is a challenge. First, we do not want to risk correctness even when the programmer provides bad hints. Recall that garbage collection was introduced to eliminate common bugs originating from programmer misunderstanding of program overall behavior. Second, a badly designed interface may require high programmer effort for producing relevant hints, and the programmer may not be willing to or not be capable of spending the time to collect such information. Finally, a naive interface design may end up not yielding improvements in garbage collection performance, and in this case there would be low incentive for implementing such an interface in the compiler or for using it even when implemented.

In this paper we propose the DSA (data structure aware) interface: an interface between the program and the memory manager that avoids the above pitfalls. First, the DSA interface allows the programmer to express the program's behavior, but in a way that requires very limited programming effort. Second, the information that is exposed to the garbage collector through the DSA interface does improve its efficiency and scalability, and also the overall program

running times. And most important, a specification of incorrect program behavior through the DSA interface will never lead to program failure or inappropriate pointer handling, but only to decreased performance.

The DSA interface concentrates on data-management applications that are centered around data structures. A database server is a typical example: usually there exists a single data structure that represents a table (e.g., a balanced tree) and the database data is stored in these tables. Although there might exist several tables in the database, each table is typically an instance of the same tree. An example benchmark in this category is HSQLDB of the DaCapo benchmark suite 2006. Cache applications such as KittyCache also fall into this category. A cache application reduces accesses to a resource (e.g., the network or a database) by caching frequently accessed data. A typical cache implementation places all items in a single (large) hash table. An additional example of applications that can be improved by the proposed interface is the set of applications that deal with item management, e.g., a management application for a wholesale company, such as SPECjbb2005. Such an application tracks the number of items in each warehouse, the registered customers, and the orders being processed. A typical implementation would place items of the same kind in a data structure (a hash table or a tree). While the application may use several instances of such data structure, they typically all share the same main node class.

Applications of the above categories share two useful properties from a memory manager point of view. First, a large fraction of the objects are accessible only via a dominant data structure. Second, removing an item from the data structure is a well defined operation; the programmer actively selects an item and removes it. While it is not correct to assume that a removed item is unreachable and can be reclaimed, it *is* safe to assume that an object is *not* reclaimable before it is removed from a (reachable) data structure. This is the property that we exploit in the proposed interface.

On the other end of the DSA interface there is a DSA memory manager that benefits from the information passed by the program through the DSA interface. The DSA memory manager that we present allocates nodes of the data structure separately from the rest of the heap. This provides additional locality for the data structure. The DSA memory manager assumes that nodes in the data structure are alive unless they were declared as deleted. It uses this knowledge to improve garbage collection efficiency. Furthermore, placing all data structures together may improve locality of reference beyond just reducing the time spent on garbage collection. In fact, improved performance (beyond reduced collection times) is visible in all the programs we evaluated.

The developers may fail to report the removal of a node from the data structure. Such missing information about node removal is taken care of by an additional mechanism of the DSA that can reclaim such nodes. On the other hand, wrong hints provided by the program, i.e., notifying the collector that objects are deleted from the data structures while they are not, will never cause the garbage collector to reclaim a reachable object. However, it may cause the garbage collection to perform additional work and thus hurt performance, and temporarily increase the space overhead.

The DSA interface is not beneficial to all programs, as some may not use a major data

structure to hold a significant fraction of its data. It is therefore important to ensure that a JVM using the interface does not claim a noticeable overhead of applications that do not use the interface. The design we propose only requires an overhead of one conditional statement in the class loader and a second conditional statement in the beginning of a GC cycle; evaluation shows (as expected) that the overhead in this case is negligible.

A typical use of data structures makes the application of the DSA interface even easier. Many programs use library implementations of data structures to organize their data. If the program chooses to use a library implementation that employs the DSA interface, then the developer needs to do very little to gain the DSA benefits. The library implementation will declare the relevant class as an appropriate data structure and will issue remove notifications to the interface. The one thing that the developer should take care of is to notify the interface when the entire data structure becomes unreachable, as discussed in Subsection 4.6.1.

We implemented the DSA memory manager in the JikesRVM [AAB+00] environment. We also modified several common data structures from the *java.util* package to support the DSA interface. We then evaluated the DSA memory manager on the KittyCache program [Kit09], the pseudo SPECjbb2005 benchmark [SPE05], the HSQLDB benchmark from the DaCapo suite 2006, and the JikesRVM itself.

For KittyCache [Kit09], a garbage-collection-aware version required modifying 8 lines of the program code. These changes yielded a 40-45% improvement in GC time and a 4-20% improvement in overall running time (depending on the heap size and GC load). For the pseudo SPECjbb2005 benchmark, a garbage-collection-aware version required modifying four lines of code, and yielded a 24-28% improvement in GC time and a 2.8-6% improvement in the overall running time. For the HSQLDB benchmark, we modified three lines of code to obtain a 75-76% improvement in GC time, and a 31-32% improvement in overall running time. Finally, modifying the Jikes implementation to use a data-structure aware garbage collection required the modification of 8 lines of the Jikes code. These modifications were tested for runs of the DaCapo 2009 benchmark suite and yielded, on average, a 11-16% improvement in GC time and a 1-2% improvement in the overall running time. This experience shows that the DSA method is applicable for various kinds of applications and it may yield a significant improvement with a small modification effort, where applicable. We did not see a performance degradation in cases where the DSA was not applied.

4.2 Preliminaries

4.2.1 Tracing Garbage Collectors

Many garbage collector algorithms employ at the heart of the collection procedure a tracing routine, which marks every object that is reachable from an initial set of roots. The tracing routine typically performs a graph traversal, employing a mark-stack which contains a set of objects that were visited but whose children have not yet been scanned. The tracing procedure repeatedly pops an object from the mark-stack, marks its children, and inserts each previously

unmarked child to the mark-stack. The order by which objects are inserted and removed from the mark-stack determines the tracing order. The most common tracing orders are DFS (depth-first search) for a LIFO mark-stack, and BFS (breadth-first search) for a mark-stack that works as a FIFO queue.

While tracing is easy to understand and implement, tracing the whole heap efficiently is a more challenging task. In large applications, the heap contains hundreds of millions of live objects, spread over several GBs of memory. We now review some of the challenges in the current tracing procedures.

Work Distribution With the wide adoption of parallel platforms, the ability to utilize several processing units to execute a garbage collection has become acute. However, a good work distribution is not trivial to achieve. In practice, each thread typically uses a local mark-stack, and synchronizes with other threads by a work-stealing mechanism or when too much (or too little) work becomes available in its local mark-stack. While these heuristic methods provide reasonable performance in practice, they are not optimal, and their performance depend on the actual heap shape. While typical heap shapes provide a more-or-less reasonable scalability, there exist extreme cases that seem inherently hard to parallelize. The simplest example of a shape that is difficult to parallelize is a long linked list. Such difficult heap shapes do occur in practice [Sie08, BP10, EP13]. The ability of our newly proposed collector to work with nodes of a data structure allows the tracing to be independent of the heap shape and creates an embarrassingly parallel tracing with excellent scalability.

Memory Efficiency: Locality It is well known that the locality of tracing tends to be bad, as the heap is traversed in a non-linear order. In addition, most objects are small, but the CPU brings from memory the entire cache-line they reside on. If two objects that share a single cache line are traced together, the pressure on memory is reduced, and efficiency increases. In the DSA memory manager, we allocate data structure nodes separately. This substantially improves the locality of the tracing procedure.

Mark-stack Management and Mark-stack Overflow The mark-stack that is used to guide the heap traversal poses a trade-off for a tracing run. A large mark-stack uses more cache lines, more TLB entries and implies more time spent on cache misses. On the other hand, a small mark-stack may cause a mark-stack overflow, which implies a large overhead. When the mark-stack overflows, it is possible to restore its state using an additional tracing over the heap. However, restoring the mark-stack is a very expensive operation, even if it happens just once in a collection cycle. Garner et al. [GBF07] reported that mark-stack operations take 11%-13% percent of the trace. Dynamically allocated mark-stacks are sometimes used to handle such issues [BBYG$^+$05], but the management of these stack chunks has its own cost. The DSA memory manager reduces the use of mark-stacks in two ways. First, data structure nodes (that have not been removed) are not put in the mark-stack. Second, objects reachable from such data structure nodes are traced in a controlled manner in order to decrease mark-stack use.

4.2.2 Data Structure Nodes

Data structures are often implemented using *nodes*. These nodes are intended to abstract the data structure from the data itself, and allow data-structure algorithms to operate on abstract generic nodes, ignoring the specific data stored inside. Data is added to the data structure by allocating a new node that represents the data and inserting it into the data structure. Data is removed from the data structure by unlinking the node that represent the data from the data structure. Given this node-representation of data structures, the interface requires that the programmer specifies the classes used to represent data structure nodes and also specifies when the nodes are removed from the data structure. Many times a single node type (class) is used to represent the nodes of the data structure and so declaring a single class for the data structure suffices. Since a *remove* or a *delete* operation is typically a well-defined logical operation on nodes in the data structure, it is usually very easy to identify code locations in which a delete is executed. Again, many times there are very few such code locations that perform a delete and often the delete would happen in a single method of the data structure. These typical behaviors significantly simplify the programmer efforts for interacting with the DSA interface.

We emphasize the separation of data structure nodes and the stored data. In standard library data structure, the data structure node is a (private) internal class that is accessed only from within the data structure. The stored data, pointed to by data structure nodes, is used in an arbitrary (complex) manner by the program. For example, in the standard (library) data structure java.util.HashMap <Integer,String>, the data structure node is HashMap.Entry.

4.3 Overview

In this section we provide an overview of the DSA interface and the corresponding DSA memory manager. We start by defining the life cycle of a node. An instance of a *node* class is first *allocated*, and then *inserted* to the data structure. Many times these two operations occur in close proximity, usually in the same function. At the application's request, the node is *removed* from the data structure. At some later point, the node becomes unreachable by the application threads, and is then *freed* or *reclaimed* by the garbage collector.

The DSA interface lets the programmer specify which classes represent data structure nodes. For a data structure to be treated as such by the DSA memory manager, the interface must be used to annotate the class in order to announce it as a data-structure node class. Allocating an instance of an annotated class implicitly notifies the memory manager that the allocated object will be inserted into the data structure. A second part of the interface is a new garbage collector function, denoted *remove*, which lets the program notify the garbage collector that a node is being deleted from the data structure. When a node is removed from the data structure, the programmer must explicitly call the memory manager's *remove* function. As a consequence, the node will later be de-allocated during a garbage collection cycle, but only after the node becomes unreachable from the program roots. We stress that a *remove*d node may still be reachable by the application, even after being deleted from the data structure. In this case, it will

not be erroneously reclaimed. Sometimes a node is allocated for insertion to the data structure but the insertion fails and the node is discarded before even being added to the data structure. The programmer should be aware of such a case, and this node needs to be passed to the memory manager's DSA interface *remove* function.

The DSA memory manager allocates the data structure nodes on specially designated pages, and tracks for each node whether it was passed to the *remove* function. During a garbage collection cycle, the garbage collector assumes that a non-removed node is still a member of a data structure and hence is still alive. Therefore, the garbage collector can treat the remaining set of non-removed data structure nodes as additional roots; these nodes are marked as alive and their children are traced.

During a garbage collection cycle, the tracing procedure benefits from this large set of objects that are known to be alive and are co-located in memory. Moreover, there is no dependency between the tracing of the nodes in this set. Thus, we let each thread grab (synchronously) a bunch of pages, and traces all live nodes that reside on these pages and add their descendants to the mark-stack. Pages are traced locally by a single thread and in memory order of nodes; that is, co-located nodes are traced consecutively.

Memory order tracing improves locality since co-located nodes are traced together. It also allows hardware prefetching, which further reduces memory latency. Multiple threads benefit from the lack of dependency, leading to good work distribution with lightweight synchronization. The "normal" roots are traced by an unmodified tracing procedure, and may exhibit bad work distribution. A thread that runs out of nodes to trace in the regular trace can then turn to tracing data structure nodes, which are always available for tracing. This provides an excellent load balance between threads that might suffer from idle times in a traditional garbage collection tracing.

4.4 Memory Management Interface

In this section we define the DSA interface between the application and the memory manager. The DSA interface includes one annotation and two functions that the memory manager provides. The annotation signifies that a class' objects are used as data structure nodes. The first interface function is used to let the memory manager know that a node has been removed from the data structure. The second interface function should be used infrequently. It asks the garbage collector to perform an extensive cleaning collection to identify and reclaim all nodes that have been removed from the data structure without proper notification from the program. We next name and explain each of these interfaces.

The annotation provided to the program is the *@DataStructureNodesClass* annotation, which is applicable for classes only. By annotating a class as *@DataStructureNodesClass*, every instance of this class is considered a node of a data structure and assumed alive until the application notifies the memory manager that the node is removed (via the first interface function). We denote a class that is annotated by the *@DataStructureNodesClass* annotation as an *annotated class*, and an instance of an annotated class is denoted an *annotated object*.

The first interface provided to the program is the *remove* function. It is the responsibility of the programmer to make sure that every annotated object is passed to the memory manager's *remove* function after it is deleted from the data structure, so that the garbage collector may free this object. Failing to do so would result in a temporary memory leak, and may slow down the application (though not result in memory failure). Unlike the free function in C, the program may still reference an object that was passed to the memory manager's *remove* function.

The parameter passed to *remove* must be an annotated object. Often this can be checked at compile time. Otherwise, a runtime check can be used to trigger an adequate exception. It is not necessary to limit the number of times that the program calls *remove* with the same object. An annotated object that was passed to the memory manager *remove* function is referred to as an object that was *announced as removed*.

A second interface function provided to the programmer is the *IdentifyLeaks()* function. This function is optional, and should be called infrequently or not at all. This function tells the collector that some objects may have been deleted from the data structure without a proper corresponding call to the collector's *remove* function. This may create memory leaks that this function is meant to solve (at a cost). A call to this function will make the next garbage collection cycle perform a full collection, which ignores knowledge about the data structures, finds and eliminates all such memory leaks.

The implementation of these functions is provided by the memory management subsystem of the virtual machine, and is described in Section 4.5.

4.5 The Memory Manager Algorithm

We now present the DSA memory manager algorithm that exploits the additional knowledge provided by the programmer via the DSA interface. The algorithm is presented as a modification over an existing memory manager algorithm, called the *basic* memory manager. For the current presentation we assume that the *basic* memory manager uses a mark sweep collection policy. Mark sweep collectors have a choice between keeping the mark bits inside the object or in a mark table aside from the objects. We assume that the *basic* collector places the mark bits in a side table which is a common choice for commercial collectors[BBYG$^+$05, DP00], and fits well into our algorithm. In Section 4.7 we present a possible modification needed for running our algorithm on the high performance Immix collector, which places the mark-bits in the object header.

The dynamic class loader identifies classes annotated by the proposed *@DataStructure-NodesClass* annotation. For each annotated class $\mathscr{C}$, the class loader creates a memory manager for $\mathscr{C}$'s objects. We call such a memory manager a *$\mathscr{C}$ memory manager* and we say that allocations of $\mathscr{C}$'s objects are directed to the *$\mathscr{C}$ memory manager*.

The *$\mathscr{C}$ memory manager* holds a set of blocks (allocation caches) designated for the $\mathscr{C}$ objects. The $\mathscr{C}$ memory manager allocates on its designated blocks which are separate from the rest of the heap. Each such block is associated with a table denoted the *member-bit table*. The member-bit table has a bit for each object in the block, specifying whether the chunk is

Algorithm 4.1 Allocation

Output: Address of the newly allocated object of type $\mathscr{C}$ (DS node)

```
1:  if allocation cache is empty then
2:     C-allocator.getNonFullAllocationCache();
3:     if all C allocation caches are full then
4:        basic.getEmptyAllocationCache();
5:        C-allocator.registerAllocationCache();
6:     end if
7:  end if
8:  allocated = allocCache.alloc(); {//same algorithm as basic but uses C allocation cache}
9:  memberBit(allocated) = true
10: return allocated
```

Algorithm 4.2 Remove

Input: Object object

```
1: if object type is not annotated then
2:    throw new Exception();
3: end if
4: memberBit(object) = false
```

currently a member in the data structure. In fact this bit is set when the object is allocated and is reset to 0 when the *remove* function is invoked with this object. Note the linkage between the member-bit and the mark-bit that the garbage collector uses to mark objects reachable from the roots. When the member bit is on (and if the program properly reports membership to the collector through the interface) then we know that the object is reachable. On the other hand, when the member bit is not set, we know that it has been removed from the data structure, but it might still be reachable from the roots. Thus, having the member-bit set (and assuming proper use of the interface) implies that the mark-bit can be set at the beginning of the trace. We denote a block that is used to allocate annotated objects (of a data structure) as an *annotated block*.

The $\mathscr{C}$ *memory manager* allocates objects just like the *basic* memory manager allocates objects. If $\mathscr{C}$'s blocks are all full, an empty block is requested from the global pool of free blocks of the *basic* memory manager, and the allocation request is served from that block. Upon allocating an object, the $\mathscr{C}$ memory manager also sets the *member-bit* corresponding to the allocated object. The allocation procedure is presented in Algorithm 4.1.

During a call to *remove*, the memory manager clears the member-bit corresponding to the passed object. At that point the object may still be reachable as some threads may hold pointers to that node even when it is not in the data structure. The removal procedure is presented in Algorithm 4.2. The algorithm should be executed in an atomic manner, for example by using the CAS instruction.

The tracing procedure is presented in Algorithm 4.3. We now discuss it in detail. The first loop (Step 1) initializes the mark bits of data structure nodes (annotated objects) to their member bits. This means that an object that belongs to the data structure (and has its member bit set) is not scanned by the standard tracing procedure, since the trace ignores marked objects. We will

Algorithm 4.3 Tracing procedure

```
 1: for each annotated block do
 2:    Copy member-bit table to mark-bit table (block){//marks live annotated nodes}
 3: end for
 4: roots locations = basic.collectRoots();
 5: barrier(); {wait for other threads to finish}
 6: basic.trace(roots locations) {trace using basic GC }
 7: for each annotated block do
 8:    {//Trace annotated nodes}
 9:    for each object in the annotated block do
10:       if memberBit(object)==true then
11:          Push the object's unmarked children to the local mark-stack
12:       end if
13:       if mark-stack size reaches a predetermined bound then
14:          Transitively trace local mark-stack.
15:       end if
16:    end for
17: end for
```

give these objects a special treatment at the end of the trace in Step 7. This initial copying of the member bits into the mark bits is executed very efficiently as it can be performed at the word level rather than for each bit separately.

The barrier in Step 5 is used to prevent races between the unsynchronized access to the mark bits during the copying of the member bits at Step 2 and the synchronized access to the mark-bits at Step 6 during the marking of objects whose member bit are reset (announced as removed).

In Step 6 the trace procedure invokes the *basic* tracing procedure, which traces the root objects. A thread that exhausts its available work for this trace proceeds to work on data structure nodes (annotated objects) in Step 7. In Step 11 the children of each annotated object that has its member bit set are pushed to the local mark-stack. Once in a while, when the mark-stack gets filled beyond a predetermined bound, all objects in the local mark-stack are traced transitively.

The *basic* collector may provide parallelism for the original trace. However, parallelism can be limited by the heap shape or heuristics used. The scanning of the data structure nodes is embarrassingly parallel as we need to get their information from consecutive addresses in the memory. Work distribution for a multithreaded execution is done by dividing iterations of the for loop at Step 9 between threads. The granularity of work distribution can be as small as a single iteration per thread, or as large as several blocks per thread. Note that iterations can be executed in any order.

4.5.1 Handling missed *remove* operations

The above description assumes that the programmer never forgets to report a deleted object. In this section we discuss a simple solution to leaks of annotated objects whose removal is not properly reported. We call an annotated object *leaked* if it is not reachable from the roots,

but has not been announced as removed via the *remove* procedure. To reclaim such objects we can simply invoke the regular garbage collector on the entire heap. The garbage collection identifies all unreachable objects whose member-bit is set. It then reclaims these objects and resets their member-bit. A simple implementation of this idea appears in Algorithm 4.4. This algorithm can be called when the garbage collection does not free enough space for required allocations, or following an explicit request by the application. The programmer may wish to call this procedure for example when a large data structure becomes unreachable and the programmer does not wish to announce the removal of each member object explicitly. Also, the programmer may invoke this procedure "once in a while" if the *remove* announcement are known to be inaccurate. It is also possible for the system to automatically invoke this procedure once every ℓ collection cycles, for an appropriate ℓ.

Algorithm 4.4 IdentifyLeak tracing procedure

```
1: roots locations = basic.collectRoots()
2: basic.trace(roots locations) {//trace normally using basic GC}
3: barrier(); {//wait threads to finish tracing}
4: for each annotated block do
5:     {//if(member-bit=true and mark-bit=false) then member-bit:=false}
6:     Member-bit table = member-bit table AND mark-bit table
7: end for
```

There is no need to modify other phases of the GC algorithm, e.g. sweep.

4.5.2 Performance Advantages over Standard Tracing

Next, we discuss the improvements of the DSA tracing algorithm over standard tracing algorithms that are not data-structure aware.

Parallel Tracing and Work Distribution The DSA collector can be easily parallelized with a good work distribution. The iterations in the loop of Step 9 can be executed in any order, and thus embarrassingly parallel. Furthermore, since the embarrassingly parallel Step 11 is executed after the standard tracing procedure, the excellent work distribution of Step 11 can aid a possible bad work distribution of the original algorithm. While some threads may be delayed by some unbalanced work, other threads need not wait, but can proceed with the data structure related work.

Disentanglement of Bad Heap Shapes The DSA garbage collector cleanly solves the complicated problem of data structures with deep heap shapes. Such data structures may harm parallel tracing [Sie08, BP10, EP13], but if such structures are annotated using the DSA interface, then they can be parallelized even in the presence of many-cores. In fact, the existence of a large data structure with deep heap shapes will *improve* the scalability of the tracing, because imbalanced work distribution would be resolved by the starving threads switching to work on scanning the data structure.

Locality The tracing of data structure nodes in the DSA collector is executed in memory order rather than in an arbitrary order determined by object pointers in the heap. Thus, locality of the tracing procedure and the ability of the hardware to automatically prefetch required data improve substantially. Two nodes that reside on the same cache line suffer only one cache miss, and TLB misses are similarly reduced.

Mark-stack Management and Control The mark-stack used in Step 6 of the DSA algorithm is expected to be smaller than the mark-stack used for the original trace. The (annotated) data structure nodes and objects reachable from them (only) will not appear in the mark-stack in the first stage of the algorithm in Step 6. Next, in the second stage at Step 7, the nodes inserted into the mark-stack are those that are reachable only from nodes of the data structure. Furthermore, when the mark-stack passes a predetermined limit, we trace the current list of objects before we add more to the stack. This is expected to provide smaller mark-stacks on average.

4.6 Programmer View

In this section we review the changes required in order for a programmer to use the DSA interface. We assume that the garbage collector in the runtime used supports the DSA interface. Given an existing program, the required changes are very lightweight. We separate the discussion to the design of data structures (which may be put in a library) and the use of data structures in a program. Let us start with the former.

4.6.1 Data Structure Designer

The designer of library data structures and also programmers of ad-hoc data structures can add the interface for data-structure-aware garbage collector with a minor effort.

First, the programmer has to annotate the data structure nodes by the *@DataStructureNodes-Class* annotation. Second, the programmer has to call the interface *System.gc.DataStructure-Aware.remove(N)* method whenever a node *N* is removed from the data structure. We stress that calling the *System.gc.DataStructureAware.remove(N)* method does not free the object *N* but rather lets the garbage collector reclaim it later when the object becomes unreachable. Therefore one can call this method even if *N* is still used by the calling thread or other threads in the system.

Third, if the data structure contains a method for a massive delete of a set of nodes in the data structure or even the entire data structure, then every node should be passed to the memory manager's *remove* function, or alternatively the *IdentifyLeaks()* interface must be invoked.

Sometimes, the question whether a library data structure benefits from the DSA interface depends on the context. For example, a HashMap can be used in a context where it holds many items and is never entirely discarded, but it can also be used in another context where it holds a few items and is frequently discarded. The DSA interface is beneficial only in the former context. For such cases it is advisable for the designer of a library data structures to create two

copies of each data structure: one with the DSA interface and one without. This allows users to use the garbage-collection-aware version of the data structure only when beneficial.

4.6.2 Using a Library Data Structure

Programmers often invoke standard data structures whose implementation is provided in a library. We now discuss the requirements of a programmer that uses a data structure that supports the DSA interface. The only requirement of such a programmer is that he will be aware of the deletion of entire data structures. Usually, nodes are inserted into and removed from the data structures. But once the data structure is not needed anymore, the program may unlink it and expect the garbage collection to reclaim all its nodes. This is the case that requires extra care for the DSA memory manager since such nodes will not be reclaimed. To solve this, the programmer may either actively invoke Algorithm 4.4 using the IdentifyLeaks interface to reclaim the unreachable data structure nodes, or the programmer can delete all nodes from the data structure before unlinking it from the program roots.

If the library provides both a DSA and a non-DSA versions of the data-structure, the programmer needs to decide which version to use. A programmer may use both versions simultaneously in the same program.

4.6.3 Some Experience with Standard Programs

To check how difficult and effective the use of the DSA interface is, we have modified KittyCache[Kit09], pjbb2005[SPE05], the HSQLDB benchmark from the DaCapo test suite [BGH+06], and the JikesRVM Java virtual machine[AAB+00]. We are not authors of the these programs; nevertheless, we discovered that the required modifications were easy and did not require a deep acquaintance with the programs involved.

In all our attempts to use the DSA interface, we only used it when the program had a substantial fraction of its data kept in a small number of data structures. We did not attempt to use the DSA with a large number of small data structures, and we believe that the overhead and fragmentation that such use will create will make it non-beneficial. Note that in general, when the use of DSA turns out to be non-beneficial for any application, it is possible to remove all overhead by simply not using it for that application.

KittyCache The KittyCache is a simple, yet efficient, cache implementation for Java. It uses the standard library ConcurrentHashMap for the cache data, and the standard library ConcurrentQueue to implement the FIFO behavior of the cache. In order to make KittyCache use GC-aware data structures we only needed to modify the library code of these two data structures and nothing in the KittyCache application itself. The changes to the library data structures are described in Subsection 4.6.4 below.

pjbb2005 The SPECjbb2005 benchmark emulates a wholesale company, and saves the warehouses data in a standard HashMap. Again, to gain the advantages of the DSA interface, we

modified the library code for the HashMap data structure.

The pseudo SPECjbb2005 (pjbb2005) is a small modification for the research community, which fixes the number of warehouses and measures running time for a specific workload instead of throughput. Additionally, it runs the benchmark multiple times to warm up the JIT compiler and measure only the last iteration. After each iteration, the data structure is discarded without freeing its entries. For this application one more modification was required to make it work with the DSA interface. To avoid going over the data structures and applying a remove notification for each node, we added a call to *IdentifyLeaks* at the end of each such iteration. This amounts to adding a single line of code to the benchmark itself. (A similar line would be needed also if we were using the original SPECjbb2005 benchmark. It would deal with entire data structure deletions at the end of each warmup phase.)

HSQLDB The HSQLDB is a SQL database written in Java. The HSQLDB benchmark in the DaCapo 2006 suite tests the database performance. Most of the database data is stored in a tree-like structure, a custom (and rather complex) data structure. Yet, adding garbage collection awareness was simplified by the fact that the main data structure node has a delete function which is called by the original code in an orderly fashion whenever a node is deleted. We annotated the data structure node class, and called the memory manager remove inside the (already existing) database delete function. Only two lines were added to the database code.

Similarly to pjbb2005, the entire data structure is discarded in each iteration. Thus, we used the *IdentifyLeaks* interface before each iteration started.

The newer DaCapo 2009 test suite contains another benchmark that tests the performance of a different SQL database called h2. The h2 database uses a different tree-like structure to store its data. Still, it already contains a delete function, so applying the DSA interface to this database was as easy as applying it to the HSQLDB database. Although we could easily modify this benchmark to be GC-aware, we ended up not measuring its performance because the Jikes RVM could not execute this benchmark.

JikesRVM The JikesRVM uses a library version of LinkedHashMap for organizing the ZipEntries. The LinkedHashMap forms a substantial data structure for Jikes RVM. This data structure is used (to the best of our understanding) to handle entries in the executed Jar file. We modified the library implementation of LinkedHashMap to be GC-aware. Library modifications are discussed in Section 4.6.4.

In addition, for the JikesRVM, it turned out to be useful to also annotate the ZipEntry objects themselves (the objects that the hash nodes reference). These nodes are not linked in a data structure of their own, yet, they are deleted only when their parent (a hash node) is deleted and so there is a well defined point in the program when such a node gets unlinked from the data structure that references it. These nodes typically become unreachable shortly after their parent node in the hash data structure gets deleted. We had to identify a couple of places where a node is allocated and not inserted to the data structure (e.g. cloning a node). For these allocation points we cleared the member bit immediately after allocating the node. This was slightly more

complex (but still only a few hours of work), and required the modification of 4 lines of code in Jikes RVM.

4.6.4 Some Experience with Standard Data Structures

For data structure examples, we have transformed several data structures from the classpath GNU project (version 0.97.2) to fit the DSA interface. The changes for HashMap include three lines of code. The modified code can be shortly presented in a diff-like syntax.

```
+@DataStructureNodesClass
static class HashEntry<K, V> extends

public V remove(Object key){
  if(equals(key, e.key)){
+  System.gc.DataStructureAware.remove(e);

public void clear(){
  Arrays.fill(buckets, null);
  size=0;
+ System.gc.DataStructureAware.IdentifyLeaks();
```

The changes to the LinkedHashMap, TreeMap and LinkedList are very similar, and we do not present them here. It was possible to use a simple inspection of method names in order to determine the code locations where nodes are removed. In the TreeMap data structure, nodes are removed in a dedicated function, which makes the modification even simpler.

We also modified the ConcurrentHashMap and ConcurrentQueue from the classpath GNU (version 0.99.1-pre). Special care was needed in the rehash function of ConcurrentHashMap since objects are cloned. In the ConcurrentQueue data structure, special care was taken to call remove on the data structure sentinel. These changes took us few hours to identify without prior acquaintance with the data structure, and we assume that a programmer of a custom data structure will be able to identify these places even more easily.

In all our modifications of a library data-structures, we generated a copy of the original data structure implementation and added the DSA interface to the copy. We then used the copy whenever we needed to use the interface.

4.6.5 Shortcoming of our algorithm

For some programs we were not able to improve performance using our interface. We note two main reasons for that.

1. The program has no dominant data structure. This is the case for most of the DaCapo benchmarks.

2. The program uses a custom data structure without a clear interface (no remove function). An example to the above is the PMD benchmark in the DaCapo suite 2009. There exists a

dominant data structure, whose nodes are objects of the *pmd.ast.Token* class. However, its data-structure pointers are public, and are modified in many code locations. We cannot tell if the programmers of PMD have invariants in mind that help identify code locations in which a node is removed, or whether they would also find it difficult to identify these locations.

4.7 Adaptation for the Immix Memory Manager

We have implemented the DSA interface and algorithm on top of the JikesRVM [AAB+00] environment version 3.1.3. Initially, we used the *basic* memory manager that uses a simple mark sweep garbage collector. However, although the DSA mechanism achieved significant improvement, this memory manager runs very slowly compared to other high performance memory managers. So, we proceeded and implemented our algorithm over the high performance Immix[BM08] memory manager. This required several modifications that we describe below.

The main issue with the Immix memory manager is that it puts the mark bit in the header of each object and not in an auxiliary table. This complicates the implementation of the DSA algorithm. When the DSA algorithm traces a data structure node, it needs to check whether its children are marked or not. If the mark bit is located in the children headers, the tracing would incur the cost of cache misses over accessing the children, even if both are members of the data structure. Even on a singly-linked list, each child is accessed at least twice, possibly by different threads. Thus, a naive implementation would nullify the memory locality benefit of our algorithm.

A possible solution is to identify the children which are data structure objects, and ignore them during the trace. Intuitively this seems correct since nodes are deleted from the data structure by unlinking data structure pointers to them. Thus, pointers from within the data structure to deleted nodes should not exist. However, ignoring children during trace may lead to correctness problems, especially when relying on an untrusted programmer. Instead, in the case of a child that is an annotated object we first check its member bit. If the member bit is on, there is no need to further trace the object, since it will be traced by the DSA tracing. Since the member bits are placed in a side table, checking the member bit usually hits the cache, instead of taking the cache miss over the child.

Another problem is marking all annotated objects prior to root tracing. When the mark bit is placed in the object header, marking all objects requires loading all of them to memory. Therefore, we modify the algorithm in the following way. The algorithm first traces all annotated objects, and then continues with the root tracing. This provides an initial embarrassingly parallel tracing for the data structure node but tracing other nodes on the heap does not get a balancing guarantee. The tracing function, for the case that the mark bit resides in the object header, is presented in Algorithm 4.5.

Finally, the *IdentifyLeak* tracing procedure cannot go over the side tables (mark bit and member bit) to fix unreachable member nodes. Instead, it needs to go over the nodes themselves and check whether the nodes are alive or not. Since the IdentifyLeak tracing procedure is not

Algorithm 4.5 Tracing procedure for Immix

```
 1: roots locations = Jikes.collectRoots()
 2: barrier(); {//same as immix}
 3: for each annotated block do
 4:    for each object in the annotated block do
 5:       if memberBit(object)==true then
 6:          Mark object
 7:          for each data structure pointer P of the object do
 8:             if memberBit(P)==false then
 9:                ProcessChild(P) {//push to markstack if not marked}
10:             end if
11:          end for
12:          for each non-data structure pointer P of the object do
13:             ProcessChild(P)
14:          end for
15:       end if
16:       if the mark-stack size reaches a predetermined bound then
17:          Transitively trace local mark-stack.
18:       end if
19:    end for
20: end for
21: Jikes.trace(roots locations) {//trace using Immix trace}
```

called frequently, this modification should not affect the algorithm performance significantly. In our implementation we set the triggering of the *IdentifyLeak* tracing procedure to include only specific invocation of the IdentifyLeaks interface, but not ”once-in-a-while” invocation.

An alternative that we did not use is to trace the annotated objects after the Immix tracing loop, but modify the trace to not scan data-structure nodes with a set member bit. The problem is that this adds some performance overhead to the core of the tracing loop and so may reduce performance.

In Algorithm 4.5 Step 16 the local mark-stack is traced after a predetermined bound, which is a parameter of the algorithm. The value of this parameter represents a tradeoff. A lower value decreases the size of the mark-stack, which is good, but it also reduces the locality advantage for tracing consecutive objects, which is not good. In our implementation the mark-stack is traced after scanning all DSA objects that correspond to a single word in the member-bit table.

4.8 Implementation and Evaluation

We compared the DSA collector with the original unmodified Immix memory manager [BM08]. For the DSA memory manager, the data structure nodes and the member-bit table are part of the heap and are accounted for heap space usage, so the comparison is fair. The measurements were run on two platforms. The first features a single Intel i7 2.2Ghz quad core processor (HyperThreading enabled), an L1 cache (per core) of 32 KB, an L2 cache (per core) of 256 KB, an L3 shared cache of 6 MB, and 4GB RAM. The system ran OS-X 10.9 (Mavericks). The

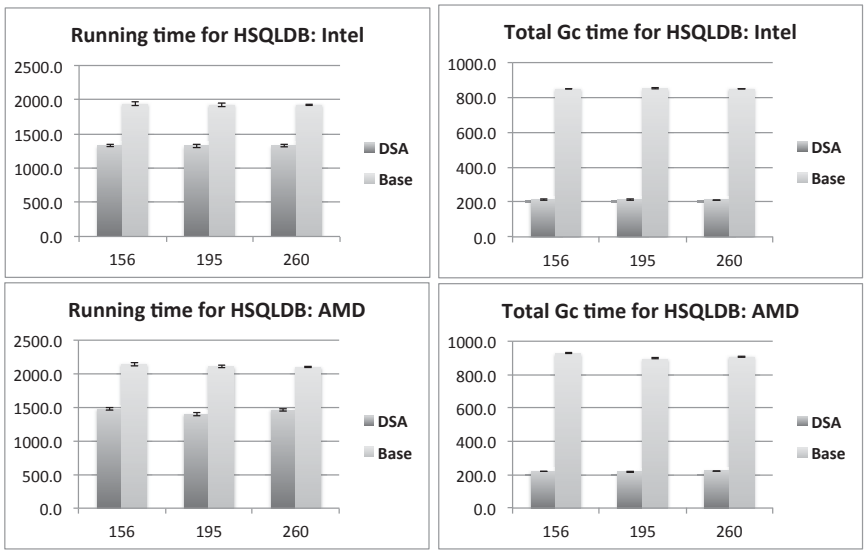

Figure 4.1: Total running time and total GC time (in milli-seconds) for *HSQLDB* benchmark. The *x*-axis is the heap size used.

second platform featured 4 AMD Opteron(TM) 6272 2.1GHz processors, each with 16 cores, an L1 cache (per core) of 16K, L2 cache (per core) of 2MB, an L3 cache of 6MB per processor, and 128GB RAM. The machine runs Linux Ubuntu with kernel 3.13.0-36.

For each benchmark we measured performance of the modified benchmark with the DSA memory manager, and the unmodified benchmark with the original Immix collector. For each benchmark we report measurements on different heap sizes. We invoked each benchmark 10 times (10 invocations), and reported the average (arithmetic mean) of their running time and the average of the garbage collection time; the error bars reports 95% confidence interval. To reduce overhead due to the JIT compiler, in each invocation we measured only the 6th iteration, after 5 warm-up executions.

The first benchmark we measured was the HSQLDB benchmark from the DaCapo 2006 suite. The HSQLDB is a SQL database written in Java, and the HSQLDB benchmark test its performance. The changes to the benchmark code were discussed at Section 4.6. The minimal heap size was 130MB. We ran the program with 1.2x, 1.5x, and 2x heap sizes. The benchmark calls *System.gc()* manually, so garbage collection is invoked before the heap is exhausted and so the performance differences between different heap sizes were negligible. The total running time and total GC time are presented in figure 4.1. The DSA version improved GC time by 75-76% and overall time by 31-32%.

Next, we present the measurements of the DSA garbage collector behavior with the pseudo SPECjbb2005 benchmark. The pjbb2005 benchmark models a wholesale company with warehouses that serve user's operations. The warehouse data is mostly stored in a HashMap, which we modified in the DSA algorithm; the changes to the benchmark code were discussed at Section 4.6. We ran the benchmark with 8 warehouses, and fixed 50,000 operations per warehouse. The minimal heap size was 500MB; we ran the program with 1.2x, 1.5x and 2x heap sizes. The

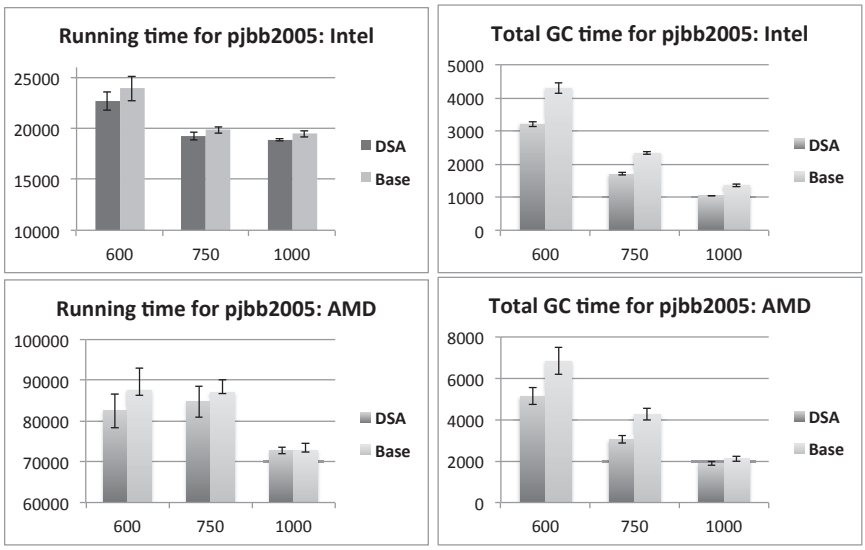

Figure 4.2: Total running time and total gc time (in milli-seconds) for the pjbb2005 benchmark with 8 warehouses and 50,000 transactions per warehouse. The *x*-axis is the heap size used.

total running time and the total GC time are presented in figure 4.2. The DSA version improved GC time by 24-28% and overall time by 2.8-6%.

We then measured the DSA garbage collector with the KittyCache stress test. It allocates a cache of size 250K entries, and then executes 2,000,000 put commands. We ran the cache with different heap sizes; the minimal heap size was 200MB. The total running time and total GC time for various heap size are presented in figure 4.3. The DSA version improves GC time by 40-45% and overall time by 4-20%.

For these benchmarks we estimated the mark-stack usage by the baseline and the DSA versions. Accounting the mark-stack exact size requires extensive synchronization. Instead, we noticed that each thread has a local mark-stack and there is a shared pool of local mark-stacks when the local mark-stack is exhausted (denoted work-packets). Thus we measured the number of work-packets in the shared pool. Each work-packet (or local mark-stack) is exactly a single 4K page. While this is not an exact evaluation, it provides the amount of memory reserved for the mark-stack usage and approximates the mark-stack usage. We executed each benchmark 3 times with 1.5x heap size and record the maximum shared pool size in each collection cycle. We then computed the average and standard deviation of the obtained record in the 3 executions and the results are depicted in Figure 4.5. For the pjbb2005 and kittyCache the mark-stack size was reduced by a factor of 3.85-4.15. For the HSQLDB benchmark the difference is very small, but the mark-stack size is rather small even in the baseline algorithm. The error bars represents standard variant. For the baseline algorithm the variance was very high, meaning that different collection cycles (in the same execution) requires different mark-stack size. In contrast, the variance is much smaller for the DSA memory manager.

Finally, we experimented with the DaCapo benchmark suite 2009. We were able to execute only 6 benchmarks: avrora, luindex, lusearch, sunflow, xalan and pmd. The other benchmarks

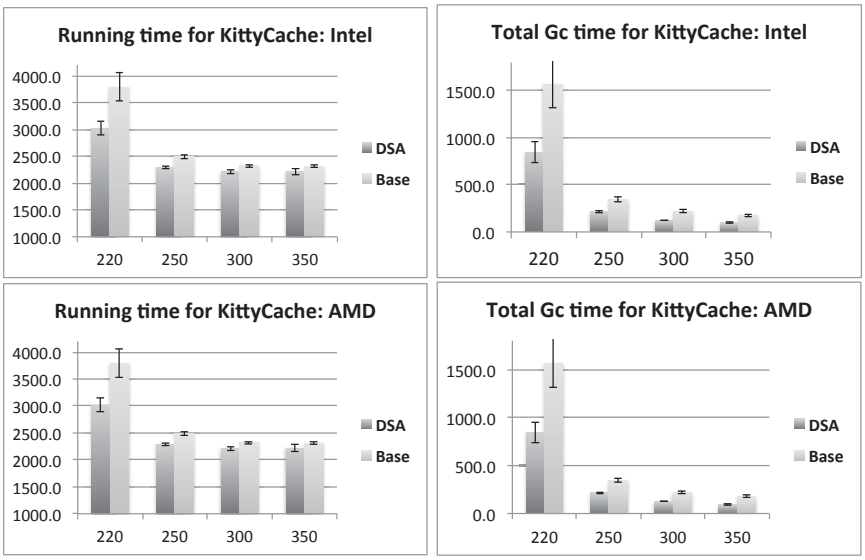

Figure 4.3: Total running time and total GC time (in milli-seconds) for *KittyCache* stress test with 250K entries. The *x*-axis is the heap size used.

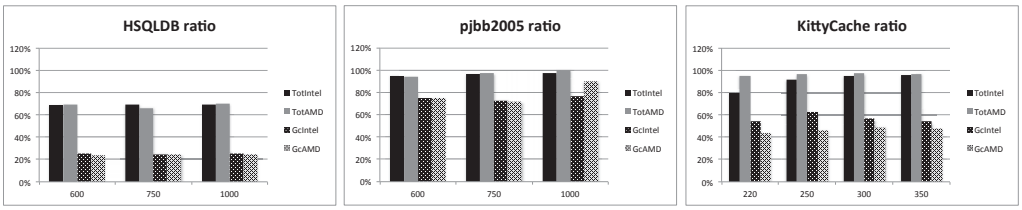

Figure 4.4: Summarizing Figures 4.1, 4.2 and 4.3. Shows the running time ratio between the DSA algorithm and the original Immix. The *x*-axis is the heap size used. Lower is better.

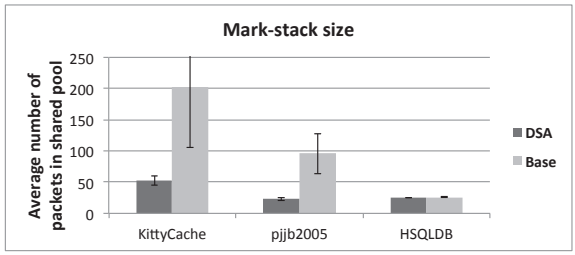

Figure 4.5: Number of work-packets in shared mark-stack.

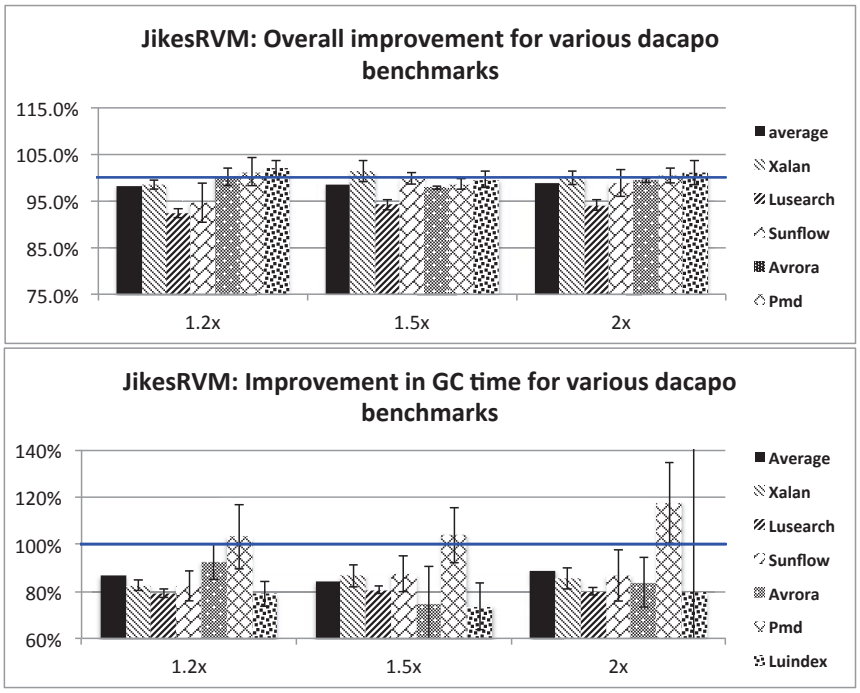

Figure 4.6: Comparing total running time and gc time for various benchmarks in the DaCapo suite. The figure only present the ratio between the modified JikesRVM and the unmodified Immix. Only a JikesRVM internal data structure was changed.

could not be executed on the JikesRVM Java virtual machine. In avrora, lusearch, and xalan, a data structure used by the JikesRVM Java virtual machine was the dominant data structure. Therefore, we added the DSA interface to the JikesRVM data structure. In the pmd benchmark, a custom data structure is significant. However, the code semantics does not expose a data-structure-like interface, so adding the DSA interface to the data structure requires deeper understanding of the benchmark (if at all possible). Luindex makes an insignificant amount of allocation, and the GC load was low. In sunflow, a dominant data structure is an array of image samples, which is already GC friendly. Therefore, we did not add the DSA interface to any of the benchmarks in the DaCapo suite 2009, and modified only the JikesRVM internal data structure. For each of these benchmarks, we compared the running times before and after modifying the JikesRVM major data structure. We do the comparison for various heap sizes, namely 1.2x, 1.5x, and 2x of the minimal running heap size. The comparison is depicted in figure 4.6. For lack of space, the comparison for the AMD machine is not depicted. The average improvement in GC time is 11-16% and the average implemented in overall time is 1-2%.

Note that the pmd benchmark suffers a slowdown due to our modification. The pmd data structure has a linked-list like structure, thus tracing it is poorly parallelized. A possible explanation to the slowdown is that in the original execution, a single thread traces this data structure while other threads trace other nodes, including the JikesRVM major data structure. In the modified execution, all threads start by scanning the JikesRVM data structure. Only later, a single thread traces the pmd data structure, which explains the slowdown in this case. Indeed,

there was no slowdown where the basic memory manager was the mark-sweep and the mark-bits were put in a side table. We stress that we were not able to apply the DSA mechanism to the pmd program, but only to the JikesRVM that executes this program.

4.9 Memory Leaks

When using the DSA interface, there is a possibility that the programmer will miss code locations in which nodes are removed from the data structure, potentially causing memory leaks. We now discuss our experience in this matter.

In the benchmarks we modified we used two methods to make sure that there were no memory leaks in the observed executions. First, we modified the IdentifyLeaks interface to check whether there existed an unreachable object that was not declared as removed. Second, we measured the number of occupied lines (which measures the amount of live memory) and ensured that it was similar in both the DSA and baseline execution.

The data structures that we modified are widely used in practice and their implementation consists of hundreds to thousands of lines of code. We did not study (or even read) the entire implementation. Instead, we searched for an internal class called "Node" or "Entry" and a function called "remove" or "delete". For all applications, except for the ConcurrentQueue library data structure, we successfully applied the DSA interface without any memory leaks in the first attempt. Namely, we never missed a deletion of objects from the data structure.

The one application for which we erred, was the ConcurrentQueue for which our first DSA version leaked memory. After inspecting the code of the remove function, which is actually a pop for the queue data structure, we discovered that when a node A is popped from the queue, it is not actually unlinked from the data structure. Instead, it becomes the new sentinel and the previous sentinel is the node that actually gets unlinked from the data structure. Incorrectly, we marked the currently popped node A as removed. The result of this mistake is that the first sentinel is never removed, and subsequently, all nodes in the queue becomes forever reachable from an un-removed data structure node. This meant that no object could be reclaimed. In our implementation this implied some frequent GC cycles (because not enough memory was freed) culminating in a cycle that was not able to proceed and then IdentifyLeaks was called. At that point, that first sentinel node was reclaimed and all other leaked nodes were reclaimed with it. From there on, performance was back to normal. Probably, the programmer of the queue would not make such a mistake.

On top of mistakes, there could also be applications and data structures for which it is easy to identify most of the removals but not all of them. We don't expect this to be frequent, but for such cases there is an easy solution. It is possible to call IdentifyLeaks once every k collection cycles. In Figure 4.7 we measure the performance overhead with a typical parameter $k = 5$. *Orig* stands for the baseline implementation with no DSA, em DSA stands for the DSA benchmark when IdentifyLeaks is called only when necessary, i.e., when an out-of-memory exception is about to be thrown, and *DSA-5* stands for the DSA benchmark when IdentifyLeaks is called once every 5 collection cycles. As in 20% of the collections the DSA interface is not used, the

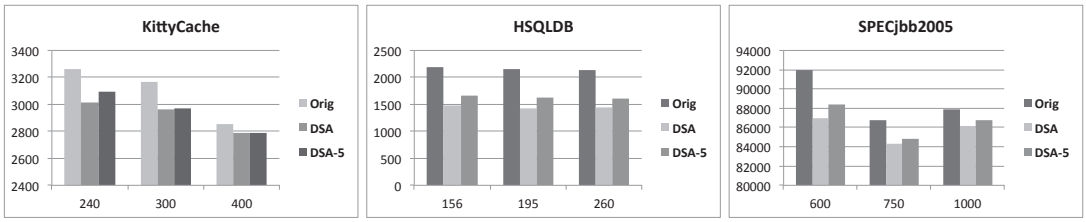

Figure 4.7: Throughput comparison for the original implementation, the DSA standard implementation, and an implementation that runs IdentifyLeaks once every 5 collection cycles.

advantage is reduced by 19-29% depending on application and heap size.

4.10 Additional Related Work

The interactions between applications and garbage collection systems have not been explored much in the literature. Both Java and C# contain an interface for the program to invoke a garbage collection cycle. The use of these functions is usually discouraged. We are unaware of any work that discusses these functions.

An immediate free instruction that can be inserted by the compiler was studied by [CR06], and [GMF06]. They suggested a compiler analysis technique that identifies unreachable objects. The compiler inserts an immediate free instruction for these objects, which reduces the number of garbage collection invocations, and improves efficiency. However, the compiler analysis is conservative. In contrast, an analysis of data structures as in this paper is difficult to perform automatically. Triggering garbage collection cycles when the live space is low is beneficial as there are less objects to trace. [BVEDB07] proposed a triggering scheme with a pre-profiling stage that identifies locations of code where it is favorable to initiate garbage collection.

[AG09] proposed a GC-application interface for evaluating user assertions. This interface allows the user to express assertions about heap layout, which the garbage collection evaluates at runtime. [WF04] proposed another GC-application interface for bounding the memory usage of a child processes. This allows the program to launch untrusted child processes without allocating a different heap for each child and without complicating the data transfer between the processes.

Recently, [RN13] used “freeing hints”. This method relies on deletions of all (or almost all) objects being specified by the programmer in the program. These deletion hints inform the garbage collector that the object should be deleted. However, the garbage collector does not delete the hinted object until it checks that the hints are correct and the hinted objects are unreachable. The authors tested their method on C programs that manually free all objects but contain some memory bugs. They have shown that their collector outperforms standard collectors and provides perfect opportunities for parallel tracing. This method provides protection against memory bugs, which increases application stability. Still, this method requires to identify all locations of object deletion in the program, which is typically difficult for programs in standard managed languages.

Allocating objects based on their type was suggested by [SGBS02]. They considered a

memory manager that places frequently used objects in the young generation. This improved garbage collection locality, eliminated barriers, and improved space efficiency.

Deep heap shapes inherently limit parallelism opportunities. [BP10] found that deep heap shapes do exist in practice. They also proposed two ideas for overcoming such heap shapes. However, one algorithm may create an unbounded amount of floating garbage, and the second algorithm was presented by means of an "all-knowing" oracle, and not as an implementable algorithm. Subsequently, [EP13] discovered that deep heap shapes are mostly caused by stacks or queues. They proposed a lock free queue implementation with low depth via shortcuts. Their solution does not apply to other linked list structures. The DSA collector eliminates the problem of deep heap shapes for all objects in the annotated data structures.

Avoiding mark-stack overflow was considered by [OBYG+02]. They divided the mark-stack into work packets, where every work packet contains a fixed number of objects. This allows the mark-stack to dynamically change its size, and to be dynamically distributed between different threads. It does not eliminate mark-stack overflow in the cases when there is not enough memory for additional work packets. [UIY12] attempted to address mark-stack overflow by improving recovery time. Upon an overflow, they record the set of places where a visited untraced node can reside and use it for faster recovery from the overflow.

Chapter 5

Limitations of Partial Compaction: Towards Practical Bounds

5.1 Background

The theoretical foundations of memory management have not been comprehensively studied. Little is known about the limitations of various memory management functionalities; information about space consumption of various memory management methods in particular is lacking. Previous work consists of Robson's classic results on fragmentation when no compaction is employed [Rob74, Rob71], a result on the hardness of achieving cache consciousness [PR02], and work on the effectiveness of conservative garbage collection and lazy reference counting [Boe02, Boe04]. A recent new work by Bendersky and Petrank. [BP11] attempted to bound the overhead mitigation that can be achieved by partial compaction.

Memory managers typically suffer from fragmentation. Allocation and deallocation of objects in the heap create "holes" between objects that may be too small for future allocation, causing available heap space to be wasted. Compaction can eliminate this problem, but compaction algorithms are notoriously costly [JHM11, AOPS04, KP06a]. Instead, memory managers today either seldom apply full compaction, or employ partial compaction, where only a (small) fraction of the heap objects are compacted to make space for further allocation [BYGK+02, BCR03, CTW05, DFHP04, PPS08].

[BP11] study the limitations of partial compaction. But despite their work being novel and opening a new direction for bounding the effectiveness of partial compaction, their results are only significant for huge heaps and objects. When used in a realistic setting of system parameters today, their lower bounds become meaningless. Suppose, for example, that a program uses a live heap space of 256MB and allocates objects of size at most 1MB. For such a program, even if the memory manager is limited and can only compact 1% of the allocated objects, the results in [BP11] would only imply that the heap must be at least 256MB, which is obvious and not very useful.

In this work we extend the lower bounds on partial compaction to make them meaningful for practical systems. To this end, we propose a new "bad" program that makes memory

managers fail to preserve low space overheads. We then improve the mathematical analysis of the interaction between the bad program and the memory manager to obtain much better bounds. For example, using the parameters mentioned in the previous paragraph, our lower bound implies that a heap of 896MB must be used, i.e., a space overhead factor of 3.5x.

In general, the more objects we let the memory manager move, the lower the space overhead that it may suffer. To put a bound on the amount of compaction work undertaken by the memory manager, we use the model set in [BP11] and bound the compaction by a constant fraction $1/c$ of the total allocated space. This means that at any point in the execution, if a space of s words has been allocated so far, then the total compaction allowed up to this point in the execution is s/c words, where c is the compaction bound.

The results we obtain involve some non-trivial mathematical arguments and the obtained lower bound is presented in a rather complex formula. The general lower bound is stated in Theorem 5.1 (on page 83). But to make clear the improvement over the previously known results, we have depicted in Figure 5.1 (on page 85) the space overhead factor for the parameters mentioned above (which we consider realistic). Namely, for a program that uses live space of $M = 256$MB and whose largest allocated object is at most $n = 1$MB, we drew for different values of c the required heap size as a factor of the 256MB live space. Note that if we were willing to execute full compaction after each deallocation, then the overhead factor would have been 1. We could have used a heap size of 256MB and serve all allocation and deallocation requests. But with limited (partial) compaction, our results show that this is not possible. In Figure 5.1 we drew our lower bound as well as the lower bound obtained from [BP11] for these parameters. In fact, throughout the range of $c = 10, \ldots, 100$, the lower bound from [BP11] gives nothing but the trivial lower bound overhead factor of 1, meaning that 256MB are required to serve the program. In contrast, our new techniques show that the space overhead factor must be at least 2, i.e., 512MB when 10% of the allocated space can be compacted. And when the compaction is limited to 1% of the allocated space, then an overhead factor of 3.5 is required for guaranteeing memory management services for all programs.

In general, the result described in this work is theoretical, and does not provide a new system or algorithm. Instead, it describes a limitation that memory managers can never overcome. As such, it does serve a practical need, by letting practitioners know what they cannot aspire to and should not expend efforts in trying to achieve. The lower bounds we provide are for a worst-case scenario and do not rule out better behavior on a suite of benchmarks. But providing a better guaranteed bound on fragmentation (as required for critical systems such as real-time systems) is not possible. Note that this bound holds for manual memory managers as well as automatic ones, even when applying sophisticated methods such as copying collection, mark-compact, Lang-Dupont, or MC^2 [JHM11].

Finally, we slightly improve the related, state-of-the-art upper bound. The upper bound is shown by presenting a memory manager that keeps fragmentation low against all possible programs. The new upper bound slightly improves the result of [BP11] by providing a better memory manager and a better analysis for its worst space overhead. This slight improvement is depicted in Figure 5.3 (on page 86), where the new upper bound is compared to the previous

upper bound of [BP11]. The upper bound theorem is stated rigorously as Theorem 5.2 (on page 85). The proposed memory manager is not meant to be a practical, efficient memory manager that can be used in real systems, but it demonstrates the ability to deal with worst-case fragmentation.

To achieve the bounds presented in this chapter, we stand on the shoulders of prior work, and in particular our techniques build on and extend the techniques proposed in [Rob71] and [BP11].

5.2 Problem Description and Statement of Results

5.2.1 Framework

We think of an interaction between a program and a memory manager that serves its allocation and deallocation requests as a series of sub-interactions of the form:

1. Allocation: The program requests to allocate objects by specifying their sizes to the memory manager and receiving in response the addresses where the objects were allocated.
2. Deallocation: The program declares objects as free
3. Compaction: The memory manager moves objects in the heap.

These sub-interactions are ordered in time and do not overlap. One sub-interaction starts only after the previous finished. This allows referring to the *time* a sub-interaction takes place. Specifically, the *time* a sub-interaction is executed is defined as the number of sub-interactions preceding it.

In this work, we consider the ability of a memory manager to handle programs within a given heap size. But if a program allocates continuously and never deallocates any memory, then the heap size required is trivially unbounded. Therefore, we assume a bound on the space the program may use simultaneously. This bound is denoted by M.

A second important parameter of the execution of a program (from a memory management point of view) is the size variation of the objects it allocates. If all objects are of fixed size, say 1, a heap space of M is always sufficient. Although holes can be created by deallocating objects, these holes can always be filled by newly allocated objects. If we denote the least size of an object by 1, the parameter n will denote the maximum size of an object. It can be thought of as the ratio between the largest and smallest allowable objects. We denote by $\mathscr{P}(M,n)$ the (infinite) set of all programs that never allocate more than M words simultaneously and allocate objects of size at most n. We denote by $\mathscr{P}_2(M,n)$ the set of programs whose allocated objects sizes are always a power of two.

As explained in the introduction, if the heap is compacted after every deallocation, fragmentation never occurs. However, frequent compaction is costly and so memory managers either perform a full compaction infrequently, or just move a small fraction of the objects occasionally. In this work we adopt the definition of [BP11] and consider memory managers that limit their

compaction efforts by a predetermined fraction of the allocated space. For a constant $c > 1$, a memory manager is c-partial memory manager if it compacts at most $\frac{1}{c}$ of the total space allocated by the program. We denote the set of c-partial memory managers by $\mathscr{A}(c)$.

Given a program P and a memory manager A, the execution of P where A serves as its memory manager is well defined. The total heap size that A uses in this case is denoted by $HS(A,P)$. For simplification, we consider the input to the program as part of the program. This makes it easier to discuss the behavior of a program that interacts with a memory manager without the complication of specifying the input. All results can easily be extended to a model where the input is specified separately from the program.

To present a lower bound on the space overhead required by any memory manager, it is enough to present one bad program whose allocation and deallocation demands would require that all memory managers have a large heap space. For an upper bound, we need to provide a memory manager that would maintain a limited heap space for all possible programs.

Our model is phrased above as one that lets the program know the address of each allocated object. Namely, the program makes decisions dynamically based on the locations in which the allocator places the objects. The knowledge of these addresses helps the program create the fragmented memory. One may wonder whether such a program can operate in a Java-like setting where the program is not given access to addresses of objects. One way to answer that is to note that it is enough for the program to know the the memory manager's algorithm (and GC triggering) in order to compute the addresses of allocated objects by itself. A second answer is that we can view this result as stating that for each possible memory manager there is program that beats it. Using the quantifiers in this manner means that the program has all the information that it needs to know.

5.2.2 Previous work

For programs that allocate only objects whose size is a power of 2, the size of largest object n divides M, and for all memory managers that do not use compaction, [Rob74, Rob71] proved lower and upper bounds that match. For his lower bound, he presented a "bad" program $P_o \in \mathscr{P}_2(M,n)$ that forces any memory manager (that does not use compaction) to have a large heap. Specifically,

$$\min_{A\in\mathscr{A}(\infty)} HS(A,P_o) \geq M\cdot\left(\frac{1}{2}\lg(n)+1\right)-n+1.$$

Note that here, and throughout this chapter, $\lg \triangleq \log_2$. For an upper bound, Robson presented an allocator A_o that satisfies the allocation requests of any program in $\mathscr{P}_2(M,n)$ using a heap size of

$$\max_{P\in\mathscr{P}_2(M,n)} HS(A_o,P) \leq M\cdot\left(\frac{1}{2}\lg(n)+1\right)-n+1.$$

For programs that may allocate objects of arbitrary size (and not only powers of 2), a weaker result is shown:

$$\max_{P\in\mathscr{P}(M,n)} HS(A_o,P) \leq M\cdot(\lg(n)+1). \tag{5.1}$$

The techniques in the proof of [Rob71] actually yield a slightly stronger result for (realistic) cases in which $M >> n$:

$$\max_{P\in\mathscr{P}(M,n)} HS(A_o,P) \leq M\cdot\left(1+\sum_{j=1}^{\lg n}\frac{2^j}{2^j+1}\right)+n(\lg(n)+1).$$

We think of that result as being Robson's best upper bound and show how we improve over that.

Subsequent to the above work, [Rob74] went on and improved the *asymptotic* heap size upper bound to be

$$\max_{P\in\mathscr{P}(M,n)} HS(A_o,P) \leq M\cdot(0.85\lg(n)+O(1)).$$

However, the latter has an asymptotic additive term $O(1)$ that is too large in practice, making it irrelevant for practical parameter settings[1].

When some compaction is allowed (but not an unlimited compaction effort), much less is known. For the upper bound, [BP11] have shown a simple compacting collector $A_c\in\mathscr{A}(c)$, that uses a heap space of at most

$$\max_{P\in\mathscr{P}(M,n)} HS(A_c,P) \leq (c+1)\cdot M$$

words, when run with any program in $\mathscr{P}(M,n)$.

They have also shown a "bad" program P_W that forces all memory managers with a c-partial compaction bound to use a large heap:
$\min_{A\in\mathscr{A}(c)} HS(A,P_W(c)) \geq$

$$\begin{cases} \frac{1}{10}M\cdot\min\left(c,\frac{\lg n}{\lg c+1}-\frac{5n}{M}\right) & \text{for} \quad c\leq 4\lg n \\ \frac{1}{6}M\cdot\frac{\lg n}{\lg\lg n+2}-\frac{n}{2} & \text{for} \quad c>4\lg n \end{cases}.$$

5.2.3 This work

Our main contribution is a new lower bound on the ability of a memory manager to keep the heap de-fragmented. While the lower bound of [BP11] is important for modeling the problem, providing some tools for solving it, and although the authors provide some tools for solving it, this bound is meaningful only for huge objects and heaps. In particular, it is higher than the obvious M only for $M > n \geq 16TB$. In this work we extend the theory enough to obtain meaningful results for practical values of M and n.

Theorem 5.1. *For any c-partial memory manager A and for any $M > n > 1$, there exists a program $P_F\in\mathscr{P}_2(M,n)$ such that for any $\gamma\leq\lg(\frac{3}{4}c):\gamma\in\mathbb{N}$,*

$$\min_{A\in\mathscr{A}(c)} HS(A,P_F) \geq M\cdot h, \tag{5.2}$$

[1] In fact, this constant is at least 30 for any asymptotic improvement over the result of Equation 5.1.

where h is set to:

$$h = \frac{\frac{\gamma+2}{2} - \frac{2^\gamma}{c}\left(\gamma+1-\frac{1}{2}\sum_{i=1}^{\gamma}\frac{i}{2^i-1}\right) + \left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\frac{\lg(n)-2\gamma-1}{\gamma+1} - \frac{2n}{M}}{1+2^{-\gamma}\left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\frac{\lg(n)-2\gamma-1}{\gamma+1}}. \tag{5.3}$$

We remark that the theorem makes use of an integral parameter γ. The theorem holds for any $\gamma \leq \lg(\frac{3}{4}c) : \gamma \in \mathbb{N}$, but obviously there is one γ that makes h the largest and optimizes the bound. Determining this γ mathematically is possible (if we do not require integral values) but the formula is complicated. In practice, however, there are very few (integral) γ values that are relevant for any given setting of the parameters, and so it can be easily computed.

Since h is given in a complicated formula, the implications of $HS(A,P_F) \geq M \cdot h$ are not very intuitive. Therefore, we chose some realistic parameters to check how this bound behaves in practice. We chose M, the size of the allocated live space to be 256MB, and n, the size of the largest allocatable object, to be 1 MB. With these parameters fixed and with the parameter γ set to the value that maximizes the bound, we drew a graph of h as a function of the compaction quota bound c. This graph appears in Figure 5.1. The x-axis has c varying between 10 to 100. Setting $c = 10$ means that we have enough budget to move 10% of the allocated space, whereas setting $c = 100$ means that we have enough budget to move 1% of the allocated space. For these c's, the y-axis represents the obtained lower bound as a multiplier of M. For example, when compaction of 2% of all allocated space is allowed ($c = 50$), any memory manager will need to use a heap size of at least $3.15 \cdot M$. Even with 10% of the allocated space being compacted, a heap size of $2 \cdot M = 512$MB is unavoidable. For these practical parameters, previous results in [Rob74] and in [BP11] do not provide any bound, except for the obvious one, that the heap must be at least of size M.

We also depicted the lower bound as a function of a varying maximum object size n. We fixed the compaction budget to $c = 50$, and the total size of live objects to $M = 256n$. The rationale for the last parameter setting is that a single object rarely makes up a significant part of the heap (larger than half a percent). Setting M to a larger value does not change the bound. We argue below that fixing $c = 50$ (i.e., compacting 2% of the allocated objects) represents a realistic setting. Under stable conditions, where the space occupied by live objects is stable and the heap is partially compacted whenever it is full, one can translate the ratio c of compacted objects over allocated objects into a bound on compaction as a percentage of the live objects space M. As an example, suppose the live space M forms a third of the entire heap space HS and suppose we compact when the heap is exhausted. In that case, $2M$ words are allocated before compaction happens and then compacting 2% of the allocated space $2M$ means compacting 4% of the live space M. Such partial compaction reflects realistic settings. For example, [PPS08] executes partial compaction every 5 garbage collection cycles. In each partial compaction cycle at most 10% of the pages that are less than 50% occupied are compacted. In the setting above, this means compacting 1% of the $3M$ heap size once $2M$ memory is allocated, or $c = 66$. In another system, [BCR03] employs partial compaction and measures the amount of compaction compared to the live size (during a garbage collection cycle). The amount of compaction falls in

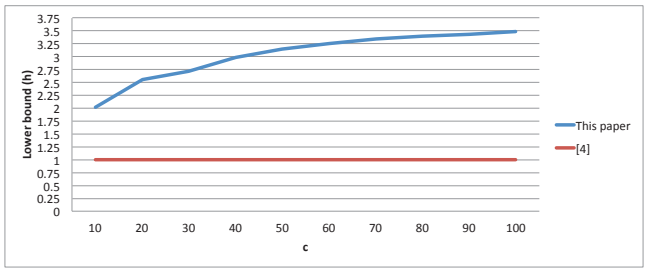

Figure 5.1: Lower bound on the waste factor h for realistic parameters ($M = 256$MB and $n = 1$MB) as a function of c

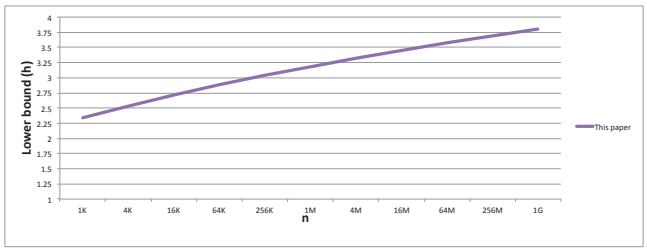

Figure 5.2: Lower bound on the waste factor h as a function of n (c=50, M=256n)

the range of $0.6\% - 4.11\%$. Translating into c values, this correlates to the range of $48 - 300$.

We let the size of the largest object n vary between $1KB$ and $1GB$, and for these n values, the y-axis represents the obtained lower bound as a multiplier of M. The graph is depicted in Figure 5.2. We could also depict the lower bound as a function of M, where n and c remain fixed. However, in a practical setting, the size of the largest object is much smaller than the total live space (i.e., n/M is small). Hence the lower bound as a function of M is very close to a constant function and does not provide additional interesting information.

We also consider the upper bound on the size of heap required. In [BP11], an upper bound of the form $(c+1)M$ was presented. However, this upper bound may become non-interesting when Robson's upper bound is stronger, meaning that the same heap size may be obtained without moving objects at all. This happens when $c > \lg n + 1$. As partial compactors often use a large c to limit the fraction $1/c$ of moved objects, such a scenario seems plausible. We provide some improvement to Robson's algorithm when little compaction is allowed and obtains better upper bound as follows.

Theorem 5.2. *There exists a c-partial memory manager $A \in \mathscr{A}(c)$ that satisfies allocation requests of any program $P \in \mathscr{P}(M,n)$ with heap size at most*

$$\max_{P \in \mathscr{P}(M,n)} HS(A_C, P) \leq M \cdot \left(\sum_{i=0}^{\lg n} a_i \right) + n(\lg(n) + 1),$$

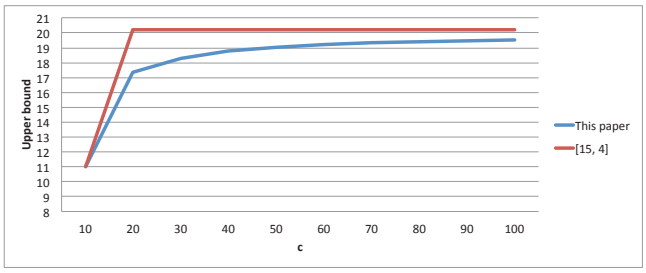

Figure 5.3: Upper bound on the waste factor for realistic parameters ($M = 256$MB and $n = 1$MB) as a function of c

where $a_0 = 1$ *and the values of* a_i, $i = 1, \ldots, \lg(n)$, *satisfy the following recursive formula:*

$$a_i = \frac{2^{i+1}}{2^i + 1} \cdot \left(1 - \sum_{j=0}^{i-1} \max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1} + 1 \rfloor}{2^i}\right) \cdot a_j\right).$$

As the formulas in this theorem are also not easy to grasp, we drew a graph comparing previously known bounds with the new result. It can be seen in Figure 5.3 that for c's between 20 and 100, Theorem 5.2 improves the upper bound, with the largest improvement being 15% at $c = 20$. We consider this result minor and the lower bound the major result in this chapter.

5.2.4 Alternative Budgeting Models

Following previous work [BP11], we budget compaction as a fraction of the total allocation space. The main reason for that is that this model is clean and it is possible to prove strong rigorous theoretical results on it. Practitioners that build garbage collected systems may think of this budgeting as unnatural because, in practice, compaction is run with garbage collection, and the volume of compaction depends on the amount of live objects (because that determines the triggering of a collection) and on the shape of the heap (e.g., freeing memory pages that are sparsely occupied). See for example [BCR03, DFHP04, PPS08]. Let us say a few words about the alternatives.

Perhaps the most natural alternative is to budget allocation as a fraction of the amount of live space. But this alternative is not as clean, it is harder to work with and it poses some problems. First of all, to adopt this budgeting model, one must also specify how often a fraction of the live space can be moved. Triggering of garbage collection may vary in practice because the size of the heap is usually not fixed. But suppose we fix the size of the heap and trigger garbage collection when the heap is exhausted. When do we decide that the heap is full? In practice that would depend on fragmentation. But suppose that we also ignore fragmentation to make the budgeting cleaner. We claim that in this simplified case and under the additional assumption that the amount of live space is stable, the resulting budget is correlated to the one used in this chapter (i.e., a percentage of the allocated objects). To see this, think of the amount of live space as stable and denote this size by M. The space allocated between two garbage collection cycles

becomes fixed in this case: HeapSize (HS) minus M. So after allocating $HS-M$ words, we allow moving a fraction of M. Since we have assumed that HS and M are fixed, then we can write HS as a multiple of M. For example, $HS = 3M$ is common in practice. In this setting, a fraction of M is also a fraction of $HS-M$.

For example, Theorem 5.1 implies that $HS = 3 \cdot M$ is impossible to serve with $c = 50$. Alternatively, this may be phrased as an impossibility to serve a live fraction that is bigger than a third of the heap. In this case, one can think of the memory manager as being restricted from compacting more than $\frac{1}{25} \cdot M$ words after allocating $2 \cdot M$ words. In general, we stick to the budgeting model of [BP11] because it is clean, requires no further assumptions and we can produce meaningful results with it.

It is worthwhile to notice that under the assumptions discussed above, namely that the amount of live space is stable and the heap size is fixed, the ratio between the heap size and M is the live ratio. Thus no allocator can keep the live ratio bigger than $\frac{1}{h} = M/HS$ in all cases. Recall that h is the lower bound presented in Theorem 5.1 whose value is set in Equation 5.3.

5.3 Overview and Intuitions

In this section we review the proof of the lower bound. The main tool in this proof is the presentation of a "bad" program that causes large fragmentation. Our bad program will always allocate objects whose size is an exponent of 2. Furthermore, to simplify the discussion (in this overview) we assume that the memory manager is restricted to aligned allocation. This means that an object of size 2^i is placed at an address divisible by 2^i.

The bad program will work in steps, allocating, in each step, only objects of size 2^i, for some i that it will determine. Consider such a step and consider a memory region that starts at an address divisible by 2^i and spans 2^i words. Denote such a memory region a chunk. If a chunk is fully populated by objects, there is no fragmentation in this chunk. At the other extreme, if a chunk is empty, a new object of size 2^i can be placed in this "hole," creating a fully populated chunk. Fragmentation actually occurs (w.r.t. a chunk) when a chunk is sparsely populated. In this case, utilization of this chunk is low, yet no new object can be placed there. When no compaction is allowed, a "bad" program should attempt to leave a small object in each such chunk, by deallocating as many objects as possible, leaving one object in each chunk.

In Figure 5.4 we illustrate objects that a program deallocates in order to create fragmentation. In each such chunk, we also mark the *density* of allocated space remaining in the chunk after the deallocation. For example, in the third chunk, one out of four equally-sized objects is left in the chunk, making the density of live space 1/4.

The bad program always tries to deallocate as much space as possible, while keeping chunks occupied by objects that hinder their reuse by the allocator. Recall that the program is restricted to allocating at most M words at any time. Therefore, the larger the deallocated space, the larger the space that can be allocated in the next step, and the larger the heap that the allocator must use. This is the main design goal of the bad program: never allow chunk reuse, and allocate as much as possible at each step (by deallocating as much as possible in the previous step).

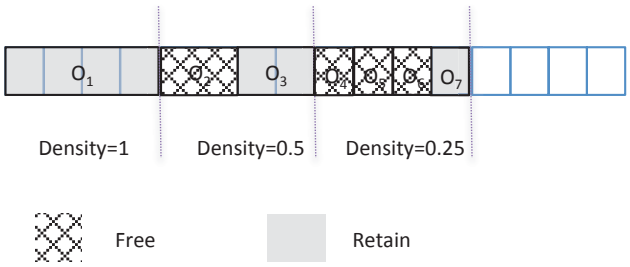

Figure 5.4: Example of different densities

When compaction is allowed, avoiding reuse is more difficult. In particular, a sparsely populated chunk can be evacuated and reused by the memory manager. If the populated space on a chunk is of size $2^i/c$ or smaller, then the memory manager can move the allocated objects away, losing a compaction budget of at most $2^i/c$, but then allocating an object of size 2^i on the cleared chunk and gaining a compaction budget of $2^i/c$. So reuse of sparsely allocated chunks becomes beneficial for the memory manager.

In order to create fragmentation in the presence of compaction, the bad program attempts to maintain dense-enough chunks. If the allocated space for a chunk is, say, $2 \cdot 2^i/c$, then either the memory manager does not reuse this chunk, or it does reuse it, but then must pay at least $2 \cdot 2^i/c$ of its compaction budget. Allocation of a new object recharges the compaction budget by $2^i/c$; thus the memory manager remains with a minus of $2^i/c$ words. This allows bounding the overall reuse by bounding the overall compaction budget.

Finally, let us discuss how we deal with objects that are not aligned and thus reside on the border of two chunks. If objects are aligned, then an object is allocated exactly on one full single chunk. In order to allocate an object, the chunk it is put on must be entirely free of objects. When an object's allocation is not aligned, it may reside on two chunks. We start by looking at smaller chunks, whose size is one-quarter of the allocated object. This means that a non-aligned allocated object must entirely fill three chunks (and partially sit on two more chunks). These three chunks must be completely free of allocated objects before the allocation can take place. If one of these three chunks is about to be reused, then the memory manager compact away every object that resides (even partially) on the reused chunk, and lose some compaction budget. Note that we have to make sure that if two adjacent chunks are reused, we do not double count budget loss due to the moving of a non-aligned object that resides on both.

5.3.1 Improvements over prior work [BP11]

In this subsection we give an overview of the three main improvements of this work over [BP11]. These improvements allowed us to achieve a better lower bound, meaningful with realistic parameter settings.

The first improvement follows from noting that, in the first steps, the allocated objects are

large while chunk sizes are small. Compaction is of no benefit to the memory manager in this case. Therefore, in these first steps, we run a program that is very similar to Robson's [1974] bad program but adapted to this case. We develop a reduction theorem to show that this program creates fragmentation even when compaction is allowed. Furthermore, these first steps nicely integrate with the algorithm that runs in the second stage during the rest of the steps. This improvement of using Robson's simpler algorithm for the first stage cannot be extended to the second stage. When moving into later steps where the chunk sizes get large, it becomes possible for the memory manager to gain substantially from moving objects. Moving a relatively small object and gaining a free chunk for further allocations is highly beneficial for the memory manager in order to be able to work with a smaller heap. Robson's bad program is not appropriate to be used in such a setting, because it assumes no compaction happens. Therefore, we go back to using the ideas from [BP11] and use a program similar to theirs in the second stage. This program attempts to keep the density of objects high in each chunk. When the density of allocated objects is high enough in a chunk, too much space needs to be moved from a chunk in order to allow allocating on it again.

The second improvement consists of a small twist in the algorithm that creates more regimented behavior during the execution, making it possible to analyze the execution and obtain the improved bound. Recall that the bad program attempts to deallocate as much space as possible in each step so that it can allocate as many objects as possible in the next step. It turns out that this behavior can create scenarios that are hard to analyze. This happens when we allocate a lot of objects in one step, and then very few in the following steps. In our algorithm, we bound the memory allocated per step. This, perhaps, does not use all the space that can be used in one step, but it guarantees sufficient allocation in all steps.

Finally, the third improvement concerns non-aligned object allocation. When an object is not aligned, it consists of two parts that lie on two different chunks. The reuse of one of these chunks for allocation requires moving this object, but such a move may also allow use of the other neighboring chunk. The analysis of these scenarios is not simple. We gain more control over the analysis by virtually assigning the non-aligned object to one of its underlying chunks. Of course, in order to reuse a chunk, this object must still be moved, but the virtual assignment of an object to one of its underlying chunk allows for easier algorithmic decisions and also computing tighter bounds on the amount of reuse that the memory manager can achieve.

5.4 Lower bound: creating fragmentation

In this section we prove a lower bound on the ability of a memory manager to keep the heap defragmented when its compaction resources are bounded. In particular, we introduce a program P_F that forces any c-partial memory manager to use a large heap size to satisfy all of P_F's allocation requests.

Let us start by explaining the ideas behind the construction of P_F. The program P_F works in two stages. The second stage is probably the major contributor to the fragmentation, but the first stage is also necessary to obtain our results. The first stage is an adaptation of Robson's

malicious program [Rob74], which attempts to fragment the memory as much as possible, when working against a memory manager that cannot move objects. We will discuss in Section 5.4.2 how this algorithm behaves against a memory manager that can move objects and show that it buys some fragmentation in this case as well. After running for 2γ steps, a second stage starts, which behaves differently. The second stage works in steps $i = 2\gamma, \ldots, \lg(n) - 2$ and in each step it only requests allocations of objects of size 2^{i+2} words. At each such step of the execution, we consider a partition of the heap space into aligned chunks of size 2^i words. This means, for example, that each allocated object either consumes four full consecutive chunks if its allocation is aligned, or it consumes at least three full consecutive chunks.

Our goal is to show that the memory manager must use many chunks. If at any point in the execution, $x+1$ chunks of size 2^j are used, then the heap must contain at least x chunks even if only one word of each chunk is used. (The last chunk may not be entirely contained in the heap.) This means that the memory manager must use a heap size of at least $x \cdot 2^j$ words.

Since we do not assume aligned allocation, objects may spread over more than one chunk. Nevertheless, each chunk that has a word allocated on it (at any point of the execution) must be part of the heap. Given an execution of the program P_F with some given memory manager, we associate with each chunk a set of objects that were allocated on it at some point in the execution. This enables tighter analysis. An object is associated with one of the chunks it resides on. This means that at least one word of the object resides on the associated chunk at the time the object is allocated. We then aim to show that $x+1$ chunks have objects associated with them, and obtain the bound as above.

The association of objects with chunks is chosen carefully to establish the bound. Note that chunk sizes dynamically change as we move from step to step. On a step change, each pair of adjacent chunks becomes a single joint chunk. So the association of chunks with objects changes between steps. The association is also updated during the execution, as objects get allocated and deallocated. The program actively maintains the set of objects associated with each chunk, and also uses this set in the second stage to determine which objects to deallocate. An object is only de-associated from a chunk when it is deallocated by P_F. It is not de-associated when an object is compacted. Note that when an object is moved, it is usually possible to put it in partially used chunks that are not fully occupied. Since the analysis gains nothing from checking the object's new location, we just let the bad program immediately deallocate any object that is being moved. In other words, rather than attempting to consider the location to which it was moved we simply deallocate it and use the reclaimed space for future allocations. Note that the chunk that it did occupy will remain part of the heap forever, so associating it with the old chunk as evidence that that chunk has been used is always fine.

We denote by $O_D(t)$ the set of objects that P_F associates with chunk D at time t, and we sometimes omit the t, when its value is clear from the context. When an object lies on the border of two chunks, we sometimes choose to associate it with both. In this case, we associate exactly half of it with each of the chunks (ignoring the actual way the object is split between them). This even split in association is used to preserve the property that object sizes (and association sizes) are a power of two. This refinement of association implies that a chunk may be associated

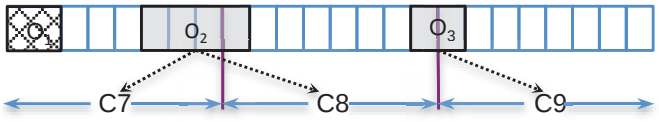

Figure 5.5: Association of objects and half objects with chunks. O_3 is associated with Chunk C_9 (only), so its density is 0.25, even though only 1 of the 8 words of O_3 is covered. In addition, half of O_2 is associated with C_8 even though only a quarter actually intersects C_8. The other half is associated with C_7. This makes the densities of C_7, C_8 and C_9 0.25.

with half an object, and a single object may be associated with two chunks.

From the memory manager point of view, a chunk that contains a small number of allocated words is a good candidate for compaction and reuse. Compaction allows reuse of a chunk's space for more allocations. As the program P_F controls which objects get deallocated, P_F will attempt to ensure that each chunk has enough associated objects to make it non-beneficial for the memory manager to clear a chunk. The *density* of a chunk is the total size of objects associated with it, divided by the chunk size. We define a *density parameter* $2^{-\gamma}$ so that P_F attempts to keep the density of a chunk at least $2^{-\gamma}$. This means that the program P_F will never choose to deallocate an object if doing so makes a chunk too sparse, in the sense that its density goes below $2^{-\gamma}$. This density will be chosen to be larger than $1/c$ to make the compaction of objects from such a chunk non-beneficial, as explained next.

Loosely speaking, according to the compaction budget rules, when the memory manager allocates an object o of size $|o|$ it gains an extra compaction budget of $\frac{1}{c}|o|$. However, if it needs to move $2^{-\gamma} \cdot |o|$ words to make space for this allocation, then its overall compaction budget decreases. (Recall that $2^{-\gamma} > \frac{1}{c}$.)

An example of density threshold and association set is depicted in Figure 5.5. Let the density threshold $2^{-\gamma}$ be $1/4$, which consists of 2 words per chunk of size 8. Half of O_2 is associated with Chunk $C7$, the other half is associated with Chunk $C8$, and the object O_3 is associated with Chunk $C9$ only. These objects suffice to make the density of each chunk at least $1/4$. The program can free the object O_1 since a density of $1/4$ is preserved even without it. In this example, we actually count the associated space of an object rather than the actual space the object occupies on a chunk. When clearing a chunk, an entire object must be moved, even if only one word of it resides on the chunk. For the sake of the analysis, we count half of this moving cost for each of the object's chunks. If we move the object, we pay for the compaction of its entire size, but both chunks gain space for allocation.

For the first steps in the computation, maintaining a high enough density is irrelevant: chunks are small enough so that when even one word is allocated on the chunk, the density is $2^{-\gamma}$. Therefore, it is never useful for the memory manager to compact, and we can simply adopt Robson's "bad" program [Rob74] with a technical variation that enables it to deal with compaction. While compaction is not beneficial to the memory manager, it may still occur and our bad program must deal with it, even though Robson's original program does not. We

build a program P_F that will be "similar" to Robson's program in the sense that it will keep a similar heap shape, it will make very similar decisions on which objects to deallocate and it will allocate the same amount of space in each step. We will then present a reduction showing that if there exists a memory manager M that can maintain low fragmentation while serving P_F with bounded compaction, it is possible to create another memory manager M' that does not move objects and maintains low fragmentation against Robson's program. Since no memory manager can keep low fragmentation against Robson's program, we get the lower bound we need.

When objects are moved by the memory manager, the simulation of the original interaction of Robson's program with its memory manager cannot simply proceed, because Robson's program does not handle movements of objects. Several new factors might influence the execution. First, the space the object is moved to becomes occupied, so new objects can no longer be allocated there. Second, vacancy is created in the old space from which an object was moved, so objects might be allocated there. And finally, the different shape of allocated objects may change the deallocation decisions of the bad program. To handle newly occupied space, P_F simply deallocates each moved object immediately after it gets moved. This ensures that new objects can be allocated as before. To handle the other two problems, we introduce *ghost objects*, which are not really in the heap but are used by P_F to remember where objects existed so that Robson's program behavior can still be imitated. The need to simulate Robson's program only occurs in the first stage of P_F. After that, a different algorithm is used, and ghost objects are not needed anymore.

The program P_F maintains a list of ghost objects along with their original locations. These are objects that have been relocated by the memory manager. For all of its deallocation considerations, P_F makes the same object deallocation decisions as Robson's original program, except that it treats ghost objects as if they still reside at their original location. In fact, each ghost object continues to exist until the original deallocation procedure of Robson's program determines that it should be deallocated. Of course, these objects were deallocated in the execution of P_F when they became ghosts, so no actual deallocation is required in the current execution. Therefore, when Robson's program deallocates them, these objects are simply removed from the list of ghost objects and are not considered further by the deallocation procedure. A diagram presenting object states is provided in Figure 5.6.

Note that memory space on which ghost objects reside may be used to allocate new objects by the memory manager, which is not aware of the ghost objects. This is fine. The deallocation procedure of P_F can view both objects as residing at the same location while making its decisions. This seeming collision is later resolved in the reduction theorem, by means of a property of the deallocation procedure: the procedure is concerned only with the location of objects modulo 2^i. Therefore, one can think of the ghost objects as existing in separate spaces at the same address modulo 2^i. This additional space used for the ghost objects will not be counted as part of the heap size, but it will allow the program P_F and a real memory manager to work consistently together. Details follow.

Definition 5.4.1. [A ghost object] We call an object that was compacted by the memory

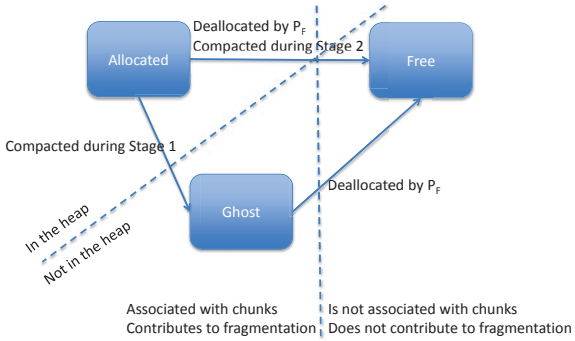

Figure 5.6: Ghost objects. When objects are moved by the memory manager at Stage I, they become ghosts, until they are de-allocated by Stage I of P_F that simulates the behavior of Robson's algorithm. In Stage 2, moved objects are deallocated immediately by P_F and so ghost objects are not required.

manager during the execution and immediately deallocated by the program P_F a *ghost* object. In the first stage of the algorithm, such objects are considered by P_F as still residing as ghosts in the original location where they were allocated. They do not affect the behavior of the memory manager, which can allocate objects to a space occupied by ghosts. When ghost objects are deallocated by the program, they disappear and are no longer considered by P_F in subsequent steps.

At each step i of the first stage, $i = 0, 1, \ldots, \gamma$, the program P_F starts by considering a partition $\mathscr{D}(i)$ of the heap into all aligned chunks of size 2^i. (Aligned here means that they start on an address that is divisible by 2^i.) The main decision that P_F makes at each step is which objects should be deallocated (by the malicious program). To this end, Robson's original program picks an offset f_i, $0 \leq f_i \leq 2^i - 1$, and examines the word at offset f_i (from the beginning of the chunk) for all chunks.

Deallocation then is executed for all objects that do not intersect the f_i word of a chunk. Note that all objects that will be allocated thereafter are all of size at least 2^i, and therefore, two adjacent chunks that have their f_i offset word occupied will never be able to hold a new object between them, even when the object is not aligned.

Definition 5.4.2. [an f-occupying object with respect to step i] For each step i during the execution of P_F and an integer f satisfying $0 \leq f < 2^i$, an object is f-occupying with respect to step i if it occupies a word at address $k \cdot 2^i + f$ for some $k \in \mathbb{N}$.

As a new step kicks in, the chunk size is doubled from 2^i to 2^{i+1}, where each chunk contains two adjacent chunks of the previous step i. P_F needs to pick a new offset f_{i+1} for the larger chunks of size 2^{i+1}, i.e., $0 \leq f_{i+1} \leq 2^{i+1}$. The new offset will be either the old offset on the

left 2^i-sized sub-chunk, i.e., $f_{i+1} = f_i$ or the old offset on the right 2^i-sized sub-chunk, i.e., $f_{i+1} = f_i + 2^i$. Robson chooses the new offset in a way that washes the most space. In a way, Robson attempts to keep the smallest objects that will still occupy words at the f_{i+1} offset. So if one of these two offsets allows capturing more space with smaller objects, this becomes the new offset f_{i+1}. To formalize this, Robson chooses f_{i+1} to be either f_i or $f_i + 2^i$, according to which one maximizes

$$\sum_{\text{o is } f_{i+1}\text{-occupying}} 2^{i+1} - |o|.$$

It is not necessary to understand the details of Robson's analysis, as we adopt it with no modification for the first stage of our program.

After running Robson's program, the program P_F sets the step number i to 2γ, as if skipping $\gamma - 1$ steps. This has the effect of increasing the chunk size, and therefore making all objects in the heap of size at most $2^{-\gamma}$ of the chunk size 2^i. As will be shown in the analysis, having small objects allows good control over the deallocation, and helps ensure that a lot of space can be deallocated by P_F even while maintaining a density of $2^{-\gamma}$. With much space deallocated, P_F gains ammunition for allocations in the steps of the second stage. Recall that P_F is limited and cannot allocate more than M words simultaneously, so it must deallocate enough space before it can allocate again.

The bad program P_F is presented in Algorithm 5.1 (on page 96). Recall that we denote the density that the program attempts to maintain in each chunk by $2^{-\gamma}$. Other inputs to P_F include M, n, which is the size of the largest allocatable object, and c, the compaction budget factor. A complete list of parameters appears in Table 5.1.

P_F starts by running some steps that are similar to Robson's algorithm and proceeds with the newly designed algorithm to deal with compaction and maintain some density in each chunk. When the memory manager moves objects using its compaction quota, the program will not try to take advantage of the moved objects in their new location. There are not enough of those to justify the trouble. Instead, it will simply delete these objects immediately and use the reclaimed space for future allocation.

5.4.1 Analysis of Program P_F

We first claim that P_F always terminates. The only loop that is not strictly bounded is the while loop at Line 17. To see that this loop terminates we need to see that the set *ChunksToHandle* eventually becomes empty. The size of this (finite) set decreases in each iteration by one, except when Line 27 is executed, in which case the set size is incremented by one. However, the number of times Line 27 is executed is bounded. Line 27 reduces the number of half-objects associated with chunks by 2 (because it combines two half objects into a single full object). Since the number of half objects is finite, the number of times Line 27 can be executed is bounded and therefore so is the while loop.

Let us now analyze the behavior of the program P_F when executing against a c-partial memory manager A. We distinguish the behavior of the program in the two stages. We denote

Table 5.1: Table of Notations

$HS(A,P)$	$\triangleq$	the heap size used by allocator A when servicing program P.
M	$\triangleq$	the maximum size of live objects
n	$\triangleq$	the maximum size of an object
c	$\triangleq$	the compaction bound
$\|o\|$	$\triangleq$	the size of an object o
P_F	$\triangleq$	the bad program; Algorithm 5.1.
A chunk at Step i	$\triangleq$	the set of memory words $\left[k\cdot 2^i,(k+1)\cdot 2^i\right)$ for some integral k
$2^{-\gamma}$	$\triangleq$	The desired density by P_F
h	$\triangleq$	the memory overhead factor. Its value is defined in Theorem 5.1
s_i	$\triangleq$	the total size of objects allocated at Stage i ($i=1,2$)
q_i	$\triangleq$	the total size of objects compacted at Stage i ($i=1,2$)
$u_D(t)$	$\triangleq$	the potential of chunk D at time t. See Definition 5.4.3
$u(t)$	$\triangleq$	the potential function at time t. A lower bound on the heap size. See Definition 5.4.4
		Terms used in Stage 1
P_R	$\triangleq$	Robson's bad program; Algorithm 5.2.
Ghost objects	$\triangleq$	objects compacted during the first stage of P_F and treated specially by P_F. See Definition 5.4.1
a f_i-occupying object	$\triangleq$	an object intersecting with word $k\cdot 2^i+f_i$ for some integral k. Used by Robson's bad program. See Definition 5.4.2
		Terms used in Stage 2
O_D	$\triangleq$	a set of objects associated with a chunk D. O_D contains a subset of the objects that intersect D. A chunk cannot be reused for allocation unless the objects in O_D are compacted. See Definition 5.4.12
$\mathscr{E}$	$\triangleq$	when three chunks are covered by 2 half objects, the middle chunk is in $\mathscr{E}$. This middle chunk is covered, even though no specific object is associated with it. See Definition 5.4.13
$q(o)$	$\triangleq$	objects compacted to location o. See Definition 5.4.15.
$x\cdot M$	$\triangleq$	the total size of memory allocated at each step of P_F's second stage. The value of x is computed at the beginning of P_F's execution.

Algorithm 5.1 Program P_F

Input: M,n,c,γ

Initially: Compute $x = \frac{1-2^{-\gamma}\cdot h}{\gamma+1}$

During the execution: If the memory manager compacts an object, ask the memory manager to deallocate this object immediately (before any other action is taken), but add this object to the set of ghost objects, with the same address it held when it was allocated.

{**//Stage I:**}

$f_0 := 0$

Allocate as many objects of size 1 as is possible (i.e., M such objects.)

for $i = 1$ to γ **do**

  Pick f_i to be either f_{i-1} or $f_{i-1} + 2^{i-1}$, according to which of the two maximizes

$$\sum_{(o \text{ is live or ghost}) \text{ and } o \text{ is } f_i\text{-occupying}} 2^i - |o|$$

  Free every live or ghost object that is non f_i-occupying

  Allocate $\left\lfloor \left(M - \sum_{o \text{ is live or ghost}} |o| \right) / 2^i \right\rfloor$ objects of size 2^i

end for

Associate objects with chunks: consider the chunk partition $\mathscr{D}(2\gamma-1)$ to chunks of size $2^{2\gamma-1}$. Each f_γ-occupying object is associated with the chunk that contains its f_γ-occupying word.

{**// Stage II:** }

for $i = 2\gamma$ to $\lg(n) - 2$ **do**

  Consider the chunk partition $\mathscr{D}(i)$ of chunks of size 2^i. Each chunk D is composed of
    chunks D_1, D_2 of the previous step, we set the association: $O_D = O_{D_1} \cup O_{D_2}$

  ChunksToHandle := $\mathscr{D}(i)$.

  while ChunksToHanle $\neq \emptyset$ **do**

    Pick $D \in$ ChunksToHandle

    ChunksToHandle := ChunksToHandle $\setminus D$

    Pick a maximal set $X \subset O_D$ such that $\sum_{o \in O_D \setminus X} |o| \geq 2^{i-\gamma}$.

    $O_D := O_D \setminus X$

    for all $o \in X$ **do**

      if o is half an object **then**

        Let o' be the other half of o, let o_{full} be the entire object, and let D' be the chunk
          where o' resides.

        Set $O_{D'} := O_{D'} \setminus \{o'\} \cup \{o_{full}\}$

        ChunksToHandle := ChunksToHandle $\cup D'$.

      else

        Free o

      end if

    end for

  end while

  Allocate $\lfloor x \cdot M \cdot 2^{-i-2} \rfloor$ objects of size 2^{i+2} but not exceeding the bound M
    on the total size of allocated memory.

  Each allocated object o fully covers 3 chunks D_1, D_2, D_3, if it covers four, pick the
    first three.

  Let o' and o'' be the first and second halves of O, and set
    $O_{D_1} := \{o'\}, O_{D_2} := \emptyset, O_{D_3} := \{o''\}$.

end for

the set of objects that P_F allocates during the first stage by S_1 and during the second stage by S_2. We denote the total size of the objects in S_1 by s_1 and the total size of the objects in S_2 by s_2. Finally, we denote the set of objects that the memory manager chooses to compact during the first stage by Q_1 and their total size by q_1. Similarly, the corresponding set of compacted objects in the second stage is Q_2, the accumulated size of which is q_2.

The execution of P_F proceeds in steps $i = 0, 1, \ldots, 2\gamma-1, 2\gamma, \ldots, \lg(n)-2$. The steps $0, 1, \ldots, 2\gamma-1$ define the first stage (however nothing is done in steps $\gamma+1, \ldots, 2\gamma-1$). The rest of the steps are executed in the second stage. At each first stage step, we use a partition of the heap into chunks of size 2^i, in an aligned manner, i.e., each chunk starts at an address that is divisible by 2^i. We denote by $\mathscr{D}(i)$ the set of all aligned chunks of size 2^i.

Our analysis is simplified by using a potential function $u(t)$, which we define next. We will show that this function represents a lower bound on the amount of memory used during the execution, and thus it is also a lower bound on the heap size. Our goal will be to show that the potential function becomes large by the end of the execution. The function $u(t)$ will be written as a sum of chunk functions $u_D(t)$, one for each chunk in $\mathscr{D}(i)$. The function $u_D(t)$ will be zero for all chunks that have never been used to allocate objects. On the other hand, $u_D(t)$ will always be at most 2^i (which is the size of the chunk D at time t) for chunks that have been used until time t.

During the analysis of the second stage, we will need to give special treatment to some of the chunks. The set of special chunks will be denoted by $\mathscr{E}$ and defined later in Definition 5.4.13. For the analysis of the first stage, one can simply think of $\mathscr{E}$ as the empty set. Let us now set the terminology and then define the potential function.

Definition 5.4.3. [The chunk function $u_D(t)$] Let A be a c-partial memory manager, and let t be any time during the execution of P_F against A in step i. Let D be a chunk of size 2^i. The function $u_D(t)$ is defined as follows.

$$u_D(t) = \begin{cases} 2^i & D \in \mathscr{E}(t) \\ \min(2^\gamma \cdot \sum_{o \in O_D(t)} |o|, 2^i) & \text{otherwise} \end{cases}.$$

Recall that $|o|$ is the size of an object o. The above definition depends on the association of objects to the chunk D, as determined by the association function $O_D(t)$. This association is computed explicitly by the program P_F and it dynamically changes during the execution. Let us now define the potential function $u(t)$.

Definition 5.4.4. [The potential function $u(t)$] Let A be a c-partial memory manager, and let t be any time during the execution of P_F against A. Let i be the step in which t occurs. The function $u(t)$ is defined as follows.

$$u(t) = \left(\sum_{D \in \mathscr{D}(i)} u_D(t) \right) - \frac{n}{4}.$$

Let us explain the intuition behind the potential function as a sum of the $u_D(t)$ items. The malicious program P_F attempts to keep the density of each chunk to at least $2^{-\gamma}$. If it succeeded

for a chunk D at time t, we consider D to be under the control of P_F; thus, D's size (2^i) is added to the potential function. If, however, the density of the chunk is smaller than $2^{-\gamma}$, i.e., the space consumed is $\mu \cdot 2^i < 2^{-\gamma} \cdot 2^i$, then we add $2^i \cdot \mu / 2^{-\gamma}$ to the potential function. This estimates the success of P_F in controlling this chunk as a fraction $\mu / 2^{-\gamma}$ of its length. Intuitively, P_F's target is to increase the amount of memory under its control, which reflects increasing the potential function.

Let us explain why the function $u(t)$ is a lower bound on the number of words in the heap. It will later be shown that, for any chunk D, $u_D(t)$ is non-zero only if there exists an object o which intersected with D at some point during the execution. We consider the heap to be the smallest consecutive space that the memory manager may use to satisfy all allocation requests. Now, if $x+1$ chunks of size 2^i are used during the execution, then at least x of them (all but the last chunk) must fully reside in the heap. Thus, the heap size must be at least $x \cdot 2^i$. As $u(t)$ sums over all used chunks, and as it accumulates at most 2^i for each, we get that $u(t)$ is a lower bound on the heap size. A caveat to that is that $u(t)$ may accumulate 2^i also for the last chunk that is not fully used in the heap. It is for this reason that $u(t)$ is defined as the sum of all $u_D(t)$ minus a single $n/4$; recall that $n/4$ is the largest 2^i possible. With this additional term, $u(t)$ is guaranteed to be a lower bound on the size of the heap used. Next, we analyze the increase of $u(t)$ during the execution.

When P_F allocates an object, the memory manager either places it on completely new chunks, which (as will be shown) increase the value of $u(t)$, or places it on a chunk already occupied by other objects that have been compacted away. As compaction is bounded, the latter will infrequently occur, and furthermore, it will be shown that such a combination of compaction and allocation does not decrease $u(t)$. It will also be shown that a step change that influences $u(t)$ does not decrease it. Finally, new objects may be placed on top of objects that were deallocated earlier by P_F. But the program P_F will manage its deallocations to not allow reuse of a chunk unless some objects are compacted away from it. Thus, we get that the function grows sufficiently to provide a good lower bound on the heap size.

The guaranteed growth of $u(t)$ and the implied lower bound are shown in two lemmas, 5.4.5 and 5.4.6. Lemma 5.4.5 asserts the increase of $u(t)$ during the first stage and bounds from above the amount of space allocated during this stage. This bound will be used to prove the theorem. Recall that q_i (defined in the beginning of Subsection 5.4.1) represents the amount of memory compacted at stage i and s_i represents the amount of memory allocated at stage i.

Lemma 5.4.5. *Let A be a c-partial memory manager, and let t_{first} be the time that P_F finishes the execution of its first stage when executing with A as its memory manager. Then*

$$u(t_{first}) \geq M \cdot \frac{\gamma+2}{2} - 2^{\gamma} \cdot q_1 - \frac{n}{4}.$$

Moreover, the total size of allocated memory during the execution of the first stage s_1 is bounded by

$$s_1 \leq M \left(\gamma + 1 - \frac{1}{2} \sum_{i=1}^{\gamma} \frac{i}{2^i - 1} \right).$$

The analysis of the second stage is summarized in Lemma 5.4.6. This lemma again asserts that $u(t)$ increases. However, the increase depends on the total space allocated in the second stage and also on the compaction budget in the second stage, which depends on the space allocated in both stages. Note that the allocator may have compaction budget left from Stage 1, so the amount of compaction at Stage 2 cannot be bounded as a fraction of allocation at Stage 2 only. To show that the increase in the potential function $u(t)$ is high, this lemma also bounds from below the amount of allocated space s_2 in the second stage. This second bound uses an additional parameter h, which depends on γ, c, n, and M and is set to the following complicated expression in order to achieve the strongest possible bound.

$$h = \frac{\frac{\gamma+2}{2} - \frac{2^\gamma}{c}\left(\gamma+1-\frac{1}{2}\sum_{i=1}^{\gamma}\frac{i}{2^i-1}\right) + \left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\frac{\lg(n)-2\gamma-1}{\gamma+1} - \frac{2n}{M}}{1+2^{-\gamma}\left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\frac{\lg(n)-2\gamma-1}{\gamma+1}}$$

The parameter h is the wasted space factor; recall that M is a bound on the live space allocated simultaneously by P_F, and $h \cdot M$ is the lower bound we show on the size of the heap that the memory manager must use to satisfy the allocation requests of P_F. The analysis will show that either the memory manager uses more than $M \cdot h$ space, in which case we are done with the proof of Theorem 5.1, or the program allocates a lot of space, as in the second part of the lemma, which will then be used to show that the heap space used in both stages is larger than $M \cdot h$, satisfying the assertion of Theorem 5.1.

Lemma 5.4.6. *Let A be a c-partial memory manager, and let t_{finish} be the time that P_F finishes its execution with A as its memory manager. Then,*

$$u(t_{finish}) - u(t_{first}) \geq \frac{3}{4}s_2 - 2^\gamma \cdot q_2.$$

Additionally, either the memory manager uses more than $M \cdot h$ space, or the accumulated size s_2 of space allocation in the second stage satisfies

$$s_2 \geq M\left(\frac{\lg(n)-2\gamma-1}{\gamma+1}\right)\left(1-2^{-\gamma}\cdot h\right) - 2n.$$

We now show how to obtain the lower bound stated in Theorem 5.1 using Lemma 5.4.5 and 5.4.6. Because the memory manager compacts at most $\frac{1}{c}$ of the total allocation, we know that $(q_1+q_2) \leq \frac{1}{c}(s_1+s_2)$. Thus,

$$\begin{aligned}
HS(A,P_F) &\geq u(t_{finish}) = u(t_{first}) + (u(t_{finish}) - u(t_{first})) \\
&\geq M \cdot \frac{\gamma+2}{2} + \frac{3}{4}s_2 - 2^\gamma \cdot (q_1+q_2) - \frac{n}{4} \\
&\geq M \cdot \frac{\gamma+2}{2} + \frac{3}{4}s_2 - \frac{2^\gamma}{c}(s_1+s_2) - \frac{n}{4} \\
&\geq M \cdot \frac{\gamma+2}{2} - \frac{2^\gamma}{c}s_1 + \left(\frac{3}{4}-\frac{2^\gamma}{c}\right)s_2 - \frac{n}{4}.
\end{aligned}$$

Now, if $HS(A,P_F) \geq M \cdot h$, then we are done. Otherwise, Lemma 5.4.6 gives us a lower bound

on s_2. The lower bound can be used here since the assumptions of Theorem 5.1 have $\gamma \leq \lg\left(\frac{3}{4}c\right)$ and so s_2 appears in a positive term in the inequality for $HS(A,P_F)$. Therefore, we use the inequality

$$s_2 \geq M\left(\frac{\lg(n)-2\gamma-1}{\gamma+1}\right)\left(1-2^{-\gamma}\cdot h\right)-2n.$$

In addition, Lemma 5.4.5 implies

$$s_1 \leq M\left(\gamma+1-\frac{1}{2}\sum_{i=1}^{\gamma}\frac{i}{2^i-1}\right).$$

In order to make the formula short and accessible, let us denote $\tau = \frac{\lg(n)-2\gamma-1}{\gamma+1}$ (which is one of the terms in Lemma 5.4.6) and $\ell = \gamma+1-\frac{1}{2}\sum_{i=1}^{\gamma}\frac{i}{2^i-1}$ (the term in Lemma 5.4.5). Now, substituting τ,ℓ in the above expressions we may write:

$$s_2 \geq M\cdot\tau\left(1-2^{-\gamma}\cdot h\right)-2n$$

(using Lemma 5.4.6);

$$s_1 \leq M\cdot\ell$$

(using Lemma 5.4.5);

$$h = \frac{\frac{\gamma+2}{2}-\frac{2^\gamma}{c}\cdot\ell+\left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\cdot\tau-\frac{2n}{M}}{1+\left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\cdot\tau\cdot 2^{-\gamma}}$$

and

(using Equation 5.3).

By our assumption $M\cdot h > HS(A,P_F)$,

$$M\cdot h > HS(A,P_F) \geq M\cdot\frac{\gamma+2}{2}-\frac{2^\gamma}{c}\cdot M\cdot\ell+\left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\cdot M\cdot\tau\left(1-2^{-\gamma}\cdot h\right)-2n.$$

Dividing the inequality by M and gathering multiplications of h on the left side we get

$$h+h\left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\tau\cdot 2^{-\gamma} > \frac{\gamma+2}{2}-\frac{2^\gamma}{c}\cdot\ell+\left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\tau-\frac{2n}{M},$$

and after dividing each side of the inequality by the common multiplier of h, we get

$$h > \frac{\frac{\gamma+2}{2}-\frac{2^\gamma}{c}\cdot\ell+\left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\cdot\tau-\frac{2n}{M}}{1+\left(\frac{3}{4}-\frac{2^\gamma}{c}\right)\cdot\tau\cdot 2^{-\gamma}} = h,$$

a contradiction. Thus, the assumption that $HS(A,P_F) < M\cdot h$ cannot hold, and we are done with the proof of Theorem 5.1.

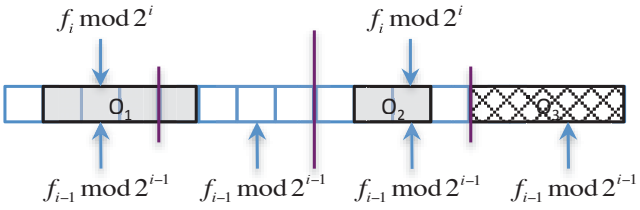

Figure 5.7: Illustrating f_i-occupying objects. O_3 is f_{i-1}-occupying but not f_i-occupying. Therefore, the object O_3 will be freed in Line 4 of Algorithm 5.2.

5.4.2 Analysis of the first stage

We now focus our attention on the first stage and prove Lemma 5.4.5. Consider an execution of the malicious program P_F with any memory manager A and look at the first γ steps. In Step i, the size of the chunks is 2^i words, and therefore the size of any chunk throughout the first stage is not larger than 2^γ words. If any object is associated with a chunk, and since any object's size is at least one word, then the fraction (or density) of live space associated with that chunk must be at least $2^{-\gamma}$. When the density is guaranteed to be that high and compaction is limited by the $1/c < 2^{-\gamma}$ fraction, it is of little benefit to the memory manager. Therefore, for these initial steps (of the first stage), we chose to use a program that is very similar to Robson's program for maximizing fragmentation when no compaction is allowed. We will analyze the slightly modified program to show that it is still useful when limited compaction is used by the memory manager.

For completeness, let us recall Robson's program in Algorithm 5.2. In the algorithm we use the term *f-occupying objects* that was defined in Definition 5.4.2. Also, an object is *live* if it is in the heap, i.e., has not been deallocated. A simple example of the behavior of Algorithm 5.2 is depicted in Figure 5.7. In this example, the object O_3 will be freed in Line 4, since it is not f_i-occupying.

Algorithm 5.2 Robson's "bad" program P_R

Initially: $f_0 := 0$

1: Allocate M objects of size 1.
2: **for** $i = 1$ to γ **do**
3: Pick $f_i \in \{f_{i-1}, f_{i-1} + 2^{i-1}\}$ that maximizes

$$\sum_{\text{o is live and } f_i\text{-occupying}} 2^i - |o|.$$

4: Free all non-f_i-occupying objects
5: Allocate as many objects of size 2^i as possible (within the M live space bound.)
6: **end for**

When compaction occurs, P_F immediately deallocates the moved objects. But we would still like to adopt the original analysis of Robson without reanalyzing for the slightly modified

version that was used in the first stage of P_F. To this end, we use a thought experiment in which we let Robson's original malicious program P_R run against an imaginary memory manager A' that does not move objects. Clearly, Robson's analysis holds for the execution (P_R, A'), as it holds for all memory managers that do not move objects. From this analysis we will also be able to deduce a lower bound on the heap size that A uses while satisfying P_F's allocation and deallocation sequence. The only difference between the first stage of P_F and P_R is that P_F must deal with compacted objects. (Such objects are deallocated by P_F, but still count for the decisions on future deallocation of all objects.) Otherwise, it behaves exactly like the original P_R.

In the discussion in this subsection we are concerned only with the execution of the first stage of P_F. In what follows, when we mention P_F, we only look at the first stage of P_F.

The imaginary memory manager A' is constructed only for this proof and has no use otherwise. We therefore do not care much about its efficiency or generality. A' will be looking at the run of P_F against A in order to make its allocation decisions. Actually, it will make sure that the program P_R makes the same allocation requests as P_F makes when running against A. But A' will satisfy them with no compaction. Since P_R will make the same allocation sequence, A' knows exactly which allocations to expect during the execution against P_R.

The memory manager A' will require more heap space to satisfy the demands of P_R than the original memory manager A needs to satisfy P_F, but A' will make sure that the number of f_i-occupying objects in each step i is similar throughout the execution. This similarity is achieved by maintaining a one to one mapping between objects in the execution of (P_F, A) and objects in the execution of (P_R, A'), such that mapped objects are always the same size, and are either both f_i-occupying or are both not f_i-occupying. Since the allocation sequence of P_R and P_F is determined by the space consumed by f_i-occupying objects, we get that these sequences are equal for both programs. Interestingly, the set of f_γ-occupying objects at the end of executing the first stage can be used to bound the value of the potential function $u(t_{first})$ (from below) at the end of P_F's first stage.

It still remains to show how we handle the case that an object is compacted, and why this does not break the maintained mapping between objects. This is exactly the reason why ghost objects were defined and used. Objects that have been moved by the memory manager and immediately deallocated by the program P_F are considered by P_F as remaining in their original location (where they were allocated) as ghosts. This means that the memory manager A can allocate space at this original location and it simply ignores these ghost objects. But the malicious program P_F does consider their sizes when it needs to decide which objects to delete and how many objects to allocate. If ghost objects are f-occupying, then they are counted in the summation there.

Let us now specify the imaginary memory manager A' (which depends on the execution (P_F, A)) such that the execution (P_F, A) is made similar to the execution (P_R, A').

Definition 5.4.7. [Memory manager $A'(A, P_F)$] The memory manager $A'(A, P_F)$ works as follows. The k-th object that P' allocates is placed in a location in the memory whose address is

equal modulo 2^γ to where A places the k-th object that P_F allocated. There are infinitely many such locations, and A' arbitrarily chooses one that does not contain any other allocated object.

Indeed the arbitrary location A' uses to place the objects may seem too much, as A' may use a huge heap for that. But all we care about in the end is the accumulated size of objects that are f_γ-occupying, and this set will be the same for both executions of (P_F, A) and (P_R, A'). Robson's analysis will guarantee that this set will be large, and we will deduce the bound we need.

We now prove that the mapping between objects in the execution of (P_F, A) and objects in the execution of (P_R, A') indeed exists and satisfies some useful properties.

Claim 5.4.8. *Consider the execution of P_F against a memory manager A, and the execution of P_R against $A'(A, P_F)$, and suppose that both finished their Step i. There is one-to-one mapping between objects in A and objects in A' with the following property.*

1. *A live or a (non-deleted) ghost object in the execution (P_F, A) is mapped to a live object in the execution (P_R, A') and vice versa (a live object in (P_R, A') is mapped to either a live or a ghost object in (P_F, A)).*
2. *The sizes of two mapped objects are equal.*
3. *The addresses of two mapped objects are equal modulo 2^γ.*

Moreover, the total number of objects allocated during Step i is equal in both execution.

Proof. The one-to-one mapping we chose maps the k-th object that P_F allocated to the k-th object that P_R allocated. By the definition of A' their address is equal modulo 2^γ and we are done with the third property.

To show the first property we need to show that an object is deallocated at Line 8 of program P_F if and only if the mapped object is deallocated at Line 4 of program P_R. Note that objects deallocated at Line 8 of P_F are permanently deleted, regardless of whether they were ghosts. An object that was not deallocated in one of these lines is either live or ghost (if it was compacted). We call objects that are not ghosts and not live *free* objects. We need to show that a non-free object is mapped to a non-free object. We combine this proof with showing that in each step the same number of objects are allocated. Since all objects allocated in Step i are of size 2^i, this provides the second property as well, i.e., that the size of objects that map to each other is equal.

We now prove that a non-free object is mapped to non-free object, and the total number of objects allocated during Step i is equal in execution of (P_F, A) and (P_R, A'). The proof is by induction on the number of steps. Before Step 1, both programs allocate M objects, and there are no free objects (since P_F does not reach Line 29 and P_R does not reach Line 4). So all objects are live and of the same size, which provides a base for the induction. Next, suppose that before starting Step i, $i = 1, 2, \ldots$, both programs allocated the same number of objects, and a non-free object is mapped to a non-free object. We show that the same holds for Step i.

In the beginning of the step execution, a new offset f_i is computed. To determine f_i, P_F considers the set of live objects, plus the set of compacted objects (in their original location)

that have become ghosts, but have not yet been deleted (as ghosts). By the induction hypothesis, these objects are mapped to objects that are live in the execution of (P_R, A'), and thus these are exactly the objects P_R considers when computing its new offset f_i. Moreover, by Property (3), an object is f_i-occupying if and only if its mapped object is also f_i-occupying, since by the induction hypothesis the object sizes are equal, and we also know that their addresses modulo 2^i are equal. Thus, both programs must compute the same new offset f_i. After setting the same f_i, we get that an object is non f_i-occupying if and only if the mapped object is also non f_i-occupying. Therefore, an object is freed at Line 8 of P_F if and only if the mapped object is freed at Line 4 of P_R.

Robson's program uses all space it has for allocation. Thus it allocates $\left\lfloor \left(M - \sum_{o \text{ is live}} |o|\right) / 2^i \right\rfloor$ objects of size 2^i. Meanwhile, P_F allocates $\left\lfloor \left(M - \sum_{o \text{ is live or ghost}} |o|\right) / 2^i \right\rfloor$ such objects. But every live object in A' is mapped to a live or ghost object in A, so they allocate exactly the same number of objects and we are done. □

The following is an implicit lemma from Robson's analysis that will help us bound the value of the potential function at the end of the first stage of P_F.

Claim 5.4.9 ([Rob74], Inequality 1)**.** *After the execution of Step i of P_R, there are at least $M\frac{i+1+1}{2^{i+1}} = M\frac{i+2}{2\cdot 2^i}$ objects that are f_i-occupying.*

We proceed with proving the first part of Lemma 5.4.5 by using Claim 5.4.9 and the mapping of Claim 5.4.8. Let us first recall the definition of the potential function. By Definition 5.4.3:

$$u_D(t) = \min(2^\gamma \cdot \sum_{o \in O_D(t)} |o|, 2^i).$$

Note that we ignore the members of $\mathscr{E}$ in Definition 5.4.3. This is because $\mathscr{E}$ is the empty set at the end of the first stage. By Definition 5.4.4 the potential function is:

$$u(t) = \left(\sum_{D \in \mathscr{D}(i)} u_D(t) \right) - \frac{n}{4}.$$

Recall also how f_γ-occupying objects are determined (Definition 5.4.2). For an integer f_γ satisfying $0 \leq f_\gamma < 2^i$ (computed by P_F), an object is f_γ-occupying if it occupies a word at address $k \cdot 2^\gamma + f_\gamma$ for some $k \in \mathbb{N}$. Finally, recall that t_{first} is the time immediately after executing Step 11. For a chunk D, $O_D(t)$ is the set of object that P_F associates with the chunk. In the first stage, association is determined by the program P_F in Step 11, where P_F associates a f_γ-occupying object with the chunk that contains its f_γ-occupying word. Thus, the number of f_γ-occupying objects is related to $O_D(t)$, which is related to the value of the potential function at time t_{first}. This relation is next used to transfer Robson's guarantee for many f_γ-occupying objects into a lower bound on the potential function.

Claim 5.4.10. *Let A be a memory manager, and let t_{first} be the time P_F finishes the execution of*

its first stage against A. Then

$$u(t_{first}) \geq M(\gamma/2+1) - 2^{\gamma} \cdot q_1 - \frac{n}{4}$$

Proof. Consider the time when Robson's program P_R finished executing the first γ step. By Claim 5.4.9 there are at least $M\frac{\gamma+2}{2\cdot 2^{\gamma}}$ live objects that are f_γ-occupying. Let t_{first} be the time immediately after P_F finished executing its first γ steps. By Claim 5.4.8, there exists a mapping between objects in the execution of P_R and P_F. Moreover, an object is f_γ-occupying in the execution of P_R if and only if the mapped object is f_γ-occupying in the execution of P_F. Thus, there are at least $M\frac{\gamma+2}{2\cdot 2^{\gamma}}$ live or ghost objects that are f_γ-occupying at time t_{first} as well. Recall that q_1 is the total size of objects compacted during P_F's first stage. Therefore, the number of objects compacted cannot exceed q_1, and therefore the number of ghost objects is at most q_1. Thus, at the end of step γ, the number of *live* (non-ghost) f_γ-occupying objects is at least $M\frac{\gamma+2}{2\cdot 2^{\gamma}} - q_1$.

Next, consider the partition of the heap into chunks of size $2^{2\gamma-1}$, and let D be a chunk in the partition. According to Step 11 of Algorithm P_F, the set O_D of the objects associated with chunk D is the set of live objects whose f_γ-occupying word falls inside D; there are at most $2^{\gamma-1}$ such objects since there are at most $2^{\gamma-1}$ words in D whose address equals f_γ moduli 2^{γ}. According to Definition 5.4.3, the value of $u_D(t_{first})$ is the accumulative size of objects in O_D multiplied by 2^{γ}; this value is capped by $2^{2\gamma-1}$ in Step $2\gamma-1$. Since the size of each associated object is at least 1, we have for each chunk D

$$u_D(t_{first}) \geq \min\left(|O_D| \cdot 2^{\gamma}, 2^{2\gamma-1}\right).$$

Since a chunk D contains at most $2^{\gamma-1}$ objects that are f_γ-occupying, we have

$$u_D(t_{first}) \geq |O_D| \cdot 2^{\gamma}.$$

The summation of $u_D(t_{first})$ over all chunks in the heap is 2^{γ} times the number of live f_γ-occupying objects. Thus

$$u(t_{first}) \geq 2^{\gamma} \cdot \left(M \cdot \frac{\gamma+2}{2\cdot 2^{\gamma}} - q_1\right) - \frac{n}{4} = M \cdot \frac{\gamma+2}{2} - 2^{\gamma} \cdot q_1 - \frac{n}{4}.$$

Next we bound from above the memory allocated by Robson's algorithm. It is used to bound the compaction allowed for a c-partial memory manager.

Claim 5.4.11. *Let A be a memory manager, and consider the execution of P_F's first stage against A. The total size of memory that P_F allocated is at most*

$$s_1 \leq M\left(\gamma + 1 - \frac{1}{2}\sum_{i=1}^{\gamma}\frac{i}{2^i-1}\right).$$

Proof. Recall that by Claim 5.4.8, the number of objects allocated by P_F (when working with

the memory manager A) at each step equals the number of objects allocated by P_R at that step, when working with the modified memory manager A'. Thus, it is enough to bound the total size of objects allocated during the execution of P_R against $A'(A,P)$. In Step 0, P_R allocates M words. In Step i, P_R allocates $\left\lfloor \left(M - \sum_{o \text{ is live}} |o|\right) / 2^i \right\rfloor$ objects, and the total allocated space is at most

$$M - \sum_{o \text{ is live}} |o|.$$

Let x_i be the number of objects that P_R allocated in Step i. The size of each object is 2^i so the total space allocated is $2^i \cdot x_i$. By Claim 5.4.9, at least $M\frac{i+2}{2\cdot 2^i}$ objects will be f_i-occupying live objects after the allocation of Step i. Exactly x_i of these are newly allocated objects (objects allocated during Step i) and the rest are old live objects. Therefore, before P_R starts allocating objects in Step i, the total number of live objects is at least $M \cdot \frac{i+2}{2\cdot 2^i} - x_i$ and clearly the total space they span is no smaller than that. If old objects take at least $M \cdot \frac{i+2}{2\cdot 2^i} - x_i$ words, then the space available for allocation in Step i is at most:

$$2^i \cdot x_i \leq M - \left(M \cdot \frac{i+2}{2\cdot 2^i} - x_i\right) = M\frac{2\cdot 2^i - i - 2}{2\cdot 2^i} + x_i.$$

Subtracting x_i from the inequality and then multiplying by $\frac{2^i}{2^i-1}$, we get

$$2^i \cdot x_i \leq M\frac{2^i - 1 - \frac{i}{2}}{2^i - 1} = M - M\frac{i}{2(2^i-1)}.$$

Finally, the total size of memory that P_R allocates in the first stage is simply a summation over all steps in the first stage, i.e.,

$$s_1 \leq M + \sum_{i=1}^{\gamma}\left(M - M\frac{i}{2(2^i-1)}\right).$$

as required. □

The proof of Lemma 5.4.5 follows from Claim 5.4.10 and Claim 5.4.11.

5.4.3 Analysis of the second stage

In this section we prove Lemma 5.4.6, which asserts a substantial growth in the potential function during the second stage. We let t_{first} represent the time where the first stage completes and t_{finish} the time the second stage (and the entire algorithm) completes.

Before restating the lemma, let us recall the variables' notations in the context of stage 2. The space allocated during the second stage is denoted by s_2; q_2 denotes the space compacted during the second stage, the variable M denotes the total size of objects that are live simultaneously, h is (by intuition) a lower bound on heap size used, and γ is a parameter set so that $2^{-\gamma}$ is the desired density. Let us recall the statement of the lemma.

Lemma 5.4.6. *Let A be a c-partial memory manager, and let t_{finish} be the time that P_F finishes*

its execution with A as its memory manager. Then,

$$u(t_{finish}) - u(t_{first}) \geq \frac{3}{4}s_2 - 2^{\gamma} \cdot q_2.$$

Additionally, either the memory manager uses more than $M \cdot h$ space, or the allocated space s_2 in the second stage satisfies

$$s_2 \geq M\left(\frac{\lg(n) - 2\gamma - 1}{\gamma + 1}\right)\left(1 - 2^{-\gamma} \cdot h\right) - 2n.$$

To show that this lemma holds, we look at changes to the potential function u that might occur during the execution. This includes allocation of new objects (initiated by P_F), compaction of objects (by the memory manager), deallocation of objects (by P_F), and a move from one step to the next one that changes the chunk sizes, and therefore the summation over the chunks and each chunk's $u_D(t)$ function. We will show that we get substantial growth of the potential function during allocations, and also that all other events do not decrease it.

Recall that the potential function is defined as (Definition 5.4.4):

$$u(t) = \left(\sum_{D \in \mathscr{D}(i)} u_D(t)\right) - \frac{n}{4},$$

and we need to show that the value of the potential function grows by at least $\frac{3}{4}s_2 - 2^{\gamma} \cdot q_2$ during the second stage of the execution.

Before we start the proof of this lemma, let us define the association of objects into chunks. This association is managed explicitly by P_F, and we state it here for completeness.

Definition 5.4.12. Let A be a c-partial memory manager, let t be a time during the execution of P_F second stage, and let i be the step in which t occurs. Let D be a 2^i chunk, and let O_D be the set of objects and half objects which P_F associates with the chunk D.

Association An object or half an object is added to the associated set of a chunk D (associates with D) only in

- P_F Line 11 (before the second stage started)
- P_F Line 38 (a new object is allocated).

De-association An object is removed from the associated set of a chunk D (de-associated) only in

- P_F Line 21 (P_F frees an object).
- P_F Line 38. This case occurs when an object o' is associated with a chunk D at time $t-1$. If at time t a new object o is allocated and placed on the chunk D, then the association set of D is changed to contain only o, and o' is deleted.

Compaction of an object *does not* de-associate it. Even thought the object is not live (because of P_F Line 2), it is still associated with the original chunk until P_F removes the association at Line 38.

Changed The association set of a chunk D may change only at the following lines:

- P_F Line 38. A new object is allocated.
- P_F Line 21. P_F frees an object.
- P_F Line 15. In this case a new step, i, kicks in, and two adjacent 2^{i-1} chunks are merge into one 2^i chunk. Objects never "disappear" during the merge; rather, the two association sets of the 2^{i-1} chunks are merged into a single association set.

We start by looking at allocations and show that whenever P_F allocates an object, either the potential function gets larger or some compaction occurs (and the potential function does not decrease). (We will later use the fact that compaction is limited to get the potential function growth we need.) Suppose an object o is allocated during Step i of the execution of P_F with A. By the definition of P_F, the size of o is $4 \cdot 2^i$. Thus, when the memory manager places o in the heap, it consumes at least three full consecutive chunks (of size 2^i) and then some additional space from the chunks neighboring these. Denote by D_1, D_2 and D_3 the three chunks that are fully covered by o and are selected in Step 36 of Algorithm 5.1. All three must be empty when o is placed. We claim that this transition from empty chunks to full chunks makes the potential function grow. We will show that the value of $u_{D_1}(t) + u_{D_2}(t) + u_{D_3}(t)$ grows, which implies that the value of $u(t)$ grows. Note that $u_D(t)$ for any other chunk D (other than D_1, D_2 or D_3) is not affected by this allocation, because the set of associated objects as well as membership in $\mathscr{E}$ (which we define in the next paragraph) only changes for D_1, D_2, and D_3.

When an object o is allocated by P_F in Line 38, P_F associates D_1 with the first half of o and D_3 with the second half of o. But since an object is associated with at most two chunks (each half can be associated with a chunk), it follows that D_2 is left with no associated object, despite of it being completely covered by the allocated object o. The goal of the set $\mathscr{E}$ is to deal with these middle chunks. This set will contain all such middle chunks and make $u_{D_2}(t)$ of Definition 5.4.3 be set to 2^i. We note that this "anomaly," of a chunk being covered by an object but with no associated object, is temporary and disappears at the next step change since the middle chunk is joined with either its left or right chunk, and they become a single chunk with which o (or o's half) can be associated. Let us now define $\mathscr{E}$.

Definition 5.4.13. [The set $\mathscr{E}(t)$] Let A be a memory manager, and consider any time t during the execution of P_F's second stage with A as its memory manager. Let i be the step where t happens. The *set of middle chunks* $\mathscr{E}(t) \subset \mathscr{D}(i)$ is the set of chunks whose left adjacent chunk and right adjacent chunk were fully covered by an object o allocated in Step i (thus, the chunk itself is also covered by o), but half of o was not associated with it. A chunk remains in $\mathscr{E}(t)$ until either a new step kicks in or an object is associated with this chunk. The latter may occur if o was compacted during step i, and another object is allocated there.

Let us now claim a simple, intuitive property about chunks in $\mathscr{E}$: if a chunk is in $\mathscr{E}$, then the two chunks adjacent to it are associated with a "big," or "recently allocated" object.

Claim 5.4.14. *Let t be a time during the execution of P_F against a memory manager A, and let i be the step in which t occurs. Let D_1 and D_2 be two consecutive chunks such that $D_1 \in \mathscr{E}(t)$. Then there exists an object o that is allocated during Step i, such that at time t half of o is associated with D_2. The same holds for D_1 when $D_2 \in \mathscr{E}(t)$.*

Proof. Let t' be the time when D_1 joined the set $\mathscr{E}(t)$, and let o' be the object whose allocation caused D_1 to enter $\mathscr{E}$. By definition of P_F in step 34, half of o' was associated with D_2 at time t'. If this is still the case at time t, we are done. Otherwise, let o'' be the last object that changed the association of D_2 before time t (which occurs in Step i). If half of o'' was associated with D_2, we are done. Otherwise, o'' was associated with both chunks that neighbored D_2, which means that half of o'' was associated with D_1, contradicting the assumption that $D_1 \in \mathscr{E}$. It is easy to see that a similar argument holds for D_1 in the case where $D_2 \in \mathscr{E}(t)$. □

The chunks D_1, D_2 and D_3 were empty before the allocation. For each of these chunks, we distinguish between the case where an object was associated with it before the allocation, and the case where no object was so associated. In the latter case, the value of the function $u_D(t)$ for that chunk grows since an object gets associated with it. In the former case, it must be that the memory manager compacted away all objects that were allocated on the chunk and made it fully available for allocation. Note that the definition of P_F rules out a third possibility that these chunks were emptied due to deallocation of the objects that previously resided on them. Object deallocation is initiated by the program P_F only (and not by the memory manager). By the definition of P_F, it only deallocates an object when there are enough other objects left on the chunk to make the remaining space size at least $2^{i-\gamma}$.

When the second case occurs, i.e., the memory manager compacts away objects from a chunk before placing o, we are not able to show that the value of the potential function grows. However, we get that some compaction occurred, and we use that to bound the number of such events. To this end, we associate some compaction value with the newly allocated object. Note that the objects that were compacted away from a chunk and then deallocated immediately by P_F are still considered associated with the chunk until a new object is placed on it. Recall (Definition 5.4.12) that associations of objects with chunks are managed explicitly by P_F, and association changes only at Line 15 (phase change), Line 21 (P_F frees objects), and Line 38 (a new object). An association terminates only when P_F frees the object in Line 15 or a new object is placed on top of its former location (and a new association kicks in) at Line 38. Therefore, to determine how much compaction occurred to free space for the allocation, we can just check the objects that were associated with these chunks right before the allocation. The formal definition follows.

Definition 5.4.15. [Compaction space associated with an object] Let o be an object allocated during the execution of P_F's second stage against a memory manager A. Let $t+1$ be the allocation time (and t be the time just before the allocation), and let D_1, D_2, and D_3 be the

chunks picked by P_F in Step 36 of P_F, after allocating o. Then we define *the compacted space associated with the object o* to be

$$q(o) = \sum_{o' \in O_{D_1}(t)} |o'| + \sum_{o' \in O_{D_2}(t)} |o'| + \sum_{o' \in O_{D_3}(t)} |o'|$$

Next, we establish some properties of the set of objects associated with every chunk.

Claim 5.4.16. *Consider any point of time in step i of the execution of P_F's second stage against a memory manager A. Then the following three properties hold:*

1. *The sets $\{O_D : D \in \mathscr{D}(i)\}$ are disjoint.*
2. *Every live object o is either associated with a single chunk or its two halves are associated with two chunks.*
3. *If a live object o is associated with a chunk D, then o intersects D.*

Proof. A half-object is associated with a chunk only once (during allocation) and never gets reassigned and so all half-objects satisfy the first property. When two half-objects are merged to a single object, this single object is assigned to a single chunk and never reassigned again. Therefore the first property always holds.

As for the second property, we show that if an object is not associated with any chunk then the object is not live. First note that when an object is allocated, it is associated (via its two halves) with two chunks. An object is de-associated only if either P_F frees this object (in steps 21 and 29), or another object is allocated and is associated with the same chunk with which the previous object was associated (in step 34). In the latter case, the object can not reside on the same chunk where the new object resides. Thus, it was compacted by the memory manager and deallocated by P_F immediately. In both cases P_F frees the object, so it is not live.

To show that the third property holds, we note that an object (or half an object) o is associated with a chunk it intersects during allocation and this association is preserved during step changes that unite chunks and their associated sets. A compacted object does not change association and therefore may foil the third property. However, P_F immediately deletes such objects and they are therefore neither live nor associated any longer. Recall that a compacted object is freed by P_F, so it is not live and does not preserve this property. □

We now show that the potential function $u(t)$ indeed increases substantially. We will show that no event causes a decrease in it, while allocations cause sufficient increase. We will compute the potential function $u(t)$ increase for each allocation, and then sum over all allocations to obtain the total increase in $u(t)$ during P_F's second stage.

Before stating the claim, let us recall the definition of potential function that is used extensively in the proof of this claim.

Definition 5.4.3:

$$u_D(t) = \begin{cases} 2^i & D \in \mathscr{E}(t) \\ \min(2^\gamma \cdot \sum_{o \in O_D(t)} |o|, 2^i) & \text{otherwise} \end{cases}.$$

Definition 5.4.4: $u(t) = \left(\sum_{D \in \mathscr{D}(i)} u_D(t)\right) - \frac{n}{4}$.

Claim 5.4.17. *Let A be a memory manager, let $u(t)$ be the potential function as defined in Definition 5.4.4, and let $q(o)$ be the compaction associated with an object as defined in Definition 5.4.15. Then, during the execution of P_F against A, the following properties of $u(t)$ hold:*

1. *No event in the execution causes $u(t)$ to decrease.*
2. *During an allocation of an object o, $u(t)$ increases by at least $\frac{3}{4}|o| - 2^{\gamma} \cdot q(o)$.*

Proof. The potential function only changes when the set of chunks changes during step transition or when the set of associated objects of a chunk changes. By definition 5.4.12 there are three types of events that may cause such a change. Note that compaction of an object by A is not an event that influences the potential function, since the object association remains unchanged.

A step transition causes P_F to repartition the heap Consider the time t when P_F repartition the heap (when a new step i kicks in) and $t-1$ is a time that still belongs to Step $i-1$. Let D be a chunk of Step i (of size 2^i). D is composed of two chunks D_1 and D_2 of Step $i-1$ (and of size 2^{i-1}). It suffices to show that $u_D(t) \geq u_{D_1}(t) + u_{D_2}(t)$.

If either $D_1, D_2 \in \mathscr{E}(t-1)$, then by Claim 5.4.14 there exists an object o that was allocated in Step $i-1$, and half of o is associated with either D_1 or D_2. Objects allocated in Step $i-1$ are of size 2^{i+1}. Since half of o is associate with D, and the size of half of o is 2^i, it holds that $\sum_{o' \in O_D} |o'| \geq 2^i$. Also note that by Definition 5.4.13, $D \notin \mathscr{E}$ since chunks are inserted to $\mathscr{E}$ only after the allocation step. Thus, by Definition 5.4.3 of $u_D(t)$,

$$u_D(t) \geq \min(2^{\gamma} \cdot 2^i, 2^i) = 2^i \geq u_{D_1}(t-1) + u_{D_2}(t-1).$$

The last inequality follows since by Definition 5.4.3 $u_{D'}(t) \leq 2^{i-1}$ for every chunk D' in step $i-1$. Otherwise, both chunks are not in $\mathscr{E}$ and we know that the objects associated with them are disjoint. Therefore, according to Step 15 in P_F,

$$\sum_{o \in O_D} |o| = \sum_{o \in O_{D_1}} |o| + \sum_{o \in O_{D_2}} |o|.$$

In this case the statement follows since

$$\min\left(2^{\gamma} \sum_{o \in O_{D_1}} |o| + 2^{\gamma} \sum_{o \in O_{D_2}} |o|,\ 2^i\right) \geq \min\left(2^{\gamma} \sum_{o \in O_{D_1}} |o|,\ 2^{i-1}\right) + \min\left(2^{\gamma} \sum_{o \in O_{D_2}} |o|,\ 2^{i-1}\right).$$

which follows by the mathematical inequality $\min(A+B, C+D) \geq \min(A,C) + \min(B,D)$.

P_F deallocates an object By definition of P_F Step 20, P_F does not free an object from a chunk D if the deletion decreases $\sum_{o \in O_D} |o|$ below $2^{i-\gamma}$. Let t be a time (in step i) when an object was deleted from a chunk D. Thus, $\sum_{o \in O_D} |o| \geq 2^{i-\gamma}$, which implies that $u_D(t) \geq 2^i$. Since 2^i is the

maximal value of $u_D(t)$ in step i, it was not higher before the deletion, and the claim holds in this case.

P_F allocates an object o Let t be the time P_F allocates an object o, and let D_1, D_2, D_3 be the three chunks that P_F picked. By definition of P_F Step 38, only the sets $O_{D_1}, O_{D_2}, O_{D_3}$ are changed during allocation. To show that the value of $u(t)$ increases by $\frac{3}{4}|o| - 2^\gamma q(o)$ after the allocation of o, it is sufficient to show that $u_{D_1}(t) + u_{D_2}(t) + u_{D_3}(t) - (u_{D_1}(t-1) + u_{D_2}(t-1) + u_{D_3}(t-1)) \geq \frac{3}{4}|o| - 2^\gamma q(o)$.

After the allocation, the size of each half an object is $2 \cdot 2^i$, so $u_{D_1}(t) = u_{D_3}(t) = \min(2^\gamma \cdot 2 \cdot 2^i, 2^i) = 2^i$, and $D_2 \in \mathscr{E}(t)$. Therefore,

$$u_{D_1}(t) + u_{D_2}(t) + u_{D_3}(t) = 3 \cdot 2^i = \frac{3}{4}|o|.$$

Next we show that $2^\gamma \cdot q(o) \geq u_{D_1}(t-1) + u_{D_2}(t-1) + u_{D_3}(t-1)$, which finishes the proof for the allocation case. Recall that by Definition 5.4.15 we have

$$q(o) = \sum_{o' \in O_{D_1}(t-1)} |o'| + \sum_{o' \in O_{D_2}(t-1)} |o'| + \sum_{o' \in O_{D_3}(t-1)} |o'|.$$

Before the allocation, if either chunk was contained in $\mathscr{E}(\sqcup - \infty)$, then by Claim 5.4.14 there exists an object o'' allocated in step i, and half of o'' is associated with D_1, D_2, or D_3. Since o'' was allocated at Step i, the size of half of o'' is 2^{i+1}. In this case $q(o) \geq 2^{i+1}$, and $2^\gamma \cdot q(o) \geq 2^\gamma \cdot 2^{i+1} \geq 3 \cdot 2^i \geq u_{D_1}(t-1) + u_{D_2}(t-1) + u_{D_3}(t-1)$. The second inequality follows since $\gamma \geq 1$, and the last inequality follows since by Definition 5.4.3 $u_D(t) \leq 2^i$ for every chunk D.

If none of the chunks are contained in $\mathscr{E}(t-1)$, then $u_{D_i}(t-1) = \min\left(2^\gamma \cdot \sum_{o' \in O_{D_i}(t-1)} |o'|, 2^i\right) \leq 2^\gamma \cdot \sum_{o' \in O_{D_i}(t-1)} |o'|$ for $i = 1,2,3$. Thus, $2^\gamma q(o) = 2^\gamma \sum_{o' \in O_{D_1}(t-1)} |o'| + 2^\gamma \sum_{o' \in O_{D_2}(t-1)} |o'| + 2^\gamma \sum_{o' \in O_{D_3}(t-1)} |o'| \geq u_{D_1}(t-1) + u_{D_2}(t-1) + u_{D_3}(t-1)$. In both cases we have $2^\gamma q(o) \geq u_{D_1}(t-1) + u_{D_2}(t-1) + u_{D_3}(t-1)$ and we are done.

We now turn to bound the memory that P_F allocates during the execution of its second stage. By its definition, in Step 34, P_F attempts to allocate $\lfloor x \cdot M \cdot 2^{-i-2} \rfloor$ objects of size 2^{i+2} without exceeding the M bound for the total size of allocated memory. In Lemma 5.4.6 the growth in potential function depends positively on the memory allocated. Thus, we would like to show that P_F allocates a lot of memory. To this end, we need to maximize x but also show that there is enough allocation budget so that the bound of M simultaneously live words does not hinder allocation. Namely, we need to show that there is enough allocation "budget" or that the space occupied by live objects in the heap is bounded from above.

The next proposition pertains to the shape of chunks. It asserts that any chunk D in the heap is either empty, or it contains a single (large) object, or the total space of its associated objects O_D is bounded exactly by $2^i/2^\gamma$, which is the density that P_F attempts to preserve in each chunk. This proposition will later be used to bound the total size of live objects in the heap. Before

stating the proposition we recall lines 20 – 21 of the program P_F. These lines stand at the heart of the proposition.

Line 20: Pick a maximal set $X \subset O_D$ such that $\sum_{o \in O_D \setminus X} |o| \geq 2^{i-\gamma}$.
Line 21: $O_D := O_D \setminus X$.

Proposition 5.4.18. *Consider the execution of P_F's second stage against a memory manager A. Let t be a time in step i when P_F allocates an object. Then for every chunk $D \in \mathscr{D}(i)$, either $|O_D(t)| \leq 1$ or $\sum_{o \in O_D(t)} |o| \leq 2^{i-\gamma}$.*

Proof. Objects are allocated only in Lines 34-38 of P_F, so it is sufficient to prove the proposition during the execution of this line. Fix the time t' after P_F finished the execution of Line 32 in Step i. We show that the Proposition holds at time t', and it does not change during Lines 34-38 of P_F.

Consider a chunk D, and the set of objects associated with this chunk $O_D(t')$. By Line 20 of P_F, $O_D(t')$ is minimal in the sense that deallocating any object associated with D would break the density inequality $\sum_{o \in O_D(t')} |o| \geq 2^{i-\gamma}$. If $O_D(t')$ contains an object larger than or equal to $2^{i-\gamma}$, this is the only object associated with D, and we are done since $|O_D(t)| \leq 1$.

Otherwise, all objects associated with D are smaller than $2^{i-\gamma}$. The size of all objects associated with D is a powers of 2; only such objects are allocated, and P_F associates either an object or half an object with a chunk. The reader should note that object sizes are always integral; objects of size 1 are allocated in the first stage and are not broken into two halves. Now, suppose by way of contradiction that $\sum_{o \in O_D(t')} |o| > 2^{i-\gamma}$. Let the size of smallest object in $O_D(t')$ be $2^k < 2^{i-\gamma}$. Since this object was not de-associated in Step 21, we know that $\sum_{o \in O_D(t')} |o| < 2^{i-\gamma} + 2^k$. Combining this inequality with our assumption, we get $2^{i-\gamma} < \sum_{o \in O_D(t')} |o| < 2^{i-\gamma} + 2^k$.

But since object sizes are powers of two, their sum $\sum_{o \in O_D(t')} |o|$ must be divisible by the size of the smallest object 2^k and we get a contradiction.

Finally, we show that the execution of Lines 34-38 of P_F does not break the guarantee of the proposition. If the set $O_D(t)$ changed during the execution of Line 38, then at most a single half object is associated with the chunk D. Therefore, the Proposition holds at any time t during execution of Line 38, as required. □

Next, we can give a lower bound on the space that P_F allocates during its second stage.

Claim 5.4.19. *Consider the execution of P_F against a memory manager A. Then either A uses more than $M \cdot h$ heap space, or the number of words that P_F allocates during its second stage satisfies*

$$s_2 \geq (\lg n - 2\gamma - 1) M \frac{1 - h \cdot 2^{-\gamma}}{\gamma + 1} - 2n.$$

Proof. Recall that the second stage consists of Steps $i = 2\gamma .. \lg n - 2$. We will show for each such step that either A uses more than $M \cdot h$ heap space for allocation, or P_F allocates

$$\left\lfloor M \cdot x \cdot 2^{-i-2} \right\rfloor \cdot 2^{i+2} \geq M \cdot x - 2^{i+2}$$

words during the execution of the step, where x is the algorithm constant that is determined at the beginning of Algorithm 5.1 (P_F) to be:

$$x = \frac{1 - h \cdot 2^{-\gamma}}{\gamma + 1}. \tag{5.4}$$

The claim then follows by summing the allocation space at the $\lg n - 2\gamma - 1$ steps of P_F's second stage:

$$\sum_{i=2\gamma}^{\lg(n)-2} \left(M \cdot x - 2^{i+2}\right) = \sum_{i=2\gamma}^{\lg(n)-2} M \cdot x - \sum_{i=2\gamma}^{\lg(n)-2} 2^{i+2} \leq (\lg n - 2\gamma - 1)\, M \cdot x - 2n.$$

Assume that A uses less than $M \cdot h$ heap space, and let us show that allocation of $\left\lfloor M \cdot x \cdot 2^{-i-2} \right\rfloor$ chunks of size 2^{i+2} is possible, in the sense that it does not exceed the total of M live words. In fact, we will show that it is possible to allocate $M \cdot x$ words, which is enough. Recall that Claim 5.4.16 states that every live object is associated with a chunk. Thus, it is sufficient to show that after allocation of $M \cdot x$ words, the total size of objects associated with some chunk is bounded by M.

We consider chunks according to the space consumed by their associated objects. First, consider chunks whose associated objects (and half-objects) are small, i.e., at most $2^{i-\gamma}$ words. By Proposition 5.4.18, the total space of objects associated with such a chunk is at most $2^{i-\gamma}$.

Next consider chunks with large associated objects, of size at least $2^{i-\gamma+1}$ and at most 2^{i+2}. By Proposition 5.4.18, if a non-small object is associated with a chunk, then this object is the only object associated with this chunk. So to bound the space such objects take, we simply bound the maximum possible space allocated by all associated large objects. Since each stage has its object size, then large objects are objects that have been allocated recently. More specifically, an object's size is at least $2^{i-\gamma+1}$ if it was allocated in Step $i - \gamma - 1$ or thereafter. We will have the maximum number of such associated objects if all possible objects are allocated in each of the steps $j = i - \gamma - 1, \ldots, i$, and all of them are still associated with chunks at the time we consider the chunks. In each of these steps, at most $\left\lfloor M \cdot x \cdot 2^{-j-2} \right\rfloor$ objects of size 2^{j+2} are allocated, and at this point (in Step i) each of them can be associated with a single chunk, or two half-objects can be associated with two chunks.

The maximum is obtained when for each recent step j (except the last) there are $M \cdot x \cdot 2^{-j-2}$ objects of size 2^{j+2}, and each is associated with a single chunk. For the most recent step i, allocated objects are of size 2^{i+2} and they are split into halves and associated with two chunks. The most space is allocated when there are $M \cdot x \cdot 2^{-i-2}$ objects of size 2^{i+2} associated with $2 \cdot M \cdot x \cdot 2^{-i-2}$ chunks. The total number of chunks associated with a large object is

$$\left(\sum_{j=i-\gamma-1}^{i-1} M \cdot x \cdot 2^{-j-2} \right) + M \cdot x \cdot 2 \cdot 2^{-i-2} = M \cdot x \cdot 2^{-i+\gamma}.$$

The total size of these objects is $M \cdot x \cdot (\gamma + 2)$.

Recall that we have assumed that the heap size is at most $M \cdot h$, and want to show that

allocation of $\lfloor M \cdot x \cdot 2^{-i-2} \rfloor$ chunks of size 2^{i+2} is possible, in the sense that it does not exceed the total of M live words. The maximum size of live memory is obtained where there are at most $M \cdot h \cdot 2^{-i} - M \cdot x \cdot 2^{-i+\gamma}$ chunks that are not associated with a large object, and each of them contains at most $2^{i-\gamma}$ words. Thus, the total size of objects associated with some chunk is at most

$$M \cdot x \cdot (\gamma+2) + M\left(h \cdot 2^{-i} - x \cdot 2^{-i+\gamma}\right) 2^{i-\gamma} = M \cdot \left((\gamma+1)x + h \cdot 2^{-\gamma}\right).$$

Finally, setting the value of x as in equation 5.4, we get M, as required.

Recall that for the proof of Lemma 5.4.6, we want to show that $u(t)$ increases substantially. Claim 5.4.17 asserts an increase of $u(t)$ by at least $\frac{3}{4}|o| - 2^{\gamma} \cdot q(o)$ for each allocation of an object o. Summing $\frac{3}{4}|o|$ over all objects allocated during stage two of P_F gives $\frac{3}{4}s_2$, which we also computed. To get a lower bound on how much $u(t)$ increases, we need to lower bound the sum of $q(o)$ over all objects allocated during stage two of P_F. (For the definition of $q(o)$ see Definition 5.4.15.)

Proposition 5.4.20. *Consider the execution of P_F against a memory manager A. Let S_2 be the set of objects allocated at P_F's second stage, and let q_2 be the total size of objects compacted during the second stage. Then $\sum_{o \in S_2} q(o) \leq q_2$.*

Proof. Consider an object $o \in S_2$ allocated at time $t+1$, and let D_1, D_2, D_3 be the chunks that P_F picked at line 36 after allocating o. Let o' be an object associated with one of the chunks D_1, D_2, D_3 at time t (just before o was allocated). By Claim 5.4.16, o' intersects one of these chunks D_1, D_2, D_3. Since o covers all these chunks, o' intersects o, and o' cannot remain in its original location at time $t+1$. Also, o' was not deallocated at Line 29, since otherwise it would no longer be associated with any chunk. Thus, o' was compacted by A before o was placed and so o' is counted with $q(o)$. Moreover, at time $t+1$, the object o' is not associated with any chunk (which follows by definition of P_F's Line 38). Therefore, o' is not considered in the summation $\sum_{o \in S_2} q(o)$ ever again (under $q(o'')$ of any other object o''). Thus, the total size of compaction A performed during second stage is at least $\sum_{o \in S_2} q(o)$. □

Now we are ready to bound the total increase in the unavailable space at P_t's second stage

Claim 5.4.21. *Consider the execution of P_F against a memory manager A. Let t_{finish} be the time when P_F finished its execution, and let t_{first} be the time when P_F finished execution of first stage. Then*

$$u(t_{finish}) \geq u(t_{first}) + \frac{3}{4}s_2 - 2^{\gamma} \cdot q_2.$$

Proof. By Claim 5.4.17, $u(t)$ does not decrease during the execution of P_F's second stage. Furthermore, during allocation of an object $o \in S_2$, $u(t)$ increases by at least $\frac{3}{4}|o| - 2^{\gamma} \cdot q(o)$. Thus,

$$u(t_{finish}) - u(t_{first}) \geq \sum_{o \in S_2} \left(\frac{3}{4}|o| - 2^{\gamma} \cdot q(o)\right) \geq \frac{3}{4}s_2 - 2^{\gamma} \cdot q_2.$$

The last equality follows from the definition of s_2 (total memory allocated at stage 2), and Proposition 5.4.20 □

The proof of Lemma 5.4.6 follows from Claim 5.4.19 and Claim 5.4.21.

5.5 Upper bound: Proof of Theorem 5.2

In this section we provide an upper bound on fragmentation that improves over the best known upper bound for a memory manager that is c-partial. The upper bound is stated as Theorem 5.2. To prove Theorem 5.2, we present a c-partial memory manager A_C that keeps the heap size small.

Theorem 5.2 There exists a c-partial memory manager $A \in \mathscr{A}(c)$ that satisfies allocation requests of any program $P \in \mathscr{P}(M,n)$ with heap size at most

$$\max_{P\in\mathscr{P}(M,n)} HS(A_C,P) \leq M \cdot \left(\sum_{i=0}^{\lg n} a_i\right) + n(\lg(n)+1),$$

where $a_0 = 1$, and the values of a_i, $i = 1,\ldots,\lg(n)$, satisfy the following recursive formula:

$$a_i = \frac{2^{i+1}}{2^i+1} \cdot \left(1 - \sum_{j=0}^{i-1} \max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1}+1 \rfloor}{2^i}\right) \cdot a_j\right).$$

Recall that M denotes the bound on the space that the program may use simultaneously, n is the maximum object size, and c is the compaction bound. Since the expression in Theorem 5.2 is not easy to digest, we presented a graph of the upper bound as a function of c in Figure 5.3. For example, if $n = 1$MB, $M = 256n$, and $c = 100$, the upper bound on the heap size is $19.53M$. Namely, our allocator could serve the requests of any appropriate program with a heap of size $19.53M$. If we change c to be 50, then the upper bound becomes 19.00, and for $c = 20$, the upper bound is $17.32M$[2].

We start with some intuition. Suppose we pad each allocated object to make its size the closest (larger) power of two. Thus, we need to consider only $\lg(n)+1$ different sizes[3]. A very simple allocator that can serve all allocations with no compaction at all is one that uses a separate heap *area* of size $2M$ for each object size. Since the maximum space the program may use simultaneously is M, and since padding increases the required memory by at most a factor of 2, then an area of size $2M$ will always suffice, even if all allocated objects are the same size. However, this yields a memory size bound of $2M(\lg(n)+1)$, while our goal is to use partial compaction and be able to work with a smaller heap.

The allocator we present will allocate an object o of size $2^{i-1} < |o| \leq 2^i$ in any area j satisfying $j \leq i$. Note that we do not use padding in the actual algorithm. When an allocation request is received for an object o of size $2^{i-1} < |o| \leq 2^i$, the proposed memory manager will simply search for a free aligned location of size 2^i in all areas $j = 0,1,\ldots,i$. If no 2^i-aligned

[2]For $c = 20$, the values of the a_i are as follows: $a_{0-5} = 1, 0.67, 0.67, 0.81, 0.89, 0.94$, $a_{6-10} = 0.95, 0.95, 0.93, 0.91, 0.89$, $a_{11-15} = 0.84, 0.82, 0.80, 0.78, 0.76$, $a_{16-20} = 0.74, 0.72, 0.70, 0.68$.

[3]To keep the presentation simple, we assume that n (the size of the largest possible allocatable object) is a power of 2, which is typically the case in practice. Otherwise, we need to round $\lg(n)$ up each time we use it.

space is free, then the heap must be sparsely populated. The analysis will show that in this case there must be a 2^i aligned-chunk in which only $1/(2c+2)$ of the words are populated. In this case, the memory manager will compact the content of this chunk, recursively using the allocator on the compacted objects to find vacant places for them. The evacuated chunk is then used for satisfying the requested allocation. In essence, this is the algorithm we use. The choice of sizes for the area used to hold each object size and the analysis showing that with such sizes everything fits well are crucial.

The choice of the area sizes is a delicate matter. Although we would like to make the areas as small as possible (to obtain an overall small heap size) we must also keep the areas large enough so that the analysis can guarantee either an empty chunk available for allocation or a sparsely populated chunk appropriate for compaction. We specify the area sizes that we choose as a recursive series as follows.

For $i = 0,\ldots,\lg(n)$, we set each area size to be $n \cdot \lceil M \cdot a_i/n \rceil$, where $a_0 = 1$ and $a_i, i \geq 1$ is a (rational) number satisfying the recursive equation:

$$a_i = \frac{2^{i+1}}{2^i+1} \cdot \left(1 - \sum_{j=0}^{i-1} \max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1}+1 \rfloor}{2^i}\right) \cdot a_j\right). \tag{5.5}$$

Recall that M is the maximum simultaneously live space, and n is the size of the largest object the program allocates. Before presenting the algorithm, we define a few terms that will help us in our presentation of the algorithm. The first term is *aligned chunks*.

Definition 5.5.1 (A 2^i-aligned chunk)**.** Let S be a memory area. A 2^i-aligned chunk is a chunk of memory of size 2^i starting at an address divisible by 2^i.

The second term we define is an *uncrossed chunk*. We think of a crossed chunk as a chunk for which there exists an object that crosses its borders. Namely, it intersects the chunk, but not fully contained in it. An uncrossed chunk does not have an object crossing its borders like that. The memory manager we present never lets an object of size 2^i or smaller to cross the boundary of a 2^i-sized chunk. If a chunk is crossed it contains part of a large object, so the memory manager never considers such chunks as candidate for compaction.

Definition 5.5.2 (uncrossed chunk)**.** Let S be a memory space, and let D be a 2^i-aligned chunk fully contained in S. Let D_{prev} and D_{next} be the two neighboring 2^i-aligned chunks of D. We call D is *uncrossed* if no object intersects with both D and one of the neighbors of D.

The intuition described above is formalized in Algorithm 5.3.This algorithm receives an allocation request (a size) from the program and it determines an area for allocation of this object.

In what follows we prove the correctness of this memory manager, but let us first analyze the heap size it requires. Assuming a reasonable implementation in which the areas are placed one after the other consecutively, the heap size required by this algorithm is simply the sum of the areas it uses. This is stated in the next claim.

Algorithm 5.3 Memory manager A_C

Input: an allocation request for an object o of size $2^{i-1} < |o| \leq 2^i$

1: **if** there exists an empty 2^i-aligned chunk in an area j, $0 \leq j \leq i$ **then**
2: place the new object o at the beginning of such chunk.
3: **else**
4: Pick an uncrossed 2^i chunk D with density $< \frac{1}{2c+2}$.
5: **end if**
6: Logically place o at the beginning of D (to reserve the space).
7: Relocate each object $o' \in D$ recursively using A_C. (Recursive calls do not use the location
8: reserved for o.)
9: Place the new object o at the beginning of D.

Claim 5.5.3. *For any program $P \in \mathscr{P}(M,n)$, the size of memory A_C uses when serving P is at most*

$$HS(A_C,P) \leq M \cdot \left(\sum_{i=0}^{\lg n} a_i \right) + n(\lg(n)+1). \tag{5.6}$$

Proof. The heap size used by A_C is at most the total size of the areas in it. Thus,

$$\begin{aligned} HS(A_C,P) &\leq \sum_{i=0}^{\lg n} n \cdot \lceil M \cdot a_i/n \rceil \\ &\leq \sum_{i=0}^{\lg n} M \cdot a_i + \sum_{i=0}^{\lg n} n \\ &= M \cdot \left(\sum_{i=0}^{\lg n} a_i \right) + n(\lg(n)+1). \end{aligned}$$

The claim follows. □

Note that the heap size required by A_C is stated using a series of numbers a_i defined recursively. The exact size can be computed for any given parameter c. However, we did not see an easy way to solve the series sum analytically. In Section 5.6 we show that if we relax the bound a bit, we can get an explicit expression.

Next we prove that the algorithm is feasible, i.e., that it satisfies the memory requests of any program in $\mathscr{P}(M,n)$ successfully and is indeed a c-partial memory manager.

We first show that A_C satisfies the compaction budget c. Recall that upon each allocation of size s, the memory manager gets a compaction budget of s/c. We will show that every allocation pays the full budget for every compaction it causes. In addition, there is no "overdraft" in the budget as many allocations are served before a first compaction is required.

Claim 5.5.4. *For any program $P \in \mathscr{P}(M,n)$ and any allocation request of size $|o|$, the memory manager A_C will relocate objects of accumulating size that does not exceed $|o|/c$.*

Proof. We prove this claim by induction on the (logarithm of the) size of o. For the base of the induction, consider the case that A_C allocates an object o of size $|o| = 1$. Since $a_0 = 1$, then

the size of the first area is (at least) M. Since the total size of live memory in this area (which exclude the newly allocated object o) is at most $M-1$, there must be at least one empty chunk of size 1. Therefore, no compaction is needed and we are done with the base of the induction.

Next, suppose that the claim holds for all allocations of A_C for objects of size at most 2^{i-1}, and let us prove the claim for objects of size $2^{i-1} < |o| \leq 2^i$. Suppose that the allocation of o causes compaction from a chunk D of size 2^i. By the algorithm, the total live space in D is at most $\frac{2^i}{2c+2}$. Let o' be one of the objects relocated from D. Clearly, $|o'| \leq \frac{2^i}{2c+2}$. Compacting an object o' reduces the compaction budget by $|o'|$ (due to the relocation of the object). Since $|o'| \leq \frac{2^i}{2c+2} \leq 2^{i-1}$, the induction hypothesis applies to o', and the recursive call for placing each such o' reduces the compaction budget by at most $|o'|/c$.

The memory manager A_C picks a chunk D of density $< \frac{1}{2c+2}$. Thus, the compaction budget reduced by relocating objects from D is at most $\frac{2^i}{2c+2}$. In addition, the recursive use of A_C for allocating all objects out of D reduces the compaction budget by at most $\frac{2^i}{2c+2} \cdot \frac{1}{c}$. Overall, the budget is reduced by at most $\frac{2^i}{2c+2} \cdot (1+\frac{1}{c}) = \frac{2^i}{2c} \leq \frac{2^{i-1}+1}{c} \leq |o|/c$. The claim follows. □

We have shown that A_C's budget is always sufficient for relocating objects. It remains to show that it is always possible to allocate an object by finding an empty chunk of the right size, or by finding a chunk whose density is small enough, so that A_C can use it for allocation after relocating objects from it. In fact, it remains to show that for any program $P \in \mathscr{P}(M,n)$ that is served by the memory manager A_C and whose size at the beginning of an allocation is $2^{i-1} < |o| \leq 2^i$, there exists a 2^i-sized chunk that is either empty or has density of at most $\frac{1}{2c+2}$.

We start with the following claim stating the conditions under which an 2^i object cannot be placed in Area j. The claim is later used to show (by way of contradiction) that A_C cannot fail to allocate a chunk because the implied conditions cannot occur.

Claim 5.5.5. *For each pair of integers i, j such that $0 \leq j \leq i$, let Area S be a memory region of size divisible by 2^i, with allocated objects on it that satisfy the following conditions:*

1. *Each object o residing on S has size $|o| \geq 2^{j-1}+1$.*
2. *Each object o residing on S starts at an address divisible by $2^{\lceil \lg |o| \rceil}$.*
3. *For each chunk D of size 2^i of S, either*
   (a) *D is uncrossed and is at least $2^i/(2c+2)$ populated, or*
   (b) *an object o' of size $\geq 2^i+1$ intersects D.*

Then the accumulated size of allocated objects residing on Area S is at least

$$\begin{cases} S \cdot \max\left(2^{-i}, 1/(2c+2)\right) & \text{for} \quad j=0 \\ S \cdot \max\left(\frac{2^{j-1}+1}{2^i}, \frac{1}{2c+2}\right) & \text{for} \quad 0<j<i \\ S \cdot \frac{2^i+1}{2^{i+1}} & \text{for} \quad j=i \end{cases}.$$

Proof. We partition the set of 2^i chunks in area S into two sets: the chunks where Property (3b) holds (the crossed chunks), and chunks where Property (3a) holds (the uncrossed chunks). Define the *alive-size* of a 2^i-chunk as follows. For an uncrossed chunk, the alive-size equals the total size of objects that reside in the chunk. For a crossed chunk, let o be the object that intersects the chunk. The alive-size of the chunk is $|o|$ divided by the number of chunks that o intersects.

Next consider the summation of the alive-size over all 2^i chunks in S. By property (2), an object of size $\leq 2^i$ resides on a single 2^i chunk, and is counted only once in the summation. An object of size $\geq 2^i+1$ is also counted once in the summation: if the object intersects k chunks, each chunk counts $|o|/k$, so the total sum is $|o|$.

The alive-size for crossed chunks is at least $\frac{2^i+1}{2}$; the minimum is obtained when a single object of size 2^i+1 intersects two chunks.

Next we compute the minimum alive-size for the three cases $j=0, 0<j<i, j=i$ for uncrossed chunks. By property (3a), the alive-size for uncrossed chunks is at least $2^i(2c+2)$.

j=0 At least 1 word per chunk is alive, so the alive-size is at least $\max\left(1, 2^i/(2c+2)\right)$.

0<j≤i By Property (1) and Property (3a), every uncrossed chunk contains at least one object of size $\geq 2^{j-1}+1$. Thus, the alive-size is at least $\max\left(2^{j-1}+1, \frac{2^i}{2c+2}\right)$.

By the claim assumption $|S|$ is divisible by 2^i, so Area S contains exactly $|S|/2^i$ chunks, and by Property 3 each chunk is either crossed or uncrossed. The total size of live objects in S is at least:

j=0 $|S|/2^i \cdot \min\left(\frac{2^i+1}{2}, \max\left(1, 2^i/(2c+2)\right)\right) = |S| \cdot \max\left(2^{-i}, 1/(2c+2)\right)$;

0<j<i $|S|/2^i \cdot \min\left(\frac{2^i+1}{2}, \max\left(2^{j-1}+1, \frac{2^i}{2c+2}\right)\right) = |S| \cdot \max\left(\frac{2^{j-1}+1}{2^i}, \frac{1}{2c+2}\right)$;

j=i $|S|/2^i \cdot \min\left(\frac{2^i+1}{2}, \max\left(2^{j-1}+1, \frac{2^i}{2c+2}\right)\right) = |S| \cdot \frac{2^i+1}{2^{i+1}}$, □

as required.

We emphasize that we do not pad objects to the next "power-of-two" size. An object that spans three 2^i-chunks does not reserve the fourth chunk. Another 2^i object can be placed in the remaining vacant 2^i-chunk.

In the following claim we show that if there is no empty chunk, and no chunk with small density, then the total size of simultaneously allocated objects must exceed M, in contradiction to the program being in $\mathscr{P}(M,n)$.

Claim 5.5.6. *Upon an allocation request of size $2^{i-1} < |o| \leq 2^i$ by a program $P \in \mathscr{P}(M,n)$ of the memory manager A_C, there exists a chunk of size 2^i that is either vacant or uncrossed and whose density is less than $\frac{1}{2c+2}$. The claim holds for an allocation request made by a program $P \in \mathscr{P}(M,n)$, or recursive calls made by A_C while serving such a program P.*

Proof. Assume by way of contradiction that there exists no empty nor uncrossed chunks with density $< \frac{1}{2c+2}$. We will show that under this assumption, the total size of live memory, including the newly allocated object o, exceeds M.

First, consider the case $i = 0$ ($|o| = 1$). Since there are M chunks of size 1, and none of which are empty, the total size of live objects at area 0 is at least M. But o is also alive, so the total size of live memory exceeds M, yielding a contradiction.

Next, let $i \geq 1$ and consider a direct call to allocation (as oppose to recursive calls made by the allocator itself). To show a contradiction, we use Claim 5.5.5 to compute the accumulating sum of live memory that must be located in Area j, for every $j \leq i$. We then sum up over the areas to show that the total size of live memory exceeds M.

Consider an area j for $j \leq i$. By definition of A_C, the size of each area a_i is divisible by n, and Properties (1) and (2) of Claim 5.5.5 hold. By our assumption, Property 3 also holds. Thus, the amount of live space at area $j < i$ is at least $\max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1}+1 \rfloor}{2^j}\right) a_j$, and the amount of live space at area i is at least $\frac{2^i+1}{2^{i+1}} \cdot a_i$.

By definition of a_i, the following holds (even with equality):

$$\frac{2^i+1}{2^{i+1}} \cdot a_i + \sum_{j=0}^{i-1} \max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1}+1 \rfloor}{2^j}\right) \cdot a_j \geq 1. \tag{5.7}$$

The above arguments, in addition to multiplying the last equation by M, prove that the total size of live memory in areas $0, 1, \ldots, i$ is at least M. Contradiction follows since the newly allocated object o is also alive, and thus the total size of live objects exceeds M.

Finally, consider a recursive call to A_C for allocating an object o'. The case $|o'| = 1$ is similar to the non-recursive call (there is always room for an object of size 1), so assume $|o'| > 1$. Let o be a newly allocated object placed at a uncrossed, sparsely populated chunk D, and let o' be the object that A_C moves from D (causing the recursive call to A_C). At this point we would like to apply Claim 5.5.5. However, the need to allocate an object and move existing objects complicates the proof. Thus, consider a snapshot of the heap just before o was allocated and consider the following modifications: remove all objects that reside on D and place o there. These modifications creates an auxiliary heap (that never existed in the execution). Still, this change does not break Properties (1)-(3) of Claim 5.5.5.

Now apply Claim 5.5.5 on this auxiliary heap and an attempt to allocate o'. As in the non-recursive argument, we get that the total live space is at least M. Since we applied Claim 5.5.5 on the auxiliary heap, this counts o but does not count o' (and all other objects that resided on D). Since both o and o' are alive, the total live memory is at least $M + |o'|$, a contradiction in any case that the claim does not hold, from which proof of the claim follows.

Claim 5.5.6 finishes the correctness proof of the memory manager. Together with Lemma 5.5.3, this concludes the proof of Theorem 5.2.

5.6 A Simpler Relaxed Upper Bound Expression

The results of Claim 5.5.3 provide an upper bound on heap size that A_C uses. The upper bound is presented as a sum over a recursive series a_i. In this section we show how to approximate this upper bound by an explicit expression (with no recursion).

We change the area sizes that A_C uses to a new series a'_i, which is simpler to analyze. We first show that the correctness proof of A_C still holds. Next, we show that the new series a'_i can be bounded by a geometric series, and so can be stated explicitly (and not only in a recursive form).

Define a'_i to be the recursive series where $a'_0 = 1$ and for $i \geq 1$:

$$a'_i = 2 - \sum_{j=1}^{i-1} \max\left(\frac{1}{c+1}, 2^{j-i}\right) \cdot a'_j - \max\left(\frac{1}{c+1}, 2^{-i+1}\right) \cdot a'_o \tag{5.8}$$

The explicit area sizes a_i appear in the correctness proof of A_C only in Equation 5.7, in the proof of Claim 5.5.6. Thus, it is sufficient to show that our new series satisfies:

$$\frac{2^i+1}{2^{i+1}} \cdot a'_i + \sum_{j=0}^{i-1} \max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1}+1 \rfloor}{2^i}\right) \cdot a'_j \geq 1.$$

If we substitute the new area sizes into Claim 5.5.6, the correctness proof of A_C that uses area sizes a'_i follows.

Rearranging the terms in Equation 5.8 by moving all terms (except the 2) to the left side and dividing by 2, we get

$$\frac{1}{2}a'_i + \sum_{j=1}^{i-1} \frac{1}{2} \max\left(\frac{1}{c+1}, 2^{j-i}\right) \cdot a'_j + \frac{1}{2} \max\left(\frac{1}{c+1}, 2^{-i+1}\right) \cdot a'_o = 1.$$

Also, by Proposition 5.6.2 below we prove that a'_i can be rewrite as a positive sum of $a'_j : j < i$. Thus, a_i is always positive as it is a sum of positive terms. Now, Inequality 5.7 for the series a' follows:

$$\begin{aligned}
& \frac{2^i+1}{2^{i+1}} \cdot a'_i + \sum_{j=0}^{i-1} \max\left(\frac{1}{2c+2}, \frac{\lfloor 2^{j-1}+1 \rfloor}{2^i}\right) \cdot a'_j \\
= \; & \frac{2^i+1}{2^{i+1}} \cdot a'_i + \sum_{j=1}^{i-1} \max\left(\frac{1}{2c+2}, \frac{2^{j-1}+1}{2^i}\right) \cdot a'_j + \max\left(\frac{1}{2c+2}, \frac{1}{2^i}\right) \cdot a'_0 \\
\geq \; & \frac{1}{2} \cdot a'_i + \sum_{j=1}^{i-1} \max\left(\frac{1}{2c+2}, \frac{2^{j-1}}{2^i}\right) \cdot a'_j + \max\left(\frac{1}{2c+2}, \frac{1}{2^i}\right) \cdot a'_0 \\
= \; & \frac{1}{2}a'_i + \sum_{j=1}^{i-1} \frac{1}{2} \max\left(\frac{1}{c+1}, 2^{j-i}\right) \cdot a'_j + \frac{1}{2} \max\left(\frac{1}{c+1}, 2^{-i+1}\right) \cdot a'_o \\
= \; & 1.
\end{aligned}$$

The inequality at the third line follows by the fact that a'_i are positive. The equality at the fifth line follows by Equation 5.8 rearranged.

To keep the mathematical analysis clear, we assume that $c+1 = 2^k$ for some integral k. If c does not satisfy this assumption, rounding it up only decrease the compaction budget available to the memory manager.

Claim 5.6.1. *Let A_C be the memory manager from Algorithm 5.3 that uses area sizes a'_i. Then for any program $P \in \mathscr{P}(M,n)$, the heap size used by A_C is bounded by*

$$\begin{aligned} HS(A_C,P) &\leq k+1+\frac{1-\left(1-2^{-k-1}\right)^{\lg n-k}}{1-(1-2^{-k-1})}M \\ &\leq \left(\lg n+1-\frac{\lg^2(n/(c+1))}{4c+4}+\frac{\lg^3(n/(c+1))}{6(2c+2)^2}\right)M. \end{aligned}$$

We start by bounding area sizes. We first consider area sizes for $i \leq k$, and show that $a'_i = 1 : i \leq k$. For $i = 0$, the equality $a'_0 = 1$ follows by the series a' definition. For $0 < i \leq k$, we show $a'_i = 1$ by induction. Let the induction basis be $i = 0$, and for the induction step suppose that $a'_j = 1$ for all $j < i$. Thus:

$$a'_i = 2-\left(\sum_{j=1}^{i-1} 2^{j-i}\cdot 1\right)-2^{-(i-1)} = 2-\left(\sum_{j=1}^{i-1} 2^{-j}\right)-2^{-(i-1)} = 1.$$

The first equality follows by area a' definition and the induction assumption. The second equality arranges the summation terms.

Bounding area sizes for $i > k$ is harder. We start by expressing the series a'_i as a simpler recursive series that contains only additive terms. Recursive series that use only additive terms are easier to reason about since they depend positively on previous terms.

Proposition 5.6.2. *Let a'_i be the series from Equation 5.8, and let $f(a'_{i-1},a'_{i-2},\ldots,a'_{i-k}) = \sum_{j=i-k}^{i-1} 2^{j-i}\cdot a'_j$. Then for $i > k$ the following holds:*

$$a'_i = f(a'_{i-1},a'_{i-2},\ldots,a'_{i-k}) = \sum_{j=i-k}^{i-1} 2^{j-i}\cdot a'_j.$$

Proof. We start with the following auxiliary equation:

$$\begin{aligned}
& a'_i - a'_{i-1} \\
= \; & (2 - \sum_{j=0}^{i-1} \max(2^{-k}, 2^{j-i}) \cdot a'_j - 2^{-k} \cdot a'_0) - (2 - \sum_{j=0}^{i-2} \max(2^{-k}, 2^{j-i+1}) \cdot a'_j - 2^{-k} \cdot a'_0) \\
= \; & - \sum_{j=i-k}^{i-1} 2^{j-i} \cdot a'_j - \sum_{j=0}^{i-k-1} 2^{-k} \cdot a'_j + \sum_{j=i-k-1}^{i-2} 2^{j-i+1} \cdot a'_j + \sum_{j=0}^{i-k-2} 2^{-k} \cdot a'_j \\
= \; & -\frac{1}{2} a'_{i-1} + \sum_{j=2}^{k} (2^{-j+1} - 2^{-j}) a'_{i-j} \\
= \; & -\frac{1}{2} a'_{i-1} + \sum_{j=2}^{k} 2^{-j} a'_{i-j}.
\end{aligned}$$

The proof follows from the auxiliary equation by:

$$a'_i = a'_{i-1} + (a'_i - a'_{i-1}) = a'_{i-1} - \frac{1}{2} a'_{i-1} + \sum_{j=2}^{k} 2^{-j} a'_{i-j} = \sum_{j=1}^{k} 2^{-j} a'_{i-j},$$

as required. □

We now show that the series a'_i is bounded from above by a geometric series. Thus, the sum of the a'_i series can be bounded by the sum of the geometric series, which is analytically computable.

Proposition 5.6.3. *Let a'_i be the series defined in Equation 5.8. Furthermore, let $q = 1 - 2^{-k-1} = 1 - \frac{1}{2c+2}$, and let q_i be the geometric series defined by $q_i = q^{i-k}$. Then*

$$\forall i \geq 0 : a'_i \leq q_i = q^{i-k}.$$

Proof. Let f be the function defined in Proposition 5.6.2. By Proposition 5.6.2, $a'_i = f(a'_j : j < i)$, and f contains only additive terms. To upper bound a'_i, we use the monotonicity of f, meaning that the larger $a'_j : j < i$ are, the larger a'_i will be. It is sufficient to show that $q_i \geq f(q_{i-1}, q_{i-2}, \ldots, q_{i-k})$. The statement then follows by induction since:

Induction base: $q_i \geq a'_i : i \leq k$ follows since $q^{i-k} \geq q^0 = 1 = a'_i$.

Induction step: Assume that $\forall j < i : q_j \geq a'_j$. Then $q_i \geq f(q_{i-1}, q_{i-2}, \ldots, q_{i-k}) \geq f(a'_{i-1}, a'_{i-2}, \ldots, a'_{i-k}) = a'_i$. The first inequality follows by our assumption. The second inequality follows by the monotonicity of f and the induction assumption.

$$\begin{aligned}
f(q_{i-1}, q_{i-2}, \ldots, q_{i-k}) &= \sum_{j=1}^{k} 2^{-j} q_{i-j} \\
&\leq \sum_{j=1}^{k} 2^{-j} q^{i-j-k} \\
&= q^{i-k} \sum_{j=1}^{k} (2q)^{-j} \\
&= q^{i-k} \cdot \frac{1}{2q} \cdot \frac{1 - \frac{1}{(2q)^k}}{1 - \frac{1}{2q}} \\
&= q^{i-k} \cdot \frac{1 - 2^{-k} q^{-k}}{2q - 1} \\
&\leq q^{i-k} \cdot \frac{1 - 2^{-k}}{2q - 1} \\
&= q^{i-k} \cdot \frac{1 - 2^{-k}}{2(1 - 2^{-k-1}) - 1} \\
&= q^{i-k} \cdot \frac{1 - 2^{-k}}{1 - 2^{-k}} \\
&= q_i.
\end{aligned}$$

This concludes the proof of Proposition 5.6.3. □

We now finish with the proof of Claim 5.6.1.

Proof. The size of area $a'_0, \ldots, a'_k$ is $k+1$. We bound the total size of areas $a'_{k+1}, \ldots, a'_{\lg n}$ by summing the geometric series q_i.

$$\begin{aligned}
\sum_{i=k+1}^{\lg n} a'_i \cdot M &\leq \sum_{i=k+1}^{\lg n} q_i \cdot M \\
&\leq \frac{1 - \left(1 - 2^{-k-1}\right)^{\lg(n)-k}}{1 - (1 - 2^{-k-1})} M \\
&= 2^{k+1} \cdot \left(1 - \left(1 - 2^{-k-1}\right)^{\lg(n)-k}\right) M \\
&= (2c+2) \cdot \left(1 - \left(1 - \frac{1}{2c+2}\right)^{\lg(n)-k}\right) M \\
&\leq (2c+2) \left(1 - e^{-\frac{\lg(n)-k}{2c+2}}\right) M \\
&\leq \left(\lg(n) - k - \frac{(\lg(n)-k)^2}{2(2c+2)} + \frac{(\lg(n)-k)^3}{6(2c+2)^2}\right) M \\
&\leq \left(\lg(n) - k - \frac{\lg^2(n/(c+1))}{2(2c+2)} + \frac{\lg^3(n/(c+1))}{6(2c+2)^2}\right) M.
\end{aligned}$$

The fourth inequality follows by the Taylor expansion to e^{-x}. Explicitly, $1-e^{-\frac{\lg(n)-k}{2c+2}} \leq \frac{\lg(n)-k}{2c+2} - \frac{(\lg(n)-k)^2}{2\cdot(2c+2)^2} + \frac{(\lg(n)-k)^3}{6\cdot(2c+2)^3}$. The last inequality follows by setting $k=\lg(c+1)$ and the mathematical equation $\lg(a)-\lg(b)=\lg(a/b)$.

The proof of the claim follows by summing the total size of areas $0,\ldots,k,k+1,\ldots,\lg n$.□

5.7 Measurements and Discussion

While this work advances our knowledge about partial compaction significantly, it still leaves a substantial gap between the obtained upper and lower bounds. As an interesting anecdotal test, we ran the bad program P_F from Section 5.4 against the good allocator A_C from Section 5.5. While such an experiment provides no rigorous information, it may shed some light on the direction in which this gap can be narrowed. We ran both programs with the following parameters: $M = 256$MB, $n = 1$MB, and $c = 50$. The result of this execution yielded a memory usage of $4.85M$. For this case, our proof shows an upper bound of $19M$ and a lower bound of $3.15M$. In what follows we discuss these results.

The first thing to note is that in the experiment A_C performed no compaction. The reason is that A_C actually compacts objects only if the density of a chunk drops below $\frac{1}{2c+2}$. But P_F never lets this happen. It never frees objects if the result is a chunk with density smaller than $2^{-\gamma} \geq \frac{4}{3c} > \frac{1}{2c+2}$. This drawback of A_C may indicate it can be improved by developing an algorithm that is able to compact memory even if a chunk density is higher than $\frac{1}{c}$. Such an allocator would be more complex than our A_C because the allocator cannot consistently compact such locations. It doesn't have enough budget for that.

The lack of compaction makes the first stage of P_F more effective than the second stage. Recall that the first stage uses Robson's program that ignores compaction while the second stage tries to handle compaction while still creating fragmentation. Indeed, the experiment shows that the first phase alone made the allocator use a space of $3.5M$ fragmentation, and the second phase only increased it by $1.35M$.

Another inaccuracy that builds the gap between the upper and lower bound is aligned allocation. The allocator A_C places objects in an aligned manner, i.e., it puts an object of size 2^i in an address $k\cdot 2^i$ for some $k \in \mathbb{N}$. However, the bad program P_F does not use this "weakness" of the allocator. P_F attempts to handle the general case in which allocation is not aligned, thus allocating objects that are 4 times larger than a chunk and considering only three-quarters of the object which span a full chunk. Closing the gap would also require building an allocator that places objects in non-aligned location to neutralize the bad behavior of a program and provide a better upper bound. Alternatively, maybe one can show that running a bad program against a non-aligned allocator is equivalent (up to a small factor) to running the program against an aligned allocator. We do not know how to show that.

Finally, we noticed that when we increased the value of x in P_F (Line 1) we got better results (more fragmentation). In this case, the heap usage was increased to $5.56M$. However, we do not know how to analyze an execution with such an x. Some of the iterations created a lot of

fragmentation while other created almost zero fragmentation. We do not know how an increase of x will behave when P_F is run against a better (or optimal) allocator.

Chapter 6

Conclusion

A major contribution of this dissertation is the introduction of optimistic memory management approaches. Optimistic approaches, where a thread tries to apply an operation and then reverses its decision, are widely used in concurrent algorithms. However, due to the conservative nature of memory managers, optimistic approaches have not until now been used for memory management: touching reclaimed objects or relying on a programmer's hints are considered unsafe.

We have shown in this dissertation, however, that optimistic approaches can be very beneficial for memory management. It is possible to build a memory manager even when allowing threads to touch reclaim objects, or even when the memory manager (optimistically) relies on programmer hints. Furthermore, such optimistic approaches can yield significant performance improvements.

On the practical side, we presented a memory management scheme for lock-free data structure that preserves the lock-freedom of the algorithm. This scheme is highly efficient, scalable, and can be applied automatically. As far as we know, this is the first memory management scheme that allows accessing reclaimed objects in order to obtain (considerable) performance improvement. We also presented a garbage collection extension that allows the garbage collector to use programmer hints about objects used by the application. The resulting garbage collection is safe: wrong programming hints cannot lead to incorrect results. Correct hints, however, significantly improve the scalability and efficiency of the garbage collector.

We also considered theoretical bounds on fragmentation. This work contributes to building a solid theoretical foundation for memory management by extending previous work to providing new lower and upper bounds on the effectiveness of partial compaction. Our lower bound is the first bound in the literature with implications for existing systems because it shows that some desirable (realistic) compaction goals cannot be achieved.

The main question left open by this dissertation is the viability of practical lock-free garbage collection [HM92]. While the *automatic optimistic access* scheme presented in this dissertation works on data structures only, such lock-free garbage collection will work for any application in its general form.

Appendix A

Appendix

A.1 A Formal Definition of Normalized Data Structures

In this section we provide the formal definition of normalized data structure implementations [TP14]. In fact, we relax the original definition a bit, because the optimistic access and the automatic optimistic access schemes can also work with data structures that do not satisfy all the constraints of the original definition. Of-course, the (automatic) optimistic access memory scheme will work with the original definition of [TP14] as well, but the relaxed restrictions that we specify below suffice.

Definition 4.1 of [TP14] (slightly relaxed): A lock-free algorithm is provided in a normalized representation if:

- Any modification of the shared data structure is executed using a CAS operation.
- Every operation of the algorithm consists of executing three methods one after the other and which have the following formats.
    1. CAS Generator: its input is the operation's input, and its output is a list of CAS descriptors. CAS descriptors are tuples of the form (address, expectedVal, newVal). This method is parallelizable (see Definition 3.4 of [TP14] below).
    2. CAS Executor, which is a fixed method common to all data structures and all algorithms. Its input is the list of CAS descriptors output by the CAS generator method. The CAS executor method attempts to execute the CASes in its input one by one until the first one fails, or until all CASes complete. Its output contains the list of CAS Descriptors from its input and the index of the CAS that failed (which is zero if none failed).
    3. Wrap-Up, whose input is the output of the CAS Execution method plus the operation's input. Its output is either the operation result, which is returned to the owner thread, or an indication that the operation should be re-executed from scratch (from the Generator method). This method is parallelizable (see Definition 3.4 of [TP14] below).

We remark that in [TP14] there is an additional requirement for a contention failure counter and for versioning that we do not need for our construction. This makes the above definition more relaxed. Of-course, the proposed memory management scheme will work well also when the data structure representation adheres to the full (stricter) definition of [TP14].

As discussed in [TP14], any data structure has a normalized lock-free implementation, but not necessarily efficient. However, interesting data structures had a small (often negligible) overhead.

We provide the definition of parallelizable methods below. An important property of parallelizable methods is that at any point during their execution, it is correct to simply restart the method from its beginning (with the same input). We will use this fact by starting the CAS generator and the wrap-up methods from the beginning whenever we detect stale values that were read optimistically following a concurrent reclamation.

The lock-free memory management algorithms we propose are optimistic. Optimistic algorithms typically execute instructions even when it is not clear that these instructions can be safely executed. In order to make sure that the eventual results are proper, optimistic algorithms must be able to roll back inadequate execution, or they stop and check before performing any visible modification of the shared memory that cannot be undone. In the (automatic) optimistic access scheme, rolling back is done by restarting from the beginning of the method (Generator or Wrap-Up). In the rest of this section, the instruction *restart* refers to restarting from the beginning of the currently executed method (which will always be the Generator or the Wrap-Up).

To simplify the discussion of a restart, we assume that the CAS generator and wrap-up methods do not invoke methods during their execution. The reason is that if a method is invoked by the CAS generator (for example), restarting the CAS generator from scratch, when a check in the invoked method detects stale values, implies returning from all invoked methods, removing their frames from the runtime stack without executing them further. Note first that this is achievable by letting the invoked routines return with a special code saying that a restart is required and the calling method should act accordingly. Note also that inlining can be applied to create a method that does not invoke other methods if no recursion is used and no virtual calls exist.

Additionally, we assume that the executions of the original data structure (with no memory management) do not trigger a trap. Adding checks and handling restarts in a trap code are more involved and are outside the scope of the current paper.

Before starting with the definition of parallelizable methods, we start by defining an avoidable execution of a method (3.3 of [TP14]). An avoidable execution of a method is an execution that can be rolled back. Such an execution should not perform any modification visible by other threads, otherwise it cannot be rolled back. For completeness we also provide Definition 3.1 and Definition 3.2 of [TP14] below.

Definition 3.1 of [TP14] (Futile CAS) *A futile CAS is a CAS in which the expected value and the new value are identical.*

Definition 3.2 of [TP14] (Equivalent executions) *Let I_1 and I_2 be two (different) implementations of a data structure D. Let E_1 be an execution over I_1 and let E_2 be an execution over I_2. Then the executions are* equivalent *if the following hold:*

Results: *In both executions all threads execute the same data structure operations and receive identical results.*

Relative Operation Order: *The order of invocation points and return points of all data structure operations is the same in both executions.*

Definition 3.3 of [TP14] (Avoidable method execution) *A run of a method M by a thread T on input I in an execution E is avoidable if each CAS that T attempts during the execution of M is avoidable in the following sense. Let S_1 denote the state of the computation right before the CAS is attempted by T. Then there exists an equivalent execution E' for E such that both executions are identical until reaching S_1, and in E' the CAS that T executes in its next step (after S_1) is either futile or unsuccessful. Also, in E' the first execution step from S_1 is executed by a thread who is the owner of an ongoing operation.*

We now recall the definition of parallelizable methods, i.e., Definition 3.4 of [TP14] and then explain why it implies that restarting from the beginning of the method is fine.

Parallelizable method (Definition 3.4 of [TP14]). *A method M is a parallelizable method of a given lock-free algorithm, if for any execution in which M is called by a thread T with an input I the following two conditions hold. First, the execution of a parallelizable method depends only on its input, the shared data structure, and the results of the method's CAS operations. In particular, the execution does not depend on the executing thread's local state prior to the invocation of the parallelizable method. Second, at the point where M is invoked, if we create and run a finite number of parallel threads, each one executing M on the same input I concurrently with the execution of T, then in any possible resulting execution, all executions of M by the additional threads are avoidable.*

Now we show that a parallelizable method can be restarted from scratch. Consider an execution where a thread T restarts a method M a finite number of times, each time with the same input. The method execution does not depend on the thread's local state, so there exists an equivalent execution where each invocation is executed by a different (auxiliary) thread. Only the last invocation (which never restarts) is considered executed by T, which is the owner thread for the ongoing operation. The auxiliary threads do not finish to execute the method. Thus, consider these threads as crashing at the point where a restart is executed. By definition of parallelizable method, there exists an equivalent execution where the auxiliary threads execution of M is avoidable. But then the auxiliary threads execute no modification observable by other threads, which is equivalent to having T execute the method alone with no restarts.

A.2 Running Example for the Optimistic Access Scheme

In this section we complete applying the optimistic access scheme on the running example of Listing 2.1 in Chapter 2 (Page 17). For simplicity of presentation and ease of reading, we wrap the warning bit check inside a macro.

Listing A.1 Applying the optimistic access scheme

```
#define CHECK {__compiler_fence(); \
    if(unlikely(*warning)) {*warning=0; goto start;}}
#define FENCE_CHECK { \//A CAS serves as a memory barrier in x86
    if(unlikely(CAS(warning,1,0))) goto start;}

bool delete(int sKey, Node *head, *tail);
descList CAS_Generator(int sKey, Node *head, *tail){
    descList ret;
    char *warning=&thread.warning;
    void **HP = thread.HP;//array of hazard pointers
    start:
    while(true)  {/*Attempt to delete the node*/
        Node *prev = head, *cur = head->next, *next;
        CHECK //protect reading cur
        while(true) {            /*search for sKey position*/
            if(cur==NULL){
                ret.len=0;
                return ret;
            }
            next = cur->next;
            cKey = cur->key
            node *tmp=prev->next;
            CHECK //protect reading next,cKey,tmp
            if( tmp != cur)
                goto start;
            if(!is_marked(next)){
                if(cKey>=sKey) break;
                prev=cur;
            }
            else{
                HP[0]=prev;//Algorithm 2.2
                HP[1]=cur;
                HP[2]=unmark(next);
                FENCE_CHECK
                if( CAS(&prev->next, cur, unmark(next)) )
                        retire(cur);
                else    goto start;
            }
            cur=unmark(next);
        }
        if(cKey!=sKey){
            ret.len=0;
            return ret;
        }
        ret.len=1;
        ret.desc[0].address=&cur->next;
        ret.desc[0].expectedval=next;
        ret.desc[0].newval=mark(next);
        HP[3]=cur;//Algorithm 2.3
        HP[4]=next;
        FENCE_CHECK
        return ret; /*Return to CAS executor*/
    }
}
int WRAP_UP(descList exec, int exec_res,int sKey, Node *head, *tail){
    if(exec.len==0)    return FALSE;
    if(exec_res==1)    return RESTART_GENERATOR; /*CAS failed*/
    else                return TRUE;
}
```

Bibliography

[AAB+00] Bowen Alpern, C Richard Attanasio, John J Barton, Michael G Burke, Perry Cheng, J-D Choi, Anthony Cocchi, Stephen J Fink, David Grove, Michael Hind, S F Hummel, D Lieber, V Litvinov, M Mergen, T Ngo, J R Russell, V Sarkar, M J Serrano, J Shepherd, S Smith, V C Sreedhar, H Srinivasan, and J Whaley. The jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[ABC+08] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *EMSOFT*, pages 245–254, 2008.

[AEH+14] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *EuroSys*. ACM, 2014.

[AG09] Edward E Aftandilian and Samuel Z Guyer. Gc assertions: using the garbage collector to check heap properties. In *PLDI*, pages 235–244, 2009.

[AOPS04] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. In *OOPSLA04*, pages 224–236, 2004.

[BBYG+05] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, 2005.

[BCCO10] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP'10*, pages 257–268, may 2010.

[BCM04] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. Myths and realities: The performance impact of garbage collection. *ACM SIGMETRICS Performance Evaluation Review*, pages 25–36, 2004.

[BCR03] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL03*, pages 285–298, 2003.

[BGH+06] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA'06, pages 169–190. ACM, 2006.

[BKP13] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *SPAA*, pages 33–42. ACM, 2013.

[BM03] Stephen M Blackburn and Kathryn S McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA*, pages 344–358. ACM, 2003.

[BM08] Stephen M Blackburn and Kathryn S McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*, pages 22–32. ACM, 2008.

[Boe02] Hans-Juergen Boehm. Bounding space usage of conservative garbage collectors. In *POPL02*, 2002.

[Boe04] Hans-Juergen Boehm. The space cost of lazy reference counting. In *POPL04*, pages 210–219, 2004.

[BP10] Katherine Barabash and Erez Petrank. Tracing garbage collection on highly parallel platforms. In *ISMM*, pages 1–10. ACM, 2010.

[BP11] A. Bendersky and E. Petrank. Space overhead bounds for dynamic memory management with partial compaction. *Principles of Programming Languages*, pages 491–499, 2011.

[BP12] Anastasia Braginsky and Erez Petrank. A lock-free B+tree. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures - SPAA '12*, page 58, New York, New York, USA, jun 2012. ACM Press.

[BVEDB07] Dries Buytaert, Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Gch: Hints for triggering garbage collections. In *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 74–94. Springer, 2007.

[BYGK+02] Ori Ben-Yitzhak, Irit Goft, Elliot Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In *ISMM02*, pages 100–105, 2002.

[CP15] Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *SPAA*. ACM, 2015.

[CR06] Sigmund Cherem and Radu Rugina. Compile-time deallocation of individual objects. In *ISMM*, pages 138–149. ACM, 2006.

[CTW05] Cliff Click, Gil Tene, and Michael Wolf. The Pauseless GC algorithm. In *VEE05*, pages 46–56, 2005.

[CWZ98] Jonathan E Cook, Alexander L Wolf, and Benjamin G Zorn. A highly effective partition selection policy for object database garbage collection. *Transactions on Knowledge and Data Engineering*, 10(1):153–172, 1998.

[DFHP04] David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. Garbage-first garbage collection. In *ISMM04*, pages 37–48, 2004.

[DHLM13] Dave Dice, Mauric Herlihy, Y Lev, and M Moir. Lightweight contention management for efficient compare-and-swap operations. In *EuroPar*. ACM, 2013.

[DHY10] David Dice, Hui Huang, and Mingyao Yang. Techniques for accessing a shared resource using an improved synchronization mechanism, 2010. Patent US 7644409 B2.

[DKP00] Tamar Domani, Elliot K Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for java. In *PLDI*, pages 274–284. ACM, 2000.

[DMMSJ02] David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. Lock-free reference counting. *DISC*, pages 255–271, 2002.

[DP00] David Detlefs and Tony Printezis. A generational mostly-concurrent garbage collector. Technical report, Sun Microsystems, 2000.

[DVY14] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '14*, pages 343–356. ACM Press, feb 2014.

[EP13] Haggai Eran and Erez Petrank. A study of data structures with a deep heap shape. In *MSPC*. ACM, 2013.

[FR04] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing - PODC'04*, pages 50–59, 2004.

[Fra] K Fraser. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/src/temp/lockfree-lib/.

[GBF07] Robin Garner, Stephen M Blackburn, and Daniel Frampton. Effective prefetch for mark-sweep garbage collection. In *ISMM*, pages 43–54. ACM, 2007.

[GMF06] Samuel Z Guyer, Kathryn S McKinley, and Daniel Frampton. Free-me: a static analysis for automatic individual object reclamation. *PLDI*, pages 364–375, 2006.

[Har01] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314. Springer, 2001.

[Her91] Maurice Herlihy. Wait-free synchronization. *TOPLAS*, pages 124–149, 1991.

[HFB05] Matthew Hertz, Yi Feng, and Emery D Berger. Garbage collection without paging. In *PLDI*, pages 143–153. ACM, 2005.

[HLMM05] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Non-blocking memory management support for dynamic-sized data structures. *TOCS*, 23(2):146–196, 2005.

[HM92] Maurice P Herlihy and J Eliot B Moss. Lock-free garbage collection for multiprocessors. *TPDS*, pages 304–311, 1992.

[HM01] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM-ISCOPE Conference on Java Grande*, pages 48–57, 2001.

[HMBW07] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *JPDC*, pages 1270–1285, 2007.

[HMGJ04] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *ISMM*, pages 73–84. ACM, 2004.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, August 2011.

[Kit09] KittyCache. Kittycache. https://code.google.com/p/kitty-cache/, 2009.

[KP06a] Haim Kermany and Erez Petrank. The Compressor: Concurrent, incremental and parallel compaction. In *PLDI06*, pages 354–363, 2006.

[KP06b] Haim Kermany and Erez Petrank. The compressor: concurrent, incremental, and parallel compaction. In *PLDI*, pages 354–363, 2006.

[LP99] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for java. In *OOPSLA*, pages 367–380, 1999.

[MA15] Adam Morrison and Maya Arbel. Predicate RCU : An RCU for Scalable Concurrent Updates. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP'15*, pages 21–30, 2015.

[Mic02] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82. ACM, 2002.

[Mic04] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *TPDS*, 15(6):491–504, 2004.

[MNSS05] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures - SPAA'05*, pages 253–262, 2005.

[MRM+12] Martin Maas, Philip Reames, Jeffrey Morlan, Krste Asanović, Anthony D Joseph, and John Kubiatowicz. Gpus as an opportunity for offloading garbage collection. In *ISMM*, pages 25–36. ACM, 2012.

[MS95] Maged M Michael and Michael L Scott. Correction of a memory management method for lock-free data structures. Technical report, DTIC Document, 1995.

[NM14] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP'14*, pages 317–328, 2014.

[OBYG+02] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent gc for servers. In *PLDI*, pages 129–140, 2002.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[Pet12] Erez Petrank. Can parallel data structures rely on automatic memory managers? In *MSPC*, pages 1–1. ACM, 2012.

[PFPS07] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. Stopless: A real-time garbage collector for multiprocessors. In *ISMM*, pages 159–172, 2007.

[PPS08] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI08*, pages 33–44, 2008.

[PR02] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *POPL02*, pages 101–112, 2002.

[PZM+10] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *PLDI*, pages 146–159, 2010.

[RN13] Philip Reames and George Necula. Towards hinted collection: annotations for decreasing garbage collector pause times. In *ISMM*, pages 3–14. ACM, 2013.

[Rob71] J.M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, 18(3):416–423, 1971.

[Rob74] J.M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):491–499, 1974.

[SBYM13] Rifat Shahriyar, Stephen M Blackburn, Xi Yang, and Kathryn S McKinley. Taking off the gloves with reference counting immix. In *OOPSLA*, pages 93–110, 2013.

[SGBS02] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–306, 2002.

[Sie08] Fridtjof Siebert. Limits of parallel marking collection. In *ISMM*, pages 21–29. ACM, 2008.

[SMB04] Narendran Sachindran, J Eliot B Moss, and Emery D Berger. mc^2: high-performance garbage collection for memory-constrained environments. *OOPSLA*, 39(10):81–98, 2004.

[SPE05] SPEC. Specjbb2005. http://www.spec.org/jbb2005/, 2005.

[Sun05] Håkan Sundell. Wait-free reference counting and memory management. In *IPDPS*, pages 24b–24b. IEEE, 2005.

[TP14] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *PPoPP*, pages 357–368. ACM, 2014.

[UIY12] Tomoharu Ugawa, Hideya Iwasaki, and Taiichi Yuasa. Improvements of recovery from marking stack overflow in mark sweep garbage collection. *IPSJ Online Transactions*, 5, 2012.

[Val95] John D Valois. Lock-free linked lists using compare-and-swap. In *PODC*, pages 214–222. ACM, 1995.

[WF04] Adam Wick and Matthew Flatt. Memory accounting without partitions. In *ISMM*, pages 120–130. ACM, 2004.

[YBF+11] Xi Yang, Stephen M Blackburn, Daniel Frampton, Jennifer B Sartor, and Kathryn S McKinley. Why nothing matters: the impact of zeroing. In *OOPSLA*, pages 307–324. ACM, 2011.

לבסוף, אנו עוסקים במגבלות התאורטיות של דחיסה. כדי למנוע מצב של פרגמנטציה, אוספי זבל משתמשים בדחיסה. אבל, דחיסה של הזכרון היא פעולה יקרה שאוספי זבל מסחריים מנסים להמנע ממנה. במקום, הם משתמשים בדחיסה חלקית של הזכרון כדי למנוע מצב של ניצולת זכרון נמוכה. עבודות קודמות סיפקו תוצאות אסיפטוטיות אבל ללא משמעות למערכות אמיתיות. בעבודה זו אנו משלבים שיטות קודמות עם שיטה קודמת למקרה שאין דחיסה כלל, מה שמאפשר להרחיב את התאוריה. החסם שאנו מראים נותן תוצאות תאורטיות משמעותיות עבור פרמטרים המשמשים מערכות אמיתיות.

ניהול הזכרון הקיימות אינן מאפשרות קריאת מידע לא נכון. מערכת ניהול הזכרון גישות אופטימיות מאפשר לתוכנית לקרוא מידע לא נכון, אבל עדיין שומרת על הנכונות הכוללת של התוכנית. עיקר הפתרון הוא יצירת אלגוריתם סינכרון שמאפשר לחוט של התוכנית שהו קרא מידע לא נכון. במקרה כזה החוט יזרוק מיידית את המידע השגוי. בצורה כזו אנו מונעים מקריאת המידע השגוי להשפיע על נכונות של התוכנית.

למרות שפיתוח השיטה דורש זהירות, השימוש בשיטה הינו פשוט. אנו מימשנו שיטה זו עבור רשימה מקושרת, טבלת ערבול, ורשימה מדלגת. מדידות מראות שהשיטה החדשה יעילה באופן משמעותית משיטות ניהול זכרון קיימות.

התרומה השניה המוצגת בתזה זו היא הרחבה של שיטת ניהול זכרון שפיתחנו להיות אוטומטית, ברוח של איסוף זבל (ניהול זכרון אוטומטי) המשמש בשפות מנוהלות כגון ג'אווה. האלגוריתם המוצג עובד עם כל מבנה נתונים ללא מנעולים המוצג בצורה סטנדרטית, בלי לדרוש מהמתכנת להגדיר ידנית מתי אובייקטים מנותקים ממבני הנתונים ובלי צורך לשנות את האלגוריתם של מבנה הנתונים. כמו שיטת ניהול הזכרון האוטומטי סריקה וטיאטוא, האלגוריתם סורק מידי פעם את האובייקטים ומשחרר אובייקטים שהתוכנית אינה ניגשת אליהם. שלא כמו שיטות ניהול זכרון אוטומטי, השיטה מאפשרת מידי פעם לשחרר אובייקטים שהתוכנית ניגשת אליהם; עם זאת, זה לא גורם לחוסר נכונות כי כאשר חוט קורא מידע לא נכון הוא יזהה זאת ויזרוק את המידע השגוי. מדידות על רשימה מקושרת וטבלת ערבול מראות שהאוטומציה מגיעה בעלות נמוכה מאוד ביחס לשיטות ניהול הזכרון הידניות הקיימות.

בהמשך, אנחנו בוחנים שיטות ניהול זכרון אוטומטי הרצות על מערכת מקבילית. איסוף זבל עשוי לפעול יותר מהר אם ברשותו ידע כללי לגבי התוכנית, המצוי בידי המתכנת. אבל, רוב שפות התכנות לא מאפשרות למתכנת להעביר כזה מידע לאוסף הזבל. הבעיה העיקרית היא שלא ניתן לסמוך על המתכנת באופן מלא, כיוון שהוא עשוי להכניס באגים או קוד זדוני. אנו פותרים בעיה זו על ידי הסתמכות על המתכנת באופן חלקי, בצורה שמאפשרת להאיץ את זמן הריצה אם המידע נכון, אך לא מאפשרת למתכנת לגרום לבעיות נכונות אם המידע המועבר שגוי או זדוני.

אנו מציגים ממשק בין התוכנית ואוסף הזבל. מבחינת המתכנת מאוד קל להשתמש בממשק זה, והוא צריך להשקיע זמן קצר כדי להגיש שיפור משמעותי מבחינת יעילות ומקביליות. בפרט, אנו מתמקדים בשימוש של התוכנית במבנה נתונים (או אוספים) עבור סידור המידע בזכרון התוכנית. אנו מאפשרים למתכנת להעביר לאוסף הזבל מידע לגבי אילו מחלקות משמשות עבור צמתים של המבני נתונים, ומתי מוציאים אותם ממבנה הנתונים. אוסף הזבל המודע למבני נתונים מניח באופן אופטימי שאובייקט שלא הוצא ממבנה הנתונים עדיין חי. אנו מראים כיצד אוסף הזבל יכול להשתמש בהנחה זו כדי להאיץ את היעילות, הלוקליות, ואיזון העבודה בין חוטים. בנוסף, הנחה זו לעולם לא תגרום לאיסוף שגוי של מידע חשוב.

השתמשנו בממשק כדי להעביר לאוסף הזבל נתונים לגבי כמה אפליקציות חשובות, בהם אפליקציה לניהול חנות בעלת מספר סניפים, אפליקצית מסדי נתונים, ועוד. הניסיון מראה שהשימוש בממשק הינו פשוט מאוד ודורש שינוי קטן מאוד של האפליקציה. מדידות מראות שעבור כמה אפליקציות חשובות השימוש בממשק יכול להוריד באופן דרסטי את הזמן שנדרש לאיסוף הזבל ואפילו לשפר את זמן הריצה הכולל של האפליקציה.

תקציר

כל שפות התכנות בימנו תומכות בניהול זכרון דינאמי. התוכנית מקצה זכרון כאשר מגיע מידע חדש. בשלב מסויים התוכנית אינה צריכה יותר את הזכרון, ואז הוא משוחרר וניתן למחזר אותו כדי להכיל מידע אחר. ניהול זכרון דינאמי מקל בצורה משמעותית על כתיבת תוכניות מורכבות, וכמעט לא ניתן לראות תוכניות שלא משתמשות בזה בצורה משמעותית. אולם, תמיכה במחזור של זכרון אינה פשוטה, ויש לה עלויות, למשל מבחינת האטה של התוכניות. בתזה זו אנו מתמקדים בעלויות של מחזור זכרון, בעיקר בהקשר של המחשבים מרובה המעבדים של ימנו. כאשר מחשבים מרובה מעבדים הופכים לפופולריים, האצה של תוכניות שרצות עליהם הופכת לחשובה.

מבנה נתונים ללא נעילות משמשים ככלי עזר חשוב ביצירת תוכניות הרצות על מכונות מרובות מעבדים. מבנה נתונים כאלו מספקים הבטחת התקדמות: גם במקרה הגרוע ביותר הסנכרון בין חוטי ריצה שונים של התוכנית לא יוכל לגרום להתקעות של התוכנית. תמיד יהיה ניתן לראות התקדמות כלשהו בעבודה. באופן פרקטי, מבנה נתונים כאלו מבזבזים פחות זמן על לחכות לחוטים אחרים, ולכן הם מספקים מקביליות מעולה, זמן ריצה מהיר, ותגובות בזמן קצר. עם זאת, למרות שמבני נתונים ללא נעילות הם חשובים ומוכרים היטב, סיפוק של ניהול זכרון עבורם (בלי לפגוע בהבטחת ההתקדמות) נשארת מסובכת. שיטות קיימות, כגון שימוש במצביעי הגנה, עשויות לגרור עלות גבוהה מבחינת זמן ריצה, כלומר האטה משמעותית של המבנה נתונים. בנוסף, מאוד קשה לשלב שיטות ניהול זכרון במבני נתונים קיימים. כל שיטות ניהול הזכרון הקיימות כיום הינן ''ידניות'', מבחינה זו שהמתכנת צריך להגדיר מתי אובייקט נותק ממבנה הנתונים כך שניתן לשחרר אותו בצורה בטוחה. החלטה מתי אובייקט נותק ממבנה הנתונים אינה פשוטה בכלל ובמקרים מסויימים אפילו עשויה לדרוש שינוי של האלגוריתם המקורי.

אנו מתחילים את התזה בהצעה לשיטת ניהול זכרון חדשנית, בשם גישות אופטימיות, עבור מבני נתונים ללא נעילות. השיטה שלנו מספקת ניהול זכרון עבור מבנה נתונים ללא נעילות אשר ניתן להציגם בצורה הסטנדרטית של תימנת ופטרנק. שיטת ניהול הזכרון שלנו שוברת את האינווריאנטה המסורתית שאסור אף פעם לגשת לאובייקט אחרי ששוחרר. כלומר, השיטה מאפשרת לשחרר אובייקטים שיגשו אליהם מאוחר יותר על ידי חוטים הרצים באופן מקבילי. שבירת האינווריאטה מסורתית זו מאפשרת להשיג מקביליות עם יעילות מצויינת, אבל גם דורשת זהירות בתכנון השיטה.

הבעיה בגישה לאובייקט אחרי שהוא שוחרר היא שהמידע הקודם כבר נמחק, והזכרון מכיל מידע אחר. קריאת מידע לא נכון עשויה לגרור ריצה לא נכונה של התוכנית. לכן, כל שיטות

המחקר בוצע בהנחייתו של פרופסור ארז פטרנק, בפקולטה למדעי המחשב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי-עת במהלך תקופת מחקר הדוקטורט של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Nachshon Cohen and Erez Petrank. Limitations of partial compaction: towards practical bounds. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI'13*, pages 309–320, 2013.

Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures - SPAA'15*, pages 254–263, 2015.

Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPLSA'15*, pages 260–279, 2015.

Nachshon Cohen and Erez Petrank. Data structure aware garbage collector. *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management - ISMM'15*, pages 28–40, 2015.

תודות

ברצוני להודות למנחה שלי על ההנחיה המושקעת בדוקטורט שלי. האמונה שלך בכך שאני יכול להצליח, העזרה בהבנת מה שחשבתי, וההתעקשות על תוצאות ברורות ומלאות היו מאוד מועילות! אני גם רוצה להגיד תודה מיוחדת על הזמינות שלך - התוצאות הכי טובות התחילו מפריצה לחדר שלך עם רעיונות משוגעים חדשים. בלי לדבר על הרעיונות ולעזור לי לפתח אותם, תזה זו לא הייתה מסתיימת.

אני רוצה להודות גם להורי שנטעו בי סקרנות לבדוק ולנסות דברים מעניינים. לבסוף, אני רוצה להודות לאשתי ולילדי על שאפשרו לי את הדוקטורט.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

ניהול זכרון: מתאורייה למעשה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

נחשון כהן

הוגש לסנט הטכניון --- מכון טכנולוגי לישראל
תמוז התשע"ו חיפה יולי 2016

ניהול זכרון: מתאורייה למעשה

נחשון כהן